

2009-01-01

Improving Throughput of Simultaneous Multithreaded (SMT) Processors using Shareable Resource Signatures and Hardware Thread Priorities

Mitesh Ramesh Meswani

University of Texas at El Paso, mitesh.meswani@gmail.com

Follow this and additional works at: https://digitalcommons.utep.edu/open_etd



Part of the [Computer Sciences Commons](#)

Recommended Citation

Meswani, Mitesh Ramesh, "Improving Throughput of Simultaneous Multithreaded (SMT) Processors using Shareable Resource Signatures and Hardware Thread Priorities" (2009). *Open Access Theses & Dissertations*. 2730.
https://digitalcommons.utep.edu/open_etd/2730

This is brought to you for free and open access by DigitalCommons@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of DigitalCommons@UTEP. For more information, please contact lweber@utep.edu.

Improving Throughput of Simultaneous Multithreaded (SMT) Processors using
Shareable Resource Signatures and Hardware Thread Priorities

MITESH R. MESWANI

Department of Computer Science

APPROVED:

Patricia J. Teller, Ph.D., Chair

Sarala Arunagiri, Ph.D.

David Williams, Ph.D.

Jeanine Cook, Ph.D.

Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

Copyright ©

by

Mitesh R. Meswani

2009

Dedication

I dedicate this dissertation to my father, Ramesh Meswani, and mother, Kaumudini Meswani.

IMPROVING THROUGHPUT OF SIMULTANEOUS MULTITHREADED
(SMT) PROCESSORS USING SHAREABLE RESOURCE SIGNATURES AND
HARDWARE THREAD PRIORITIES

by

MITESH R. MESWANI

B.E.C.E., M.S.C.S

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at El Paso
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science
THE UNIVERSITY OF TEXAS AT EL PASO

December 2009

Acknowledgements

This work would not have been possible without the support, love, and guidance of many people. First and foremost, I would like to thank my advisor and committee chair, Dr. Patricia J. Teller. I realized my love for computer architecture during her class in Fall 2000. During this long journey as a Ph.D. student, she has been a fantastic source of inspiration and wisdom. I am grateful for having her as a colleague and as a friend, and without her unfaltering support and guidance this dissertation would not have been possible.

I want to thank Dr. Sarala Arunagiri for her guidance – many of her insights helped shape this dissertation – and the other members of my committee, Dr. David Williams and Dr. Jeanine Cook, for their help. Also, I want to thank all the members of the HiPerSys lab, especially Ricardo Portillo and Yipkei Kwok, for listening and carefully evaluating my research ideas and providing constructive feedback during my talks in the lab.

With respect to the experimental research described in this dissertation, I want to acknowledge that this research was conducted using equipment provided through an IBM SUR grant. For this I am grateful to Dr. Teller (PI of the grant), IBM, and UTEP (which provided a cost share). In addition, I want to thank Amir Simon from IBM for his valuable assistance in providing an operating system patch that was crucial to conduct this dissertation research.

From a very personal perspective, I want to thank my father, Ramesh Meswani, mother, Kaumudini Meswani, and brother, Vishal Meswani, for their sacrifices and support during the course of my education. And, of course, I want to thank my wife, Ingrid, for her support and encouragement. Finally, I thank God for his blessings.

Abstract

Simultaneous multithreading (SMT) allows multiple hardware threads to execute concurrently on a processor core, potentially increasing the utilization and throughput of the core by a factor of the degree of multithreading. However, such performance gains may not be achieved due to contention for resources shared by the threads. The IBM POWER5 processor, which has two hardware threads per core, provides software-controlled hardware thread priorities that control the ratio of decode cycles allocated to the two hardware threads of a core and, therefore, the degree of resource contention between them. By default, the priorities of the two hardware threads are set to equal values and, as a result, each gets half of the decode cycles. Several studies have shown that many scientific applications do not achieve best throughput at equal priorities. The best priority pair, i.e., the priority settings for a given co-schedule of two application threads that provide best throughput, depend on the characteristics of the application threads.

In this dissertation we present a methodology for predicting the best priority pair for a given co-schedule of two application threads. Our approach exploits resource-utilization information that is collected during an application thread's execution in single-threaded mode. This information provides insights about the availability of resources that are shared by threads concurrently executed in SMT mode for use by another co-scheduled application thread.

The main contributions of this dissertation are:

- (1) Demonstration of the efficacy of using non-default hardware thread priority pairs to improve SMT core throughput: Using a POWER5 simulator, we show that equal (default) priorities are not the best for 82% of the 263 application trace-pairs studied.
- (2) The concept of a “Shareable Resource Signature”: this signature characterizes an application's utilization of critical shareable SMT core resources during a specified execution time interval when executed in single-threaded mode.
- (3) A best priority pair prediction methodology: Given shareable resource signatures of an application-thread pair, we present a methodology to predict the best priority pair for the application-thread pair when co-scheduled to run in SMT mode.

- (4) An implementation and validation of the methodology for the IBM POWER5 processor, which shows that the following:
- a) 17 of 10,000 possible signatures are sufficient to characterize 95.6% of the execution times of a set of applications that consists of 20 SPEC CPU2006 benchmarks (1 data input), three NAS NPB benchmarks (3 data inputs), and 10 PETSc KSP solvers (12 data inputs). The cgs and lsqr PETSc KSP solvers have signatures that are independent of input data, while one of three NAS NPB benchmarks (bt-mz) has a signature that is independent of the input data.
 - b) For 21 co-schedules of applications, each with a signature that characterizes 95% of its execution time, our validation study shows the following:
 - i. *Predicted best priorities yield higher throughput than default priorities for all but one of the 21 co-schedules.* Initial results showed that two co-schedules (462.libquantum, 437.leslie3d) and (bt-mz.A, lu-mz.A) experience a throughput loss of 7.46% and 20.05%, respectively, at predicted priorities, as compared to that achieved at default priorities. Further investigation shows that for the co-schedule (bt-mz.A, lu-mz.A) mapping and executing the co-schedule with the predicted best priorities on hardware threads (5, 4), instead of (4, 5), results in a 3.56% higher throughput as compared to default priorities – this is in contrast to the 20.05% throughput loss experienced when executed on hardware threads (4, 5). Although we have not verified it, one possible reason for this is that the processor core favors one hardware thread over the other. Re-executing the co-schedule (462.libquantum, 437.leslie3d) on hardware threads (5,4), instead of (4, 5), results in predicted priorities yielding lower throughput than the default priorities. Thus, we claim that predicted best priorities yield equal or higher throughput than default priorities for 20 of the 21 co-schedules studied, and for the outlier the throughput loss is 7.46%.

- ii. *Using non-default priorities improves throughput.* The default priority pair yields best throughput for only six of the 21 co-schedules. For the remaining 15 the default priority pair yields throughput that is between 0.74% and 14.10% lower than that achieved with the best priority pair.
- iii. *Using the predicted best priority pair, rather than default priorities, improves throughput or at least provides throughput equal to that achieved with default priorities.* For 11 of the 21 co-schedules both the default and predicted priorities yield equal throughput. For nine of the 21 predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For two of these nine co-schedules the predicted priority pair yields a throughput improvement of less than 5%. Furthermore, for three the throughput improvement associated with executing with the predicted priority pair, rather than default priorities, is between 5% and 10% and for the other four the improvement is greater than 10%.
- iv. *Using predicted best priority pairs appears to be most applicable to floating-point “intensive” applications:* For eight co-schedules comprising applications for which the utilization of the floating-point unit exceeds that of the fixed-point unit by 10% or more, the predicted priority pairs, as compared to the default priorities, yield a throughput improvement between 3.56% and 16.42%. This result indicates that the methodology for predicting best priority pairs is most applicable to applications for which floating-point unit utilization dominates that of the fixed point unit by at least 10%.

Table of Contents

Acknowledgements.....	v
Abstract.....	vi
Table of Contents.....	ix
List of Tables	xii
List of Figures.....	xiii
1 Introduction.....	1
1.1 SMT Processor Throughput: $IPC_{\text{aggregate}}$	2
1.2 The Problem.....	3
1.3 Thesis Statement.....	5
1.4 Best Priority Pair Prediction Methodology.....	5
1.5 Dissertation Contributions	7
1.6 Dissertation Outline	9
2 Background.....	11
2.1 Introduction.....	11
2.2 Instruction Level Parallelism.....	12
2.3 Hardware Multithreading.....	13
2.4 Simultaneous Multithreading (SMT).....	16
2.5 IBM POWER5 Processor	17
2.6 SMT Implementation On Other Processors.....	26
2.7 IBM Performance Simulator for Linux on POWER	29
2.8 Operating System Support for IBM POWER5 SMT	30
3 Related Research	33
3.1 Job Scheduling.....	34
3.2 Thread Throttling.....	41
3.3 Summary.....	45
4 Pilot Simulation Study	49
4.1 Performance Study.....	49
4.2 IBM POWER5 Simulator and Hardware Thread Priorities Investigated.....	49
4.3 Benchmarks and Traces	52

4.4. Trace Generation	56
4.5 Experimental Design	59
4.6 Results.....	60
4.7 Conclusions.....	64
5 Best Priority Pair Prediction Methodology.....	66
5.1 Intuition.....	67
5.2 Phase 1: Shareable Resource Signature Generation	67
5.3 Phase 2: Prediction Framework Development	78
5.4 Phase 3: Prediction Validation.....	80
6 IBM POWER5 Implementation	84
6.1 Experimental Environment.....	84
6.2 Phase 1: Shareable Resource Signature Generation	87
6.3 Phase 2: Prediction Framework Development	124
6.4 Phase 3: Prediction Validation.....	134
6.5 Lessons Learned	141
6.6 Conclusions.....	142
7 Conclusions and Future Work	144
7.1 Contributions	144
7.2 Future Work.....	147
7.1 Contributions	155
7.2 Future Work.....	158

References.....	166
Appendix A: Utilization Validation Microbenchmarks	173
Appendix B: Signature Microbenchmarks	186
Appendix C: PETSc KSP Benchmark.....	348
Appendix D: Linux Kernel Modifications and Modules	369
Appendix E: Time Interval Data.....	374
Appendix F: Signature Data	375
Appendix G: Prediction and Validation Phase Results	376
Appendix H: Simulation Data	378
Curriculum Vita	379

List of Tables

Table 2.1: POWER5 Hardware Thread Priority Levels	23
Table 2.2: Effect of Hardware Thread Priorities on Decode Cycle Allocation.....	23
Table 3.1: Characteristics of Related Research and our Research	45
Table 3.2: Metrics used by Related Research and our IBM POWER5 Implementation.....	46
Table 4.1: Hardware Thread Priority Pairs used in Pilot Simulation Study	51
Table 4.2: SPEC CPU2000 Benchmarks used in Pilot Simulation Study	53
Table 4.3: SPEC CPU2006 Benchmarks used in Pilot Simulation Study	55
Table 4.4: Number of Instructions in Captured Traces	58
Table 4.5: Number of Trace Co-schedules associated with Different Benchmark Suite Pairs	60
Table 4.6: Comparisons of Best, Worst, and Default $IPC_{\text{aggregate}}$	64
Table 6.1: Thread Priority Pairs Considered in this Study	86
Table 6.2: Signatures Generated.....	Error! Bookmark not defined.
Table 6.3: Predictable Signature Set.....	Error! Bookmark not defined.
Table 6.4: Applications used to Evaluate Accuracy of POWER5 Prediction Methodology.....	136
Table 6.5: Prediction Accuracy Results.....	138

List of Figures

Figure 1.1: Illustration of Single-threaded and SMT Execution Modes	3
Figure 1.2: Utilization of Four Critical Shareable Core Resources by an	6
Figure 2.1: Illustration of the Utilization of Functional Units for Different Forms of Multithreading	15
Figure 2.2: Conceptual Instruction Execution Pipeline of a SMT Core with Two Hardware Threads.....	17
Figure 2.3: POWER5 Chip Layout (Source [20])	19
Figure 2.4: Instruction Execution Pipeline of a POWER5 Core (Source [10]).....	20
Figure 2.5: Instruction and Data Flow in the POWER5 Instruction Execution Pipeline of a Core (Source[10]).....	20
Figure 2.6: SPARC Core Execution Pipeline Block Diagram (Source [4])	27
Figure 2.7: Intel Netburst Microarchitecture Pipeline (Source [6])	29
Figure 4.1: Distribution of the 263Trace Co-schedules w.r.t. Best Priority Pair	62
Figure 4.2: Distribution of 263 Trace Co-schedules formed from Int2000, FP2000, Int2006, FP2006, stream2, and Imbench w.r.t. Best Priority Pair	62
Figure 4.3: Distribution of 263Trace Co-schedules w.r.t. Worst Priority Pair.....	63
Figure 5.1: Prediction via Signature Microbenchmarks	79
Figure 5.2: Best Priority Pair Prediction Methodology	83
Figure 6.1: Two-DCM p550 Configuration.....	85
Figure 6.2: POWER5 Shareable Core Resources that can be Monitored by Performance Counters.....	90
Figure 6.3: Six-cycle Pipelined Execution of Short Latency	94
Figure 6.4: Number of Cycles the Issue slot is not Available during the Execution.....	94
Figure 6.5: Main Loop of FPU-Stress Microbenchmark.....	103
Figure 6.6: Main Loop of FXU-Stress Microbenchmark	106
Figure 6.7: Main Loop of L2 Cache-Stress Microbenchmark.....	110
Figure 6.8: Accessing an Integer Array with a Stride of One Cache Line	110
Figure 6.9: Main Loop of TLB-Stress Microbenchmark.....	113
Figure 6.10: Performance Event Groups used to Measure Resource Utilization	118
Figure 6.11: Resource Utilization Histogram of Critical Resources	121
Figure 6.12: Distribution of Execution Time Intervals by Benchmark Suite.....	121
Figure 7.1: Stacked Bar Graph of Signatures found in two NAS and two SPEC Benchmarks	150
Figure 7.2: Scatter-plot of Signatures found in 2 SPEC CPU2006 and NAS NPB Benchmarks.....	151
Figure 7.1: Stacked Bar Graph of Signatures found in two NAS and two SPEC Benchmarks	161
Figure 7.2: Scatter-plot of Signatures found in 2 SPEC CPU2006 and NAS NPB Benchmarks.....	162

1 Introduction

The utilization and throughput of a processor are limited by the difference between the speeds of the processor and the memory subsystem [1]. *Simultaneous multithreading* (SMT) [1] addresses this problem by allowing two or more hardware threads of an SMT processor core (normally a CPU) to concurrently execute independent instruction streams every clock cycle. SMT allows an application (or application thread) executing on a hardware thread of a core to utilize shared resources that are left idle by application threads running on other hardware threads. In this way, SMT potentially increases processor utilization and throughput. The throughput of an SMT core can be calculated as the aggregate IPC (the average number of instructions executed per cycle) of its individual hardware threads. In the best case, throughput can increase, as compared to that of a core that does not support multithreading, by a factor equal to the number of hardware threads, i.e., the core's degree of SMT.

The hardware threads of an SMT core share most of the core's resources. Thus, two application threads, i.e., a co-schedule, executing in SMT mode on the hardware threads of a core compete for the shared core resources. The competition for these resources can potentially limit the performance benefits attributable to SMT. To address this problem, the IBM POWER5 processor [10] provides eight hardware thread priorities (0 to 7) [20], which can be used to control core resource contention.

Although the proposed methodology is applicable to application threads, in general, in this dissertation, we confine ourselves to SMT processor cores with two hardware threads (henceforth synonymous with *threads*) and co-schedules of two sequential applications, and we assume the hardware thread priority specifications of the IBM POWER5 processor. In this case, if the *hardware thread priorities* (henceforth synonymous with *thread priorities*) are set to different values that are both greater than 1, then the higher-priority hardware thread receives relatively more decode cycles than its lower-priority counterpart. This gives the application executing on the higher-priority thread the opportunity to get more work done. By default, the processor assigns equal priorities to both threads – this is called the *default priority pair*. For a given co-schedule during a specified execution time interval, we refer to the pair of hardware thread priorities that achieves best processor throughput among all available hardware thread priority pairs as the *best priority pair*.

In Chapter 4 of this dissertation we show that the default priority pair does not always yield best throughput and the best priority pair depends on the characteristics of the associated co-schedule. Hence, in this dissertation, we address the question: How can application characteristics can be used to predict the best priority pairs for a given co-schedule comprising two applications? To address this question, we propose to use *Shareable Resource Signatures* (henceforth synonymous with *signatures*) associated with intervals of an application's execution time.

The remainder of this chapter is organized as follows. First, Section 1.1 defines the throughput metric. Section 1.2 introduces the problem of resource contention in SMT processors. Section 1.3 states the hypothesis of this dissertation, while Section 1.4 presents an overview of our best priority pair prediction methodology. Section 1.5 lists the contributions made by this dissertation and, finally, Section 1.6 describes the organization of the remainder of this dissertation.

1.1 SMT PROCESSOR THROUGHPUT: $IPC_{\text{AGGREGATE}}$

SMT processor throughput can be quantified by the average number of instructions executed per cycle (IPC). Since n hardware threads can concurrently execute independent instruction streams on an SMT processor, the aggregate processor throughput is the sum of the throughputs of the individual hardware threads, i.e.,

$$IPC_{\text{aggregate}} = \sum_{i=1}^n IPC_i \quad (1.1)$$

where the IPC of Thread _{i} , IPC_i , is given by

$$IPC_i = \frac{Ins_i}{Cyc_i}, \quad (1.2)$$

where Ins_i is the number of instructions executed by Thread _{i} and Cyc_i is the number of cycles executed by Thread _{i} .

The same metrics can be used to measure the throughput of a core of an SMT processor but, in this case, n is the number of hardware threads supported by a core. For example, for a core of the IBM POWER5, $n = 2$; whereas, $n = 4$ for the processor as a whole because it has two cores.

1.2 THE PROBLEM

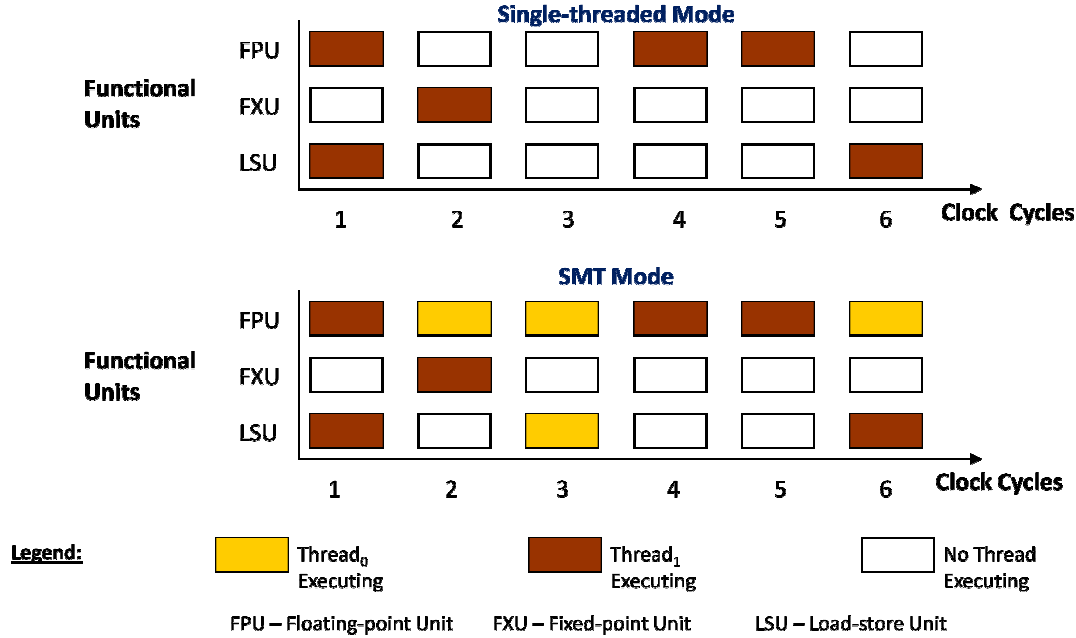


Figure 1.1: Illustration of Single-threaded and SMT Execution Modes

The top chart of Figure 1.1 illustrates the utilization of an SMT processor core's functional units by an application, Application_B, executing on Thread₁ in *single-threaded mode*, i.e., only one application is executing on the core. In contrast, the bottom chart of this figure illustrates the utilization of a co-schedule of two independent applications, Application_A and Application_B, concurrently executing in SMT mode on the two hardware threads of the core. In this figure, the utilization information is shown for six cycles of execution. In this figure, the X- and Y-axes of both charts represent clock cycles and three functional units, respectively. The three functional units shown in both charts are: the floating-point unit (FPU), fixed-point unit (FXU), and the load-store unit (LSU), respectively. For each cycle, a shaded box indicates that an application executing on a thread is utilizing the resource shown on the Y-

axis. For example, during cycle 1 of single-threaded mode execution, the application executing on Thread₁, i.e., Application_B, utilizes the FPU and LSU. As illustrated in the figure, when Application_B executes on Thread₁ in SMT mode, the application executing on Thread₀, i.e., Application_A, can use shared resources that are left idle by Application_B. For example, during cycles 2, 3, and 6, Application_A uses functional units left idle by Application_B. Thus, by allowing two applications to utilize shared core resources, processor throughput can potentially double.

However, such throughput gains are limited by both the characteristics of the applications and their contention for shared core resources, e.g., functional units, caches, and translation-lookaside buffers (TLBs). For example, referring again to Figure 1.1, consider the use of the FPU during cycles 4, 5, and 6 in SMT mode. A conflict arises if Application_A wants to use the FPU during cycles 4 and 5 since Application_B is using the FPU during these cycles. Accordingly, Application_A must wait until cycle 6 to gain access to the FPU. This example illustrates that even though SMT processors have the potential to improve processor utilization and, thus, throughput, contention for shared resources limits such gains.

The degree of contention for shared processor resources and the resultant loss of potential throughput gains depend on the characteristics of the executing applications. To address this problem, processors such as the IBM POWER5 and IBM POWER6 [10, 21] provide software-controlled hardware thread priorities. These priorities can be used to control the allocation of decode cycles to the two hardware threads, thus, potentially decreasing resource contention. If one of the threads has a higher priority than the other, it is allocated a larger proportion of decode cycles. In this way, an application executing on a thread with higher priority can, potentially, make faster forward progress than its lower-priority counterpart. The best priority pair is dependent on the characteristics of the two applications comprising the co-schedule. In this dissertation we investigate the following research question: How can application characteristics be used to predict the best priority pairs for a given co-schedule comprising two applications?

1.3 THESIS STATEMENT

The hypothesis of this dissertation can be stated formally as follows:

Assume a given co-schedule of two independent applications, $Application_A$ and $Application_B$, and an SMT processor core with two hardware threads and the software-controlled hardware thread priority specifications of the IBM POWER5 processor. In this case, we hypothesize that the usage of shareable core resources by $Application_B$ when executed in single-threaded mode may provide information concerning the availability of these resources for the use of $Application_A$ when executed concurrently with $Application_B$ in SMT mode. This information may be useful to predict the best priority pairs for the co-schedule.

1.4 BEST PRIORITY PAIR PREDICTION METHODOLOGY

We propose a best priority pair prediction methodology, described in detail in Chapter 5, which is based on the concept of Shareable Resource Signatures. A *Shareable Resource Signature*, i.e., *signature*, characterizes, for a specified execution time interval and a specific SMT processor core with software-controlled hardware thread priorities, an application's degree of utilization of the set of *critical shareable core resources* (henceforth synonymous with *critical resources*) when executed in single-threaded mode. A core's set of critical shareable core resources consists of those shareable core resources that have a significant impact on core throughput. For example, as shown in Figure 1.2, a signature could be associated with the utilization of four critical shareable core resources, e.g., *floating-point unit*, *integer unit*, *L2 cache*, and *TLB*, represented by the symbols F, I, C, and T, respectively, during an interval of one second. For this example, assume ten levels of utilization per resource, where each level represents values lying in an interval equal to 10% of the maximum possible utilization of a resource, i.e., level 1 represents utilization between 0% and 10%, level 2 represents utilization between 11% and 20%, etc. Then, for this example, during the given execution time interval, the application utilizes resource *F* at level 1, resource *I* at level 7, resource *C* at level 5, and resource *T* at level 3. In this case, the application's signature for the given execution time interval could be represented by the string

F1 I7 C5 T3, where the letters represent the critical shareable core resources and the numbers represent the associated levels of utilization.

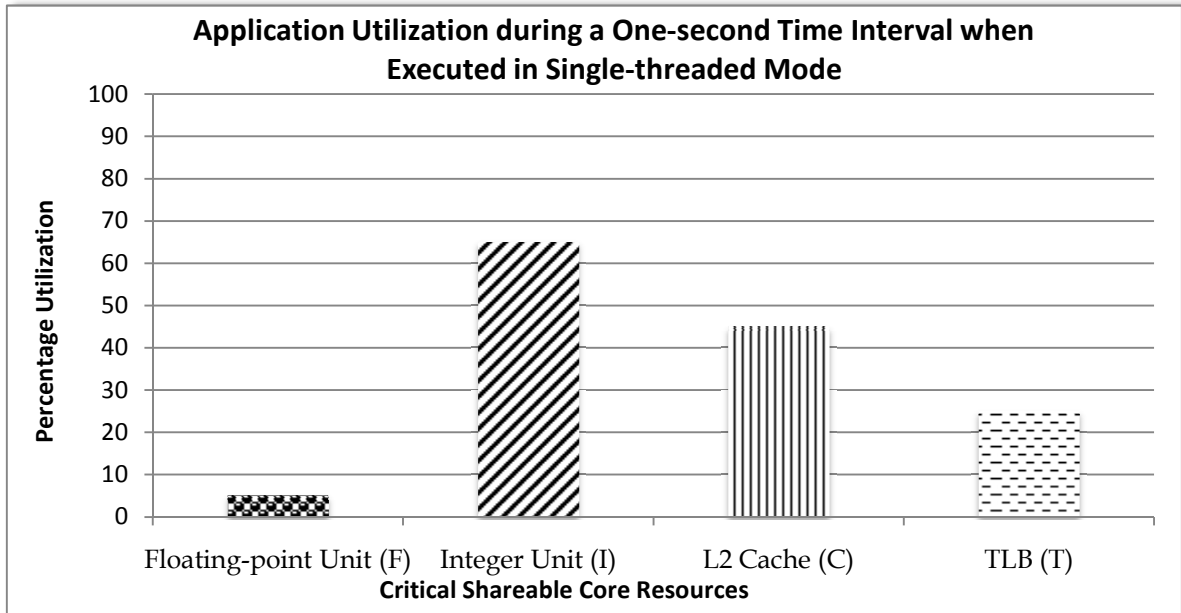


Figure 1.2: Utilization of Four Critical Shareable Core Resources by an Application during a One-second Time Interval

Given a processor architecture that supports software-controlled hardware thread priorities and a set of target applications, using our methodology, described in Chapter 5, first signatures of the set of target applications are created and then best priority pair predictions are generated for every pair of the signatures in the resultant signature set. Accordingly, these predictions are restricted to co-schedules comprising applications that have signatures in this set. Also, since, at present, only the IBM POWER5 and IBM POWER6 processors [10, 21] support software-controlled hardware thread priorities, our methodology can be used only on IBM POWER5 and IBM POWER6 processors – but it can be considered for next-generation processors that adopt such support.

For signature pairs, within the generated signature set, that characterize either co-schedules of program segments or the majority of the execution times of two applications, our methodology can be used to predict the best priority pairs. To validate the implementation of the methodology on the IBM POWER5 we fix the predicted best priority pair at the start of the execution of a co-schedule and it is

never changed. As a result, in the implementation, we can predict only for co-schedules with signature pairs that characterize the majority of the execution times of two applications, 95% in the case of the IBM POWER5. Nonetheless, our methodology also can be used for applications with multiple signatures. However, for such applications a mechanism must be developed that is able to identify changes in signatures during application execution and dynamically adapt priorities. Although we do not provide a proof in this dissertation, we conjecture that contiguous program segments that are characterized by the same signature correspond to a program phase of execution.

1.5 DISSERTATION CONTRIBUTIONS

This dissertation makes the following contributions:

1. Demonstration that judicious setting of hardware thread priorities can be used to improve SMT processor throughput: Using a POWER5 simulator, this dissertation shows that equal (default) priorities are not the best for 82% of the 263 co-scheduled applications trace-pairs.
2. The concept of a “Shareable Resource Signature”: For an application executing in single threaded mode and for a specified execution time interval, the shareable resource signature characterizes the application’s utilization of critical shareable core resources. Note that late in this research, we found that our notion of an application signature is quite similar to the notion of a base vector proposed by [27].
3. A three-phase best priority pair prediction methodology: Assume that the signature pair characterizes either a co-schedule of program segments of two applications or the majority of the execution times of two applications, 95% in the case of the IBM POWER5. In these cases, the proposed methodology can predict best priority pairs for the co-schedule of program segments or the entire execution times of the two applications, respectively. Implementation of the methodology results in a prediction table that contains a predicted best priority pair for each signature pair in the table. The coverage of the signature pairs in the table depends on the set of applications used in the implementation of the methodology.

4. An implementation of the methodology for the IBM POWER5 processor, which shows that the following:
 - a. 17 out of 10,000 possible signatures are sufficient to characterize 95.6% of the execution times of a set of applications that consists of 20 SPEC CPU2006 benchmarks (1 data input), three NAS NPB benchmarks (3 data inputs), and 10 PETSc KSP solvers (12 data inputs). The cgs and lsqr PETSc KSP solvers had signatures that were independent of input data, while one of three NAS NPB benchmark (bt-mz) had a signature that was independent of the input data.
 - b. For 21 co-schedules of applications each with a signature that characterizes 95% of its execution time, our validation study shows the following:
 - v. *Predicted best priorities yield higher throughput than default priorities for all but one of the 21 co-schedules.* Initial results showed that two co-schedules (462.libquantum, 437.leslie3d) and (bt-mz.A, lu-mz.A) experience a throughput loss of 7.46% and 20.05%, respectively, at predicted priorities, as compared to that achieved at default priorities. Further investigation shows that for the co-schedule (bt-mz.A, lu-mz.A) mapping and executing the co-schedule with the predicted best priorities on hardware threads (5, 4), instead of (4, 5), results in a 3.56% higher throughput as compared to default priorities – this is in contrast to the 20.05% throughput loss experienced when executed on hardware threads (4, 5). Although we have not verified it, one possible reason for this is that the processor core favors one hardware thread over the other. Re-executing the co-schedule (462.libquantum, 437.leslie3d) on hardware threads (5,4), instead of (4, 5), results in predicted priorities yielding lower throughput than the default priorities. Thus, we claim that predicted best priorities yield equal or higher throughput than default priorities for 20 of the 21 co-schedules studied, and for the outlier the throughput loss is 7.46%.

- vi. *Using non-default priorities improves throughput.* The default priority pair yields best throughput for only six of the 21 co-schedules. For the remaining 15 the default priority pair yields throughput that is between 0.74% and 14.10% lower than that achieved with the best priority pair.
- vii. *Using the predicted best priority pair, rather than default priorities, improves throughput or at least provides throughput equal to that achieved with default priorities.* For 11 of the 21 co-schedules both the default and predicted priorities yield equal throughput. For nine of the 21 predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For two of these nine co-schedules the predicted priority pair yields a throughput improvement of less than 5%. Furthermore, for three the throughput improvement associated with executing with the predicted priority pair, rather than default priorities, is between 5% and 10% and for the other four the improvement is greater than 10%.
- viii. *Using predicted best priority pairs appears to be most applicable to floating-point “intensive” applications:* For eight co-schedules comprising applications for which the utilization of the floating-point unit exceeds that of the fixed-point unit by 10% or more, the predicted priority pairs, as compared to the default priorities, yield a throughput improvement between 3.56% and 16.42%. This result indicates that the methodology for predicting best priority pairs is most applicable to applications for which floating-point unit utilization dominates that of the fixed point unit by at least 10%.

1.6 DISSERTATION OUTLINE

The remainder of the dissertation is organized as follows. Chapter 2 provides background information, including an overview of instruction-level parallelism, multithreading, and the IBM POWER5 processor and hardware thread priorities, while Chapter 3 presents related research. Our pilot

study of the performance effects of hardware thread priorities, which was conducted using the POWER5 simulator, is described in Chapter 4 – this study motivated this dissertation research. The best priority pair prediction methodology is described in Chapter 5 and an implementation of the methodology, which uses the IBM POWER5 processor, is described in Chapter 6, along with an analysis of the accuracy of the implementation. Finally, Chapter 7 presents conclusions and future work.

2 Background

This chapter presents the background material necessary to understand simultaneous multithreaded (SMT) processors. It is organized as follows. Section 2.1 motivates the advent of SMT processors by introducing the problem of underutilized processors. Sections 2.2 and 2.3 discuss ways to improve processor utilization that are based on instruction-level parallelism and hardware multithreading. Simultaneous multithreading, a particular form of hardware multithreading, is described in Section 2.4. Sections 2.5 and 2.6 present the SMT implementation of the IBM POWER5, and those of the Intel processors and the Sun Niagara, respectively. The IBM POWER5 simulator, which was used in the initial stages of this research, is briefly described in Section 2.7. Finally, SMT support provided by operating systems for the IBM POWER5 processor is explained in Section 2.8.

2.1 INTRODUCTION

The gap between the speed of processors and the speeds of memory and I/O devices has continued to widen [1]. As a result, processors remain underutilized. A system with low processor utilization results in low system throughput, causing application performance to suffer. Accordingly, underutilized systems lead to an increase in the cost of computation. Hence, to achieve a specified throughput, more equipment must be purchased and, as a result, the infrastructure is strained and power consumption and cooling requirements increase.

Stalls incurred by applications in a processor's instruction execution pipeline cause processor cycles to be wasted and the processor to be underutilized. Stalls occur due to three types of hazards: (1) structural, which are due to resource conflicts, (2) data, which are due to data dependences among instructions, and (3) control, which are due to branch mispredictions and instructions that cause the flow of execution to change. A detailed explanation of these hazards is available in [1]. In addition, stalls occur due to cache and TLB misses.

The number of stall cycles injected into the pipeline depends on the latency associated with the cause of the stall. For example, a cache miss that hits in main memory is a high-latency operation, whereas a conflict for a functional unit that resolves in less than 10 cycles in the instruction execution

pipeline is a low-latency operation. Multi-level caches, hardware prefetching, and related technologies have reduced the cost of memory stalls. Nonetheless, processors remain underutilized.

2.2 INSTRUCTION LEVEL PARALLELISM

There exists inherent parallelism in an application's instruction stream — this is called instruction level parallelism (ILP). ILP can be exploited to execute multiple independent instructions in parallel. This property has been used by system architects to implement hardware and software techniques to improve processor utilization.

Some of the hardware techniques based on ILP to improve processor utilization are:

- *Superscalar processors:* A superscalar processor has replicated functional units, which help to reduce the number of structural hazards and stalls related to unavailable functional units. This translates to better exploitation of available application ILP and, thus, more opportunities for parallel execution.
- *Speculative execution:* Some processors have hardware that implements branch prediction and supports speculative execution. Such processors can speculatively fetch instructions from the predicted branch path and begin to execute them. Accordingly, if a branch is predicted correctly, execution does not stall. This reduces stalls associated with the fetching of instructions from the branch path and increases available ILP.
- *Out-of-order (dynamic) execution:* In a superscalar processor, stalls due to data hazards can cause functional units to be idle. Dynamic scheduling or out-of-order execution is a technique implemented in hardware that allows instructions to execute out of program order to hide the latency associated with data hazards. In general, an instruction is still issued and committed in program order, however, it begins execution as soon as its operands are available and can, thus, execute out of program order. A number of schemes like scoreboarding and Tomasulo's

algorithm, described in [1], use hazard detection and register renaming to take advantage of out-of-order execution.

- *Multiple-issue/fetch*: In a superscalar processor, to further improve the utilization of pipeline resources, multiple instructions are fetched and issued every clock cycle. A larger instruction window, i.e., the number of instructions in the pipeline, increases the potential to extract application ILP.

Some of the software techniques based on ILP to improve processor utilization are:

- *Loop unrolling*: This technique, combined with dependence analysis, allows a compiler to reduce the number of iterations of a loop, thus, reducing the number of branches.
- *Software pipelining*: This technique reduces stalls associated with data hazards without unrolling a loop. It reorganizes the loop body, interleaving instructions from different iterations. In this way, it increases the number of independent instructions that can be executed concurrently.
- *Software-based multiple-issue using very long instruction words*: An alternative to the superscalar approach is to issue multiple instructions using a single Very Long Instruction Word (VLIW). The advantage of VLIW is that the hardware need not explicitly check for dependences. Dependency checking is the responsibility of the compiler to guarantee that the instructions in a VLIW do not have dependences.

Modern processors employ a combination of the above-mentioned hardware and software techniques. This enables the processor to potentially complete more than one instruction per cycle (IPC).

2.3 HARDWARE MULTITHREADING

Despite the hardware and software techniques described above, processor utilization continues to remain low due to several factors that limit the use of application ILP. For example, without very accurate branch prediction, an application's ILP is limited by the size of a basic block. And, today's,

wide-issue, out-of-order, superscalar processors may have to resolve a branch every two or three clock cycles. In addition, the use of pointers and indirect memory references reduce the accuracy of branch predictors. Hence, the instruction stream from one application provides limited opportunities to exploit ILP.

Increasing the number of applications executed concurrently gives the processor the opportunity to extract ILP from a larger pool of instructions. This allows the processor to keep the functional units busier and, thus, improve resource utilization. Accordingly, explicit hardware multithreading [2] has emerged as a design choice to improve processor utilization and, thus, processor throughput. Hardware multithreading allows applications to execute on the independent hardware threads of a processor core. The literature [2] discusses two main approaches to hardware multithreading: fine-grained and coarse-grained.

A fine-grained multithreaded core switches between hardware threads every clock cycle, whereas a coarse-grained multithreaded core switches only on high-latency stalls, such as L2-cache misses. Fine-grained multithreading overcomes throughput losses due to both short and long stalls. However, compared to coarse-grained multithreading, it slows down the execution of individual applications. The primary disadvantage of coarse-grained multithreading is its inability to overcome throughput losses due to short stalls. However, since a coarse-grained multithreaded core switches hardware threads only on costly stalls, it is less likely to slow down the execution of individual applications.

This research focuses on simultaneous multithreading (SMT), which is a variant of fine-grained multithreading. On each cycle, an SMT core can issue multiple instructions from multiple applications and can concurrently execute them on multiple hardware threads. Hence, SMT can improve processor utilization but it may potentially decrease individual application performance. SMT is implemented on modern superscalar, multiple-issue, out-of-order processors, such as the IBM POWER5 and POWER6 [10, 21] and Intel processors [5, 6], to exploit both thread- and instruction-level parallelism.

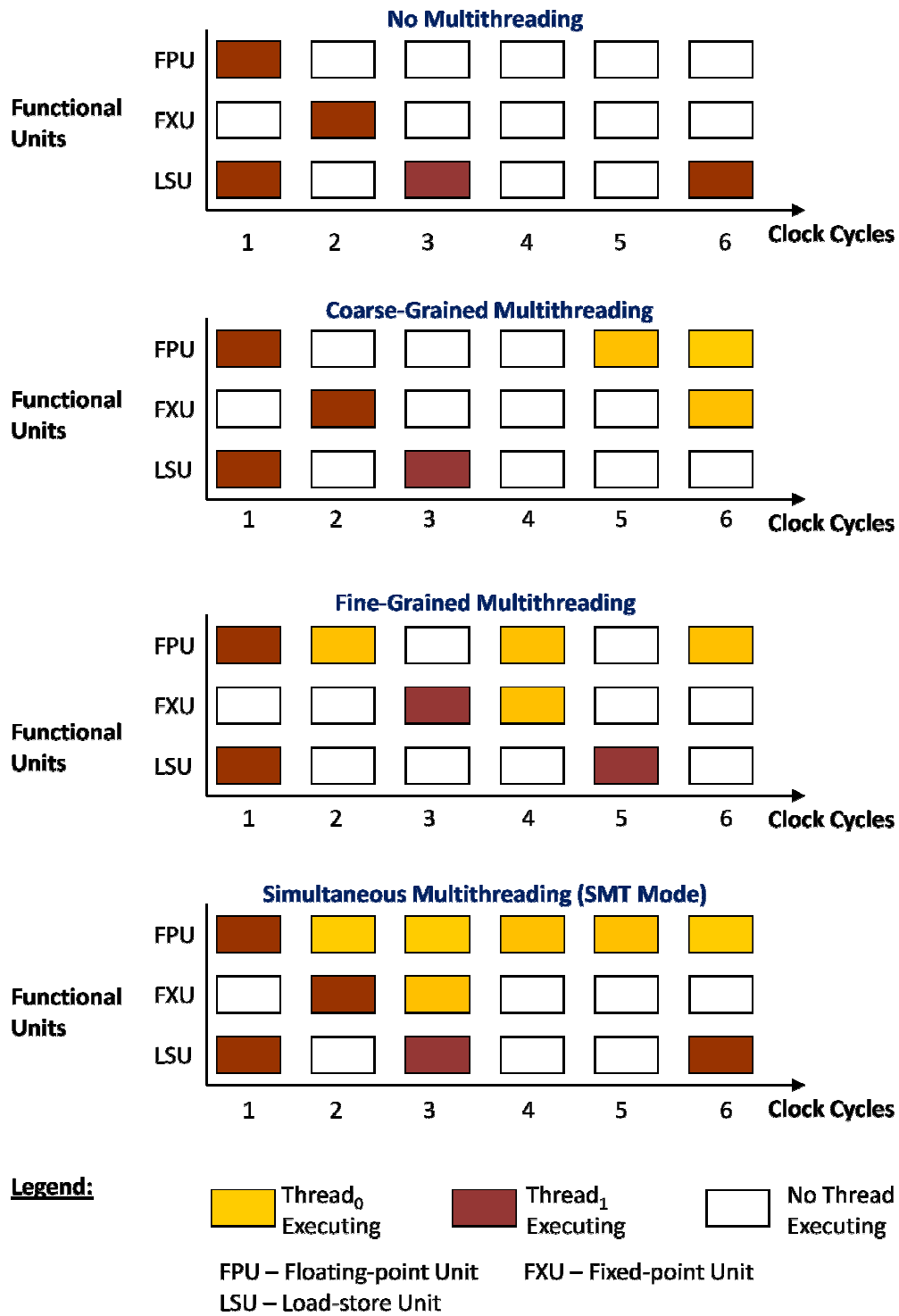


Figure 2.1: Illustration of the Utilization of Functional Units for
Different Forms of Multithreading

Figure 2.1 illustrates the utilization of different functional units in the following processor configurations: (1) a superscalar processor core with no multithreading (topmost chart), (2) a superscalar processor core with coarse-grained multithreading (second chart from top), (3) a superscalar processor core with fine-grained multithreading (second chart from bottom), and (4) a superscalar SMT processor core executing in SMT mode (bottom chart). Each multithreaded configuration in this figure has two hardware threads, Thread₀ and Thread₁.

The Y-axis of each chart represents a subset of the processor's functional units, while the X-axis represents clock cycles. At any given cycle, the three resources (functional units) are either idle (not used by Thread₀ or Thread₁) or busy (used by one of the threads). If a resource is idle during a specific cycle, it is represented by a non-shaded box. In contrast, if the resource is busy, it is represented by a shaded box. Comparing the use of the three resources in the six clock cycles shown in the graphs, SMT mode has the best utilization; of course, this may not always be the case.

2.4 SIMULTANEOUS MULTITHREADING (SMT)

Simultaneous multithreading allows the multiple hardware threads of an SMT processor core to concurrently execute independent instruction streams every clock cycle. SMT allows an application executing on a hardware thread of a core to utilize shareable processor resources that are left idle by applications running on other hardware threads, thus, potentially increasing processor utilization and throughput. Typically, the individual hardware threads of a core are supported by separate hardware, e.g., by separate fetch buffers and program counters. At every cycle, applications executing on the hardware threads of a core contend for its shareable resources, e.g., instruction execution pipeline resources and the memory subsystem.

Figure 2.2 depicts an SMT core with two hardware threads. Each hardware thread has an independent program counter, instruction fetch buffer, and write-back stage. As seen in this figure, where white boxes indicate shareable resources, the threads share the majority of the core's instruction execution pipeline resources.

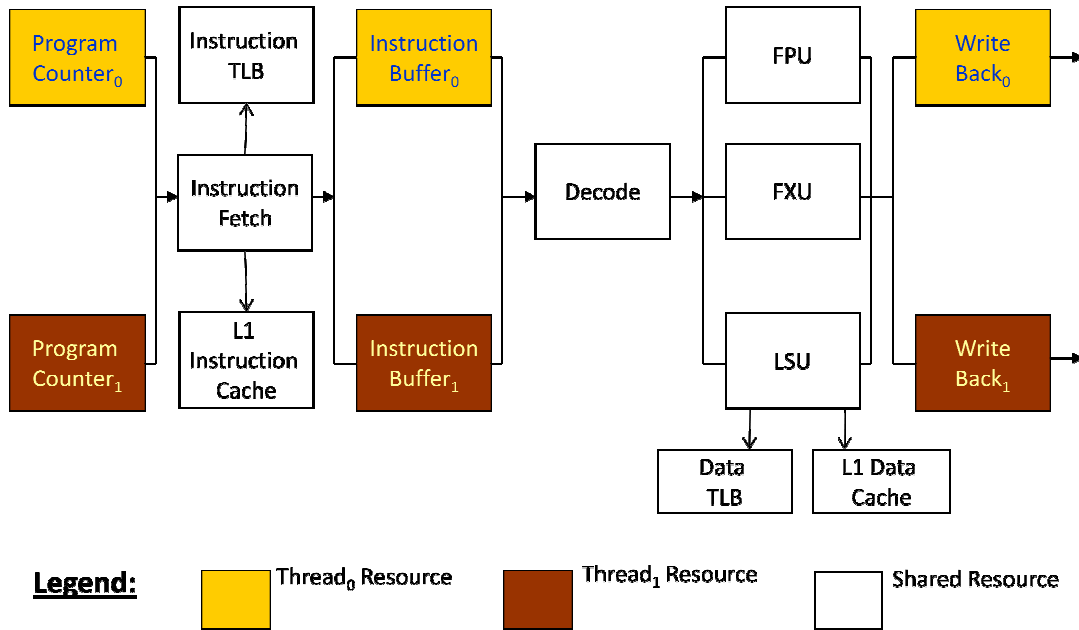


Figure 2.2: Conceptual Instruction Execution Pipeline of a
SMT Core with Two Hardware Threads

Applications running on the hardware threads of a core continue execution as long as they can utilize resources that are left idle by applications executing on other hardware threads. In the best case, applications running on two threads of a core execute without resource conflicts, and core performance is equal to the aggregate performance of the two hardware threads. Thus, by allowing two applications to utilize SMT core shareable resources, core throughput can potentially double. Hence, while SMT processors have the potential to improve processor utilization and, thus, throughput, contention for shared resources limits such gains. The problem of shareable-resource contention in SMT cores was illustrated in Chapter 1.

2.5 IBM POWER5 PROCESSOR

Simultaneous multithreading has been implemented in processors by Sun [4, 22], Intel [5, 6], and IBM [10, 21]. SMT-mode enabling and disabling is supported by all three manufacturers' processors. In the Sun Niagara, hardware thread priorities can be assigned only by hardware, whereas they can be assigned by software in the IBM processors. Thread priorities are not supported by Intel processors.

Since the IBM POWER5 processor has software-controlled hardware thread priorities, it is used in this research. Thus, currently, our methodology is useful only for IBM POWER5 and POWER6 [10, 21] processors. Sections 2.5.1, 2.5.2, and 2.5.3 describe the POWER5 instruction execution pipeline, hardware thread priorities, and other SMT-related hardware tunables that are not used in this research. IBM's SMT performance studies are discussed in Section 2.5.4.

2.5.1 IBM POWER5 Instruction Execution Pipeline

In the year 2000, IBM introduced coarse-grained multithreading in the STAR processor [9]. This processor was available in the IBM eServer pSeries model 680. In 2001, IBM released the POWER4 processor, which is a dual-core processor that takes advantage of thread-level parallelism at the chip level. IBM's POWER5 processor design [10] enhances that of the POWER4 by adding two hardware threads per core. As such the POWER5 is a 64-bit, dual-core, simultaneous-multithreaded (SMT) processor with two speculative superscalar cores that support out-of-order execution. Each processor core, with its two hardware threads, supports both SMT and single-threaded modes of execution.

Figure 2.3 presents the layout of a typical POWER5 chip. Each of its two SMT cores has a 64KB L1 instruction cache, 32KB L1 data cache, and 1.9MB L2 unified cache. Additionally, the chip has an L3-cache directory and a memory controller. POWER5 servers can have up to four multi-chip modules (MCMs), each with four POWER5 chips and a 36MB L3 cache.

Each core has the following translation resources, which are shared between its two hardware threads: (1) two 128-entry effective-to-real address translation (ERAT) caches, one for instructions (I-ERAT) and one for data (D-ERAT); and (2) one, unified, 1024-entry TLB. The effective address of a small page is looked up in an ERAT first. The TLB is accessed on an ERAT miss and is used for address translation of large pages.

Figure 2.4 illustrates the POWER5 instruction execution pipeline of a core, which can be partitioned into the following stages: instruction fetch, decode and dispatch, execution, and retirement. The following functional units are available: two nine-stage pipelined floating-point units (FPUs), two six-stage pipelined load-store units (LSUs), two non-pipelined fixed-point units (FXUs), one non-

pipelined branch execution unit (BXU), and one non-pipelined condition register logical unit (CRL). The following issue queues are available: two 18-entry issue queues shared by the two FXUs and the two LSUs, a 12-entry issue queue for each of the two FPU, one 12-entry issue queue for the BRU, and one ten-entry issue queue for the CRL.

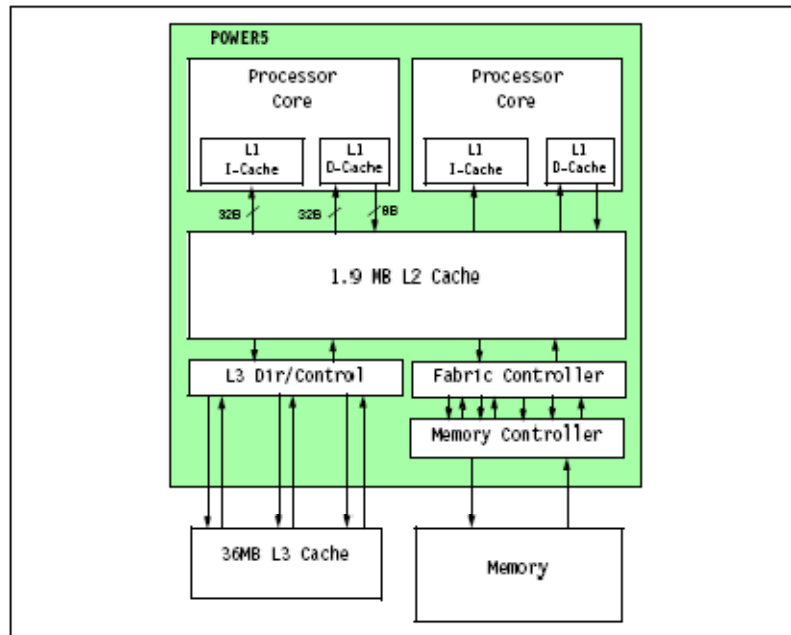


Figure 2.3: POWER5 Chip Layout (Source [20])

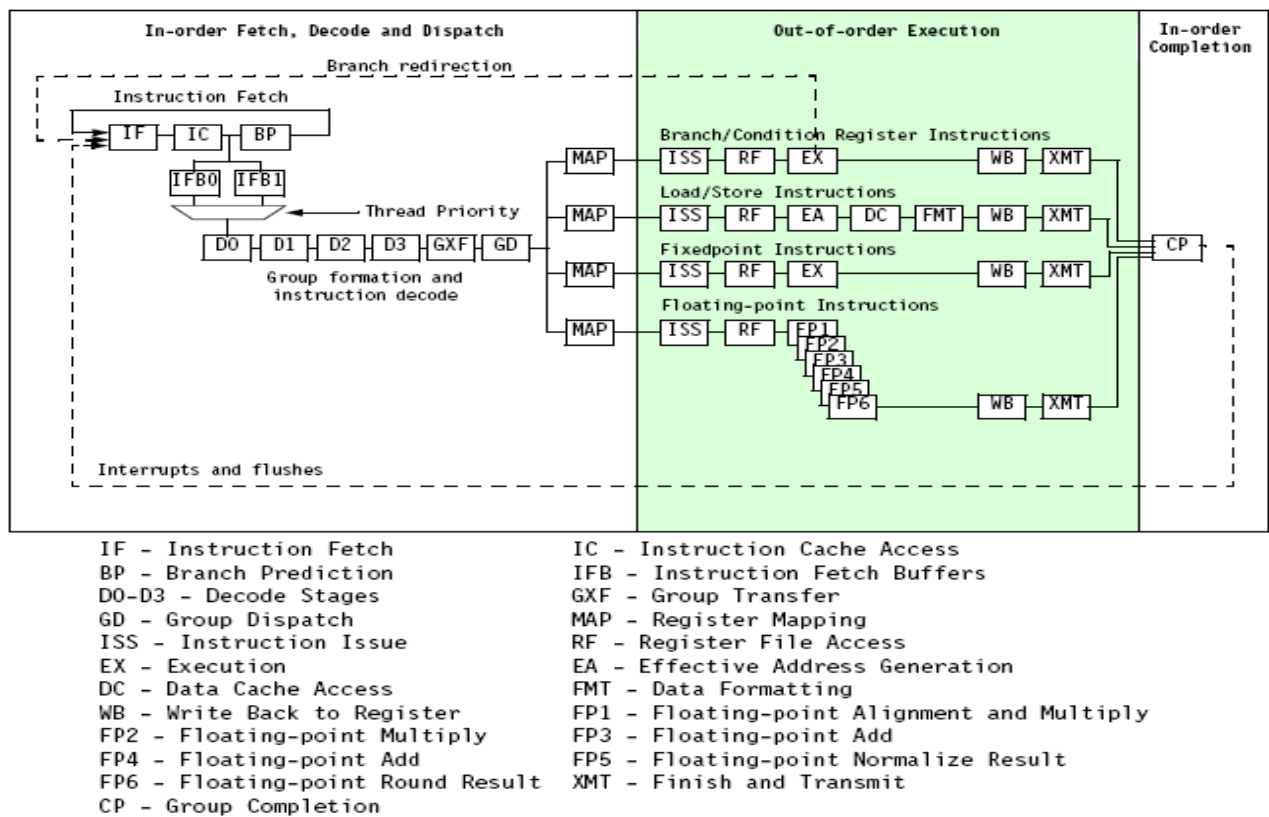


Figure 2.4: Instruction Execution Pipeline of a POWER5 Core (Source [10])

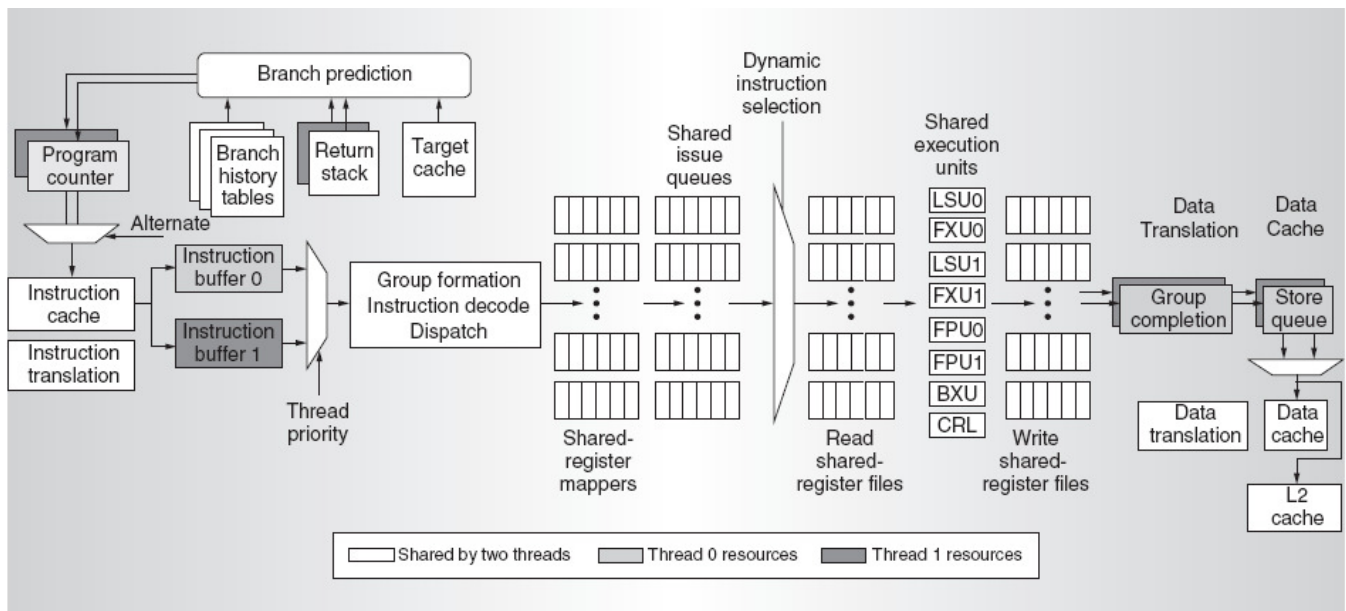


Figure 2.5: Instruction and Data Flow in the POWER5

Instruction Execution Pipeline of a Core (Source [10])

Figure 2.5 shows the flow of instructions and data in the POWER5 instruction execution pipeline of a core. The instruction fetch stage has two program counters and two 24-entry instruction fetch buffers, one program counter and one instruction buffer for each of the two hardware threads. Since instruction fetches alternate between the two threads, during every other cycle eight instructions of one of the threads are fetched and stored in its buffer. Every cycle, the instructions in each thread's buffer are scanned by the branch prediction hardware, which updates each thread's program counter based on its prediction. The branch prediction hardware consists of three branch history tables shared by the two threads. In addition, the threads share a branch target cache and a return stack to predict the return addresses of subroutine calls.

Instruction decoding alternates between the two threads. In any given cycle the processor forms a group of up to five instructions of one of the threads. The group of instructions is decoded and assigned a group entry in the shared 20-entry group completion table (GCT). When the resources required for dispatch are available, the group is dispatched and issued as a group to the issue queues; there is one issue queue per functional unit. The physical registers of the decoded instructions are renamed to the shared 240 logical registers; subsequently the instructions enter the issue queues shared by the two threads. The issue logic selects eligible instructions from the issue queues for execution. For instruction issue, there is no distinction between instructions from the two threads. A group of instructions is marked as "complete" when all the instructions in the group complete execution; the instructions in a group are completed in order and the groups of each thread are completed in order. The processor can commit up to two groups of instructions per cycle, one from each thread.

2.5.2 IBM POWER5 Hardware Thread Priorities

The POWER5 implements hardware thread priorities that can be used to control the ratio of decode cycles allocated to each of the two threads of a core. Based on the priority settings of the threads, Thread₀ and Thread₁, instructions from the threads are decoded and dispatched, i.e., issued, in the following manner. If the default priority pair, i.e., equal priorities, is assigned to a co-schedule, instructions from a thread are decoded every other cycle. If the priorities are not equal, then the decode

cycles are not alternated between the threads but, instead, are allocated according to a ratio, explained later, where i cycles are first allocated to Thread₀ and then j cycles are allocated to Thread₁. The hardware thread priorities listed in Table 2.1 can be used by software to control the ratio of the decode cycles allocated to the two threads of a core. Hardware thread priorities also are used by the POWER5's dynamic resource balancing (DRB) logic to improve performance in SMT mode. The DRB is hardwired logic that attempts to provide resource-sharing fairness between a co-schedule's applications. This is accomplished by the following actions of the DRB:

- a. If an application executed on a thread incurs more than a fixed threshold number of L2-cache misses, inhibit the thread's instruction decode, until the number of outstanding L2 cache misses goes below the threshold.
- b. If an application executing on a thread is executing a long instruction such as a sync instruction, flush the instructions that are waiting for dispatch and inhibit the thread's instruction decoding until the sync instruction is completed.
- c. If an application running on a thread uses more than a fixed threshold number of group completion entries, reduce the thread's priority, until the number of group completion entries used by the thread goes below the threshold.

There are eight hardware thread priority levels (0 through 7) that determine the relative number of decode cycles allocated to the two threads of a POWER5 core – the default priority setting of a thread is 4. The hypervisor is allowed to assign all eight levels; the supervisor (operating system) is allowed to assign levels 1 through 6; and user-mode software can assign levels 2 through 4. The difference in the priorities of the two threads of a core determines the relative number of decode cycles allocated to each. The higher-priority hardware thread receives relatively more decode cycles than its lower-priority counterpart. This gives the application executing on the higher-priority thread the opportunity to get more work done. If both threads have priorities greater than 1, then decode cycle allocation is calculated as follows: Given i as the priority of Thread₀ and j as the priority of Thread₁, the decode cycle share for the lower-priority thread is given by $1/2^{(i-j+1)}$, while the higher-priority thread's decode cycle share is $1 - 1/2^{(i-j+1)}$. For example, if Thread₀ has priority 4 and Thread₁ has priority 2, then Thread₁ gets $1/2^{(4-2+1)} =$

1/8 of the decode cycles, i.e., one cycle out of every eight cycles. In contrast, Thread₀ gets 7/8 of the decode cycles, or seven out of eight decode cycles. First Thread₀ gets seven decode cycles and then Thread₁ gets one cycle. Thread priority settings where at least one thread has priority 0 or 1 are special cases; a priority of 0 turns off a thread and a priority of 1 puts a thread into a low-power state. The decode cycle allocation for all priorities is shown in Table 2.2.

Table 2.1: POWER5 Hardware Thread Priority Levels

Thread Priority Level	Description	Software Privilege Level Required to Set the Priority
0	Thread Shut-Off	POWER Hypervisor Mode
1	Very Low	Supervisor Mode
2	Low	User/Supervisor Mode
3	Medium Low	User/Supervisor Mode
4	Normal	User/Supervisor Mode
5	Medium High	Supervisor Mode
6	High	Supervisor Mode
7	Extra High	POWER Hypervisor Mode

Table 2.2: Effect of Hardware Thread Priorities on Decode Cycle Allocation

Priority _{Thread0}	Priority _{Thread1}	Decode Cycle Allocation
0	0	Both Thread ₀ and Thread ₁ are stopped.
0	1	For power savings, Thread ₁ decodes up to five instructions every 32 clock cycles and Thread ₀ is stopped.
0	>1	Thread 1 uses all processor resources; it fetches and

		executes instructions every clock cycle.
1	1	Every 64 cycles, each thread will start up to five instructions.
1	>1	Thread ₁ gets access to all of the execution resources and Thread ₀ gets resources that are not used by Thread ₁ . Thread ₁ 's performance should be similar to single-threaded mode.
>1	>1	The number of decode cycles used by Thread ₀ , before yielding to Thread ₁ , is determined by the formula $1/(2^{(lx-yl+1)})$.

2.5.3 Other SMT Hardware Tunables

Besides being able to set thread priorities, the software also can set the execution mode, i.e., it can control whether the core executes in single-threaded or SMT mode. In our initial experiments we found that for nearly 99% of application trace-pairs, SMT mode gave better throughput than single-threaded mode. Hence, the execution modes that can be set by software, which are described below, were not used in this research.

- *SMT Mode Enable/Disable*: This mode of execution can be set on the command line. It allows the software to switch the processor from SMT mode to single-threaded mode and back to SMT mode.
- *SMT Snooze*: The software can use this option to put a hardware thread into the dormant state (snoozing). A thread in this state can be awakened by an interrupt or by the hypervisor *H_PROD_POWER* system call. The advantage of “snoozing” a thread, rather than running the operating system idle process, is that it frees up all the core resources that would be used by the thread, which can be utilized by the core’s active thread. The down side is that the “snoozing” thread incurs a start-up latency of several thousands of cycles to switch from the dormant state to the active state.

- *SMT Snooze Delay*: This tunable parameter is related to the *SMT Snooze* option and can be used by the operating system to wait a fixed amount of time before “snoozing” a thread. The intent of this parameter is to allow the operating system to monitor the run queue for ready-to-run processes before “snoozing” a thread, thus, preventing wasted cycles due to thread start-up latency.

2.5.4 SMT Performance Studies

An IBM study [11] evaluated the performance of SMT in the POWER5 processor for workloads that belonged to compression, database, and neural network application classes. The authors ran their experiments on a dual-core POWER5 chip with four hardware threads, two per core. The single-threaded mode run time is recorded as the time taken to execute identical copies of the same workload on one hardware thread of each of the two cores. The SMT mode run time is recorded as the time taken to execute four identical copies of the same workload on both hardware threads of both cores. The SMT gain was calculated as the ratio of SMT mode run time to single-threaded mode run time. This study showed that SMT mode provided a performance gain of 12% to 41% except for the neural network workload, which showed a performance drop of 11% when executed in SMT mode – this was due to a higher number of L2-cache misses observed in SMT mode as compared to single-threaded mode.

Another IBM study [20] quantified SMT gain for the following parallel benchmarks: BLAST, AMBER, and Gaussian03. The authors executed these workloads on an IBM eServer p570 machine with two IBM POWER5 processors, which has a total of four cores and eight hardware threads. Each benchmark was executed with eight software threads (i.e., with a parallelism level of eight) in both single-threaded and SMT modes. In single-threaded mode, one software thread was assigned to each core with SMT disabled, while in SMT mode a pair of the eight software threads was assigned to each of the four cores, i.e., to its two hardware threads. SMT gain was calculated as the ratio of the time taken to execute the program in SMT mode to the time taken to do the same in single-threaded mode. The results of this experimentation showed that BLAST, Gaussian03, and AMBER had SMT gains of 20%, 15%, and 16%, respectively.

2.6 SMT IMPLEMENTATION ON OTHER PROCESSORS

This section focuses on SMT implementations of other processors. Sections 2.6.1 and 2.6.2 describe the SMT implementations of the Sun Niagara and Intel processors, respectively.

2.6.1 Sun Niagara

The Sun Niagara [4, 22] is a processor with eight cores that support in-order execution. Each core has four hardware threads and, thus, there are a total of 32 hardware threads on a chip. Each core has a 16KB L1 instruction cache and an 8KB L1 data cache. The eight cores share a 3MB unified L2 cache. For address translation each core has a separate 64-entry TLB for instructions and data address translation.

Figure 2.6 shows the design of the execution pipeline of each core of the processor. The four hardware threads of a core share the majority of the resources of the core, however, each thread has an independent program counter, instruction fetch buffer, register set, and load/store buffer entries.

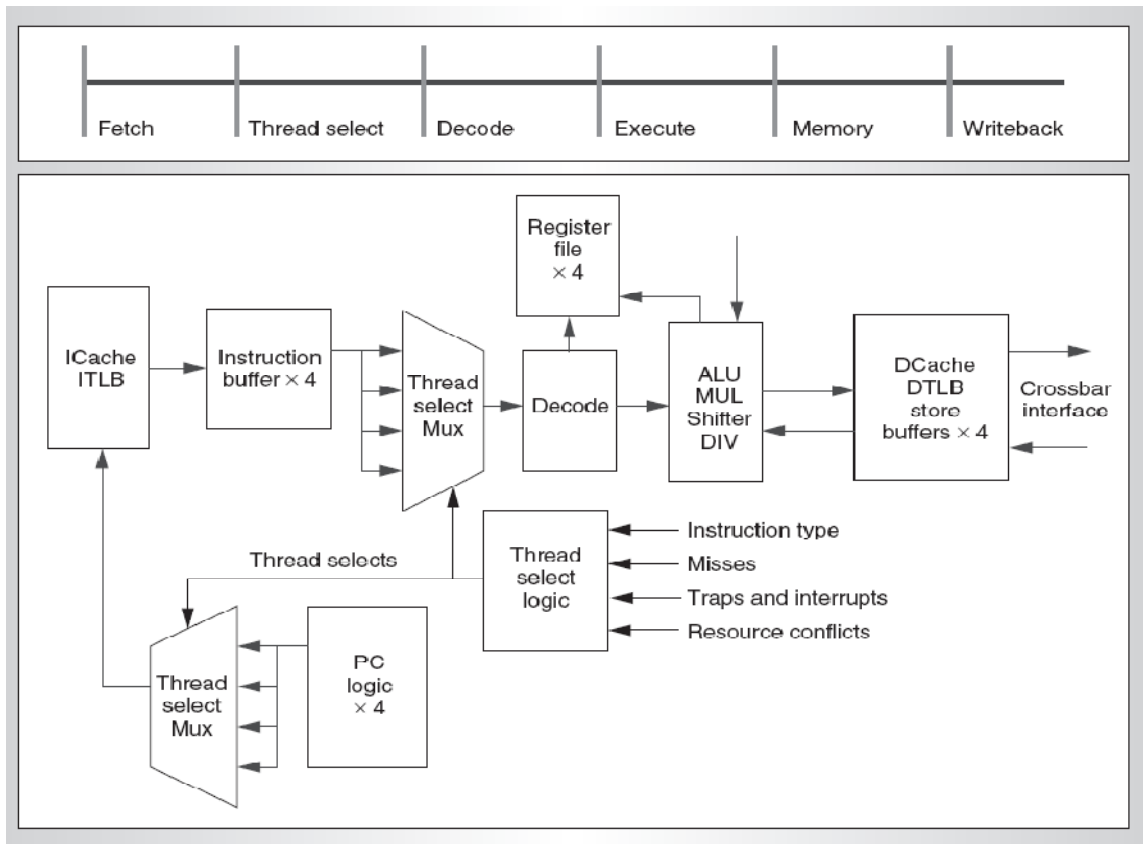


Figure 2.6: SPARC Core Execution Pipeline Block Diagram (Source [4])

The pipeline can be broken down into six stages: Fetch, Thread Select, Decode, Execute, Memory, and Writeback. During the fetch stage up to two instructions are fetched for the thread selected by the thread-select multiplexer. The fetched instructions are stored in the thread's instruction buffer.

The thread chosen for the fetch stage also is chosen for decoding. There is one each of the ALU, multiply, divide, shift, and load/store units. The load/store, multiply, and divide units have multi-cycle latencies and, hence, a thread must stall if it needs to use any of these units and it is busy.

To guarantee fairness, every cycle thread selection among the active threads is accomplished using the least-recently-used policy. The thread-selection policy assigns lower priority to threads that are executing long-latency operations such as multiplies and loads to ensure that they do not monopolize resources and impede the forward progress of other active threads. Additionally, on a long latency operation, such as an L2 cache miss, a thread becomes unavailable for selection until the miss is satisfied. In contrast, the DRB hardware on the IBM POWER5 lowers the priority of threads when a thread executes a sync instruction, uses more than a threshold number of group completion table entries,

or has more than a threshold number of L2 cache misses. In the Sun Niagara, neither the user software nor the operating system kernel is allowed to set the priorities of hardware threads.

2.6.2 Intel Hyper-Threading

Intel introduced SMT, with two hardware threads per core, as Hyper-Threading on the Xeon and Pentium 4 processors in 2002 [5, 6]. Currently the technology is available on the following Intel processors: Pentium 4, Mobile Pentium 4, Pentium 4 Extreme Edition, Xeon MP, Nehalem, and Dual-core Xeon.

Figure 2.7 shows the design of Intel's Netburst microarchitecture pipeline [6], which supports out-of-order execution and SMT with two hardware threads. This microarchitecture is used for the Pentium 4 processor; additional details are available in [58]. The threads share the functional units, caches, and TLB. Like the IBM POWER5, the threads have separate program counters and instruction fetch buffers. In contrast, the register set, micro-op queue entries, reorder buffer entries, and load/store buffer entries are statically partitioned between the two hardware threads and unused resources are not available to other threads.

The IA-32 instruction set is relatively complicated, hence, the fetched instructions are translated into simple RISC-like microoperations (μ ops) before they are stored in the trace cache, which is used instead of an L1 instruction cache. The instructions are fetched from the trace cache in order, executed out of order, and retired in order.

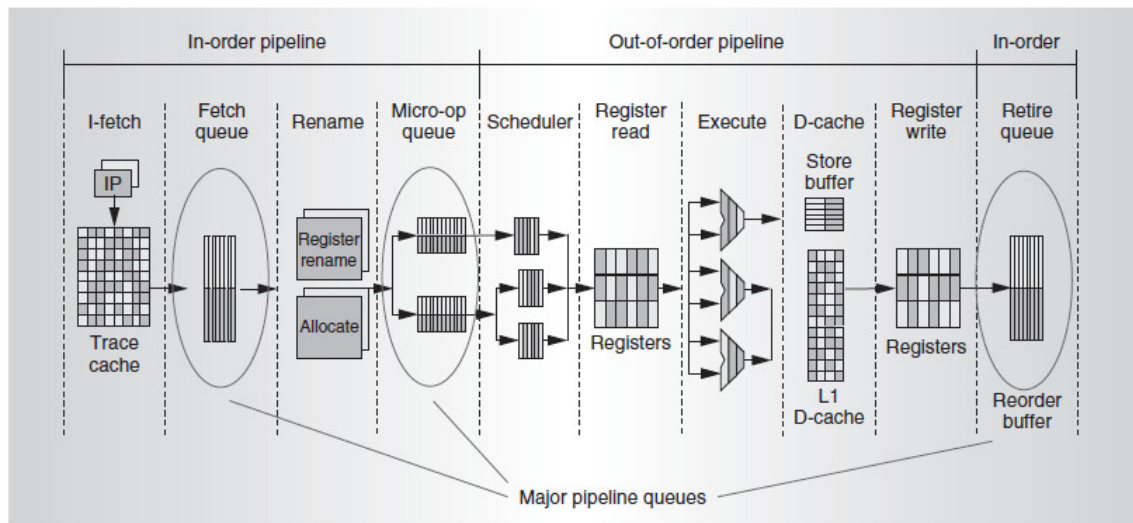


Figure 2.7: Intel Netburst Microarchitecture Pipeline (Source [6])

A thread's fetched instructions are stored in the instruction buffer. The decode stage alternates every cycle between the instruction buffers of the two threads and places the instructions in the μ op queue. The Execute unit can dispatch instructions from one or both threads to the scheduler for execution. Completed instructions are retired in program order and in a given cycle, the retirement logic retires up to three instructions of only one thread. The retirement logic alternates between the two threads every cycle.

The processor is allowed to execute in two modes of operation: single-threaded mode or SMT mode. In single-threaded mode the active hardware thread is assigned all the resources on the processor. Hardware thread priorities are not supported in this architecture.

2.7 IBM PERFORMANCE SIMULATOR FOR LINUX ON POWER

There have been a number of simulators built to study SMT processors [12-16]. This section describes the simulators that could be used to study SMT on the IBM POWER5.

The IBM Performance Simulator for Linux on POWER [17] is available for PowerPC architectures running Linux. The simulator, which takes as input an instruction trace, simulates one core of the dual-core IBM POWER5 processor [10], which has two hardware threads per core. As mentioned previously, the difference between the priorities of the two threads controls the ratio of the decode

cycles allocated to the two threads, which, in turn, allows an application executing on the higher-priority thread to potentially make faster forward progress through the execution pipeline.

The simulator permits the user to select between SMT and single-threaded modes of execution and, in SMT mode, to select the hardware thread priorities described in Section 2.5.2. Its output provides detailed performance data concerning cycles per instruction (CPI), branch prediction, instruction fetch, caches, the prefetch unit, and the functional units, as well as an instruction histogram. Because the simulator models one core of the dual-core IBM POWER5 processor, it can be used to study SMT behavior but cannot be used to study SMP (symmetric multiprocessor) behavior.

The relevant command-line arguments required to run the simulator in SMT mode are:

- **Trace0**: trace to be executed on Thread₀
- **Trace1**: trace to be executed on Thread₁
- **num_inst**: number of architected instructions to run before exiting
- **t0_prio**: priority of Thread₀
- **t1_prio**: priority of Thread₁

The relevant command-line arguments required to run the simulator in single-threaded mode are:

- **Trace**: trace to be executed
- **num_inst**: number of architected instructions to run before exiting
- **num_inst_to_skip**: number of instructions of the trace to skip before starting execution; this can be set to zero to execute all instructions

2.8 OPERATING SYSTEM SUPPORT FOR IBM POWER5 SMT

IBM POWER5 SMT is supported by the AIX 5L V5.3 [18] and Linux 2.6 [19] operating systems. This section describes the support provided by these operating systems for process scheduling and the use of hardware thread priorities to improve kernel performance.

2.8.1 Process Scheduling

A multiprocessor kernel can run on a multi-core SMT processor without modifications. However, it is not able to use the multithreading feature of a core. In contrast, the AIX 5L V5.3 and

Linux 2.6 kernels for the POWER5 are SMT-aware; they provide separate run queues for each hardware thread.

Affinity domains, supported by both the AIX and Linux kernels, are groupings of hardware resources in hierarchical domains. The lowest level of the hierarchy represents resources that are closest to each other. In SMT mode the lowest-level affinity domain consists of a single hardware thread; at the next level are the hardware threads of a core. Affinity-domain scheduling attempts to schedule a task to the same domain. To take advantage of saved state in the cache hierarchy or buffers, preference is given in hierarchical order, lowest level to highest level.

The AIX kernel schedules multiple tasks of a parallel task group to the hardware threads of a core. Since such tasks often share code and data, this affinity scheduling policy can leverage the benefit of cached data shared by multiple tasks. In contrast, for independent sequential tasks, the operating system distributes tasks to hardware threads on separate physical processors in a round-robin fashion. This ensures that all the physical processors have an equal load. After all the cores of each processor have been allocated one task, if there are additional tasks, then the work is distributed in a round-robin fashion to the second hardware thread of each core.

As mentioned before, in an SMT processor, threads executing in SMT mode may compete with each other for shared resources. Such interference slows down application execution, e.g., an active thread's execution may slow down even if the second thread is executing the idle loop. Thus, in light-load conditions, the AIX and Linux kernels do the following: (1) distribute processes to only one hardware thread of each core and execute the idle loop on the second hardware thread, and (2) after a fixed tunable time quantum, put the second hardware thread into the dormant state and transition the processor from SMT mode to single-threaded mode. Thus, for maximum performance, under light-load conditions the active hardware threads are executed in single-threaded mode. When the workload increases, the dormant threads are awakened, the processor transitions to SMT mode, and the work is distributed evenly across all hardware threads.

2.8.2 Use of Hardware Thread Priorities

As described in Section 2.5.2, in the POWER5 the difference between the priorities of two hardware threads dictates the ratio of processor decode cycles allocated to the threads. This allows application software to throttle threads.

AIX and Linux assign the default priority of 4 to each thread, causing each thread to decode one out of every two decode cycles, and modify these priority settings to improve system performance, i.e., a thread's priority is decreased when it is doing non-productive work and is increased when it needs to expedite its progress. For example, both AIX and Linux use spinlocks, which are similar to binary semaphores, to serialize access to data structures. If two threads, Thread₀ and Thread₁, executing on the same core access such a data structure, then one of them, say Thread₀, attains access by executing an atomic test-and-set operation, which tests the availability of the associated lock and sets it. If the other thread, Thread₁, accesses the data structure while Thread₀ owns the lock, then Thread₁ busy-reads the lock variable, consuming processor resources and slowing down the progress of Thread₀. In such situations, the operating system decreases the priority of the thread waiting for the spinlock and resets its priority when the process acquires the lock. Additionally, AIX increases the priority of the thread holding the lock and resets it when the thread gives up the lock.

A thread executes the operating system idle loop if there are no ready-to-run tasks, i.e., the thread spins in a loop continuously checking the head of the run queue for ready-to-run tasks. This consumes valuable processor resources and can slow down an application thread executing on core's other hardware thread. In this case, both operating systems decrease the priority of the thread executing the idle loop to the lowest priority possible, thus, minimizing its interference with the other thread.

3 Related Research

Simultaneous multithreading (SMT) was first introduced in 1995 in the form of an Alpha processor simulator [3]. Subsequently, SMT has been implemented in processors from Sun [4, 22], Intel [5-8], and IBM [10, 21]. Since their introduction, researchers have proposed methodologies to improve the throughput of SMT processors. For example, related research [23, 25-29, 30-35, 57] has introduced techniques for forming co-schedules and, given a co-schedule, improving its performance.

Only the work of C. Boneti and his colleagues [33-35] study the use of hardware thread priorities in SMT processors to enhance performance but, as described in Section 3.2, the goal of their research is quite different from ours. With respect to application characterization our work is most similar to that of Doucette, et al. [27], which was published in a workshop in 2007 and of which we were not aware until recently. It uses the availability of a set of microarchitectural resources to form intensity base vectors, which are similar, in spirit, to our shareable resource signatures. As described in Section 3.1, the goal of their research also is quite different from ours. In addition, similar to [27] and [34], we use crafted microbenchmarks that stress processor core resources. Below we summarize the major differences between our work and related research, while additional differences as well as similarities are presented in Sections 3.1 and 3.2. Unless otherwise noted, in this chapter the term co-schedule refers to a co-schedule of two applications.

In terms of methodology, the performance metrics we use to characterize application threads (henceforth called applications) depend on the characteristics of the processor architecture and, thus, it is not surprising that our set of metrics for the IBM POWER5 implementation, presented in Chapter 6, differ from those used in [23, 25-29, 30-32, 57], which use other processors for validation purposes. Only the work of C. Boneti and his colleagues study application characterization [33, 35] on the IBM POWER5. Their methodology, which is described in [33] and [35], and summarized in Section 3.2, characterizes individual MPI tasks of data-parallel applications that iterate multiple times over two phases. In the first phase the tasks perform computations – this is called the *work phase*, and after performing the computations, in the second phase the tasks wait at a barrier to synchronize – this is

called the *communication phase*. They characterize applications using the time spent in each of these two phases. In contrast, we characterize applications based on their utilization of critical shareable core resources in single-threaded mode. Similarly, [25, 27, 30, 31, 57] characterize applications during their execution in single-threaded mode, while [23, 28, 29, 32-35] characterize applications during their execution in SMT mode. We characterize an application during its single-threaded mode execution, rather than its SMT mode execution, because the latter is dependent on the behavior of the co-scheduled application. Our methodology and that of [23, 29, 30-35] characterize each interval of application execution time, whereas [25, 26-28, 57] capture metrics for an application's entire execution time. The finer granularity allows us to capture changes in an application's execution behavior with respect to its use of critical shareable core resources.

In the following sections, we further compare our work to the related research that is presented in this chapter. In particular, in Section 3.1 we review research that studies job-scheduling methodologies, i.e., methodologies to form co-schedules from the applications in a job queue and in Section 3.2 we explore thread-throttling mechanisms that are meant to improve the throughput of a given co-schedule. Finally, Section 3.3 summarizes the related research and points out how it differs from ours.

3.1 JOB SCHEDULING

Several researchers have studied job scheduling on SMT processors [23, 25, 27-29, 32, 57]. The general idea of this research is to form co-schedules from the entries in a job queue and select that which has best throughput among all possible co-schedules. In contrast, given a co-schedule, our work attempts to intelligently set hardware thread priorities to optimize attained throughput. As such, our research is complementary to much of the research described in this section. It can be used in conjunction with many of the techniques described herein to improve throughput. In the following discussion, we further differentiate our work from the related research presented in this section.

Snively, et al., using an SMTSIM Alpha simulator, describe an operating system (OS) process scheduler, i.e., the SOS (Sample, Optimize, Symbios) "symbiotic" scheduler [23]. This job scheduler attempts to predict the best co-schedule among the available jobs in the job queue. For prediction

purposes, the scheduler randomly selects a set of 10 different co-schedules and executes each one such that each job in a co-schedule executes for five million cycles. During this phase, called the sampling phase, for each co-schedule, the scheduler periodically records processor-wide metrics such as IPC and data cache misses. Next, during the optimize phase, it analyzes the metrics of the 10 co-schedules to predict the best co-schedule among them, which is run in the symbiosis phase for two billion cycles. The authors' experiments indicate that co-schedules with the lowest standard deviation of IPC across different measurement periods in the sample phase performed the best. Hence, they recommend picking the co-schedule that has the lowest standard deviation of IPC. The SOS symbiotic scheduler increased the performance of the SPEC CPU2000 benchmark suite by 17% over that achieved using a naïve scheduler. This methodology evaluates the quality, in terms of performance, of selected co-schedules using IPC and data cache activity metrics. In contrast, our methodology predicts the best priority pair for a given co-schedule using individual application performance metrics, associated with components of the memory hierarchy and the execution pipeline, collected in single-threaded mode. In our work the IPC metric is used to evaluate the accuracy of our predictions.

Nakajima, et al. [25] propose a process scheduling optimization for non-multiprogrammed workloads executed on a dual-core, Intel Xeon Hyper-Threading processor (Hyper-Threading is the trademark name used by Intel for SMT). This optimization forms co-schedules that are meant to help the applications comprising the co-schedules attain their processor resource requirements and, thus, improve performance. Two metrics are used to determine an application's processor resource requirements, i.e., number of floating-point operations and number of L2 cache hits per second of single-threaded execution. The applications are pre-profiled in single-threaded mode and the metrics are stored for use by the scheduler. The way the scheduler works is as follows. In SMT mode four applications are executed on four hardware threads of the dual-core processor, each core has two hardware threads. The scheduler monitors, during each scheduling time-slice of 10 milliseconds, the two metrics for each of the four applications. At the end of each time slice, for each application, it compares the metrics recorded in SMT mode with those captured in single-threaded mode and determines if the application is attaining its resource requirements. If this is not the case, it forms co-schedules of applications such that an

application with high utilization of a given resource is paired with an application with low utilization of the given resource. Using this scheduler, the authors increased the performance of the SPEC CPU2000 benchmarks by 6% over the naïve scheduler. Similarly, our methodology, given a co-schedule, determines one of the application's shareable core resource requirements and, compares this with those of the co-scheduled application. However, rather than using this information to determine if the applications comprising a co-schedule are meeting their resource requirements, we use it to determine the setting of hardware thread priorities for the given co-schedule. Although, for the POWER5 we do not implement dynamic setting of hardware thread priorities for co-schedules of applications with multiple signatures, our predictions could be used for co-schedules of such applications.

Doucette, et al. [27] present a methodology that predicts the relative slowdown of a given target application in a co-schedule executed on an UltraSPARC T1 processor. An application is characterized according to its execution-time sensitivity and intensity with respect to a set of critical microarchitectural resources. This is done by forming two base vectors for each application, i.e., a sensitivity vector and an intensity vector. The base vectors have an element for each of the following resources: floating-point unit, L1 instruction cache, L1 data cache, L2 cache for instructions, and L2 cache for data. To form these base vectors, a set of microbenchmarks, each of which stresses a unique shareable SMT core resource, is used. The sensitivity of an application, $Application_A$, to a particular resource, $Resource_x$, is measured by first executing $Application_A$ with the microbenchmark that stresses $Resource_x$, i.e., $Microbenchmark_x$. Next, the resultant execution time of $Application_A$ in SMT mode is compared with the execution time of $Application_A$ executed alone on the SMT core. The calculated slowdown in execution time is an element in the application's sensitivity base vector. Similarly, the intensity of $Application_A$ w.r.t. $Resource_x$ is measured by comparing the resultant execution time of $Microbenchmark_x$ in SMT mode executed with $Application_A$ to the execution time of $Microbenchmark_x$ executed alone on the SMT core. This is an element in the application's intensity base vector. Given a possible co-schedule of two applications, $Application_A$ and $Application_B$, and their sensitivity and intensity base vectors, the slowdown of $Application_A$ is calculated by the dot product of $Application_A$'s sensitivity base vector and $Application_B$'s intensity base vector. The authors predict that $Application_A$

will incur minimum slowdown when co-scheduled with an application with which the resultant dot product is minimized. To test their methodology the authors predict the minimum slowdown of co-schedules formed using six SPEC CPU2000 benchmarks executed on one core of a Sun UltraSPARC T1 processor. In this research only two out of the four hardware threads were used for experiments. For four out of the six benchmarks, their prediction accurately identified the co-schedule that resulted in the minimum slowdown of the given target application in a co-schedule. Except for the research goal, there are many parallels between our research and that described in [27]. Note, however, that the research presented in this dissertation was done without knowledge of this previous research. Like this research, we approximate the availability of a set of critical shareable SMT core resources and use that information to form signatures, which are similar, in spirit, to the authors' intensity base vectors. Where our work differs is in the approach we use to form signatures. We estimate the utilization of critical shareable SMT core resources using performance counters rather than calculating the relative difference between the execution times of base vector microbenchmarks and co-schedules of the base vector microbenchmarks and the target application. Another difference between our work and that described in [27] is that we capture signatures periodically during application execution, rather than estimating resource utilization based on the application's entire execution. This finer granularity allows us to detect changes in application behavior, allowing us to more accurately identify the "best" settings of hardware thread priorities throughout an application's execution. It is likely that this finer granularity also could provide more accuracy in terms of co-schedule formation.

Moseley, et al. [28] propose a methodology to form co-schedules from the applications in a job queue based on prediction of the aggregate IPC of each possible co-schedule. The authors compare the accuracy of two different statistical methods, linear regression and recursive partitioning, to predict aggregate IPC using performance counter data (TLB, caches, integer unit, floating-point unit, re-order buffer, and issue queues) associated with co-schedules formed from the SPEC CPU2000 benchmarks executed on an Intel Pentium 4 Hyper-Threading processor. They found that linear regression was more accurate than recursive partitioning across all possible co-schedules, whereas both methods had comparable accuracy for co-schedules formed from two SPEC INT benchmarks or two SPEC FP

benchmarks. Our research also uses performance counter data, but instead of using the data to predict aggregate IPC, we use the data to form signatures, which are used to predict best hardware thread priorities.

Using the NAS OpenMP benchmarks, McGregor, et al. [29] study the behavior of their OS process scheduler for a machine with four Intel Xeon Hyper-Threading processors, i.e., eight hardware threads. The research evaluates the effectiveness, in terms of optimizing utilization of memory bandwidth, of co-schedule formation and the enabling and disabling of Hyper-Threading. The following three metrics are used for the evaluation: number of L3-cache misses, number of bus transactions, and number of stall cycles experienced by an application in SMT mode. Each of the three metrics is evaluated independently to determine the metric that is the most effective in optimizing utilization of memory bandwidth. The metric being evaluated is used to adjust the number of running processes and the co-schedule assigned to the two hardware threads of each processor. The co-schedule formation policy first evaluates, based on the metric being evaluated, if a particular application should run in single-threaded mode or in SMT mode. Once the number of processes to schedule is determined and SMT mode is selected, co-schedules are formed of an application with a high value of the metric being evaluated and an application with a low value. In their experiments the authors found that the number of bus transactions metric gave the best predictions of co-schedules that optimize memory bandwidth utilization. In this work, L3-cache miss, bus transaction, and stall cycle activity were evaluated to choose SMT or single-threaded mode, and if SMT mode is chosen, to form co-schedules. We too use application-specific metrics, albeit different ones. However, instead of measuring them in SMT mode, we measure them in single-threaded mode because our goal is to characterize an application's utilization of shareable SMT core resources, which changes when executed with another application in SMT mode and is dependent on the characteristics of the co-scheduled application. The NAS benchmarks also are used in the evaluation of our best priority pair prediction methodology; however, we use the sequential benchmarks rather than the parallel ones.

Bulpin, et al. [32] implemented a process scheduler for an Intel Pentium 4 Hyper-Threading processor, which has only one core with two hardware threads. Given an application executing on a

hardware thread, the scheduler forms a co-schedule by selecting another application from the job queue (i.e., the set of candidate applications) that is predicted to form a co-schedule with best system speedup. The system speedup of a co-schedule is the sum of the performance ratios of the co-schedule's two applications, where the performance ratio of an application in a co-schedule is the ratio of its execution time in SMT mode to its execution time in single-threaded mode. Prediction of the co-schedule with best system speedup is based on an analytical model that is developed using the following process. All possible co-schedules formed from pairing the given application and one of the candidate applications are executed in SMT mode. The co-schedules are executed in order to record the execution time of each co-schedule's applications in SMT mode and to characterize each application in each co-schedule in terms of L1- and L2-cache miss rates, the number of instructions completed, and the number of floating-point operations. Additionally, after executing each co-schedule's applications in single-threaded mode, the system speedup of each co-schedule is calculated. Using the four performance metrics and the system speedup metric associated with each possible co-schedule, an analytical model was developed that, given a co-schedule, takes as input the four metrics of each application in the co-schedule and outputs the predicted system speedup of the co-schedule. To evaluate their model, the authors built a process scheduler that for each co-schedule that has executed on a system keeps a history in terms of the four metrics for each application in the co-schedule. When an application must be selected to execute with the application that is currently running on the core, the scheduler looks at the history of co-schedules, which includes the currently running application, and selects one that is predicted to attain best system speedup. This scheduler was implemented in the Linux 2.4.19 kernel; rather than build the scheduler from scratch, the authors exploited the Linux scheduler's assignment of dynamic priorities to processes. In Linux, the dynamic priority of a process in the job queue determines the job that will be chosen for execution; the process with the highest dynamic priority is scheduled. To implement the new scheduler, the Linux scheduler was modified so that when a scheduling decision needs to be made, the scheduler assigns the highest dynamic priority to the process executing the application that is predicted to form the co-schedule with the best system speedup. For the SPEC CPU2000 benchmark suite, this scheduler increased overall system performance by 3.2% over the native Linux 2.4.19 process scheduler.

Instead of changing priorities at the operating system scheduler level, our methodology changes priorities at the hardware level, providing finer-grain control. At the operating system level, process priorities can be adapted at the granularity of a time slice, whereas at the hardware level, process priorities, potentially, depending on the cost of adaptation, can be adapted at finer granularities. Thus, priorities at the hardware level may provide more possibilities for performance enhancement. Similar to our methodology, this work considers the single-threaded execution of applications as compared to their execution times when executed as part of co-schedules in SMT mode. In terms of the metrics used for application characterization, both this work and ours use L2-cache and floating-point unit activity; in contrast, we do not use the number of L1-cache misses and the instruction count.

Shepelov, et al. [57] implemented a process scheduler for processors with heterogeneous cores using architectural signatures of applications – the goal is to schedule processes to attain best performance. The architectural signature of an application is formed using an application’s memory re-use distance. The re-use distance of a memory location indicates the number of unique accesses between consecutive accesses to the memory location. The application is pre-profiled during a pilot execution to collect information that can be used to measure an application’s re-use distance. The re-use distance is used to predict, for any given memory hierarchy, the application’s last-level cache miss rate. The predicted last-level cache miss rate is the only element of the architectural signature of an application. To estimate the execution time of an application, the authors assume that all applications have a base IPC of 1.5 on any core. For an application that executes n instructions on a core with clock frequency f , the base execution time is $(n / 1.5) / f$. The predicted execution time of an application on a core is calculated as the sum of the base execution time and the time required to service last-level cache misses from main memory. The authors implemented a scheduling algorithm that assigns processes to cores that vary in clock frequency and memory hierarchy configuration. Using predicted execution time, the scheduler assigns processes to cores. For the SPEC CPU2000 benchmark suite, the scheduler improved performance by 13% over the default scheduler, which randomly binds processes to cores. We also use architecture-sensitive signatures but the definition is quite different. Instead of the last-level cache miss rate, our signature characterizes an application’s utilization during an execution time interval, when

executed in single-threaded mode, of critical shareable SMT core resources that effect throughput. Of course, the last-level cache miss is one such resource.

3.2 THREAD THROTTLING

Given a co-schedule, thread-throttling mechanisms have been studied to address the issue of fairness [26, 30], prioritizing the throughput of one application of a co-schedule [31], and overall throughput [33-35]. In this context, fairness has been defined as access to SMT processor resources such that applications executed as a co-schedule in SMT mode experience the same relative performance as compared to their execution in single-threaded mode. The goal of our research is different from that of [26, 30, 31] and most similar to that of [33-35], which also use POWER5 hardware thread priorities to attempt to improve SMT processor throughput. In our discussion of [33-35], we clearly compare our work with theirs. In the following discussion, we further differentiate our work from the other related research presented in this section.

Fedorova, et al. [30] designed an OS process scheduler for an UltraSPARC T1 execution-driven simulator. This work allocates time slices to an application based on its fair share of the L2 cache, e.g., in an n -threaded system, the performance obtained with a cache of $1/n^{th}$ the size. Using a detailed regression model, the expected IPC of a process is dynamically predicted based on it having its fair share of the L2 cache. If an application is not meeting the predicted IPC then additional time slices are allocated to the application until the predicted IPC is achieved. Similarly, if an application exceeds its predicted IPC, then fewer time slices are allocated until the predicted IPC is achieved. For the SPEC CPU2000 benchmarks the author's process scheduler improved performance in the range of 3% to 56% over the native Solaris scheduler. The only similarity to our work is the goal of improved performance and the measurement of that performance in terms of IPC.

Grunwald, et al. [26] study the issue of fairness between co-schedules executed concurrently on two hardware threads of an Intel Pentium 4 with Hyper-Threading. Their study shows that malicious code executed by one application of a co-schedule can slow down the execution of the other by a factor of 10 to 20. The authors recommend stalling the issue of instructions from the offending application

until fairness is restored. Similarly, our methodology uses control of allocated decode cycles to throttle the execution of a hardware thread; this mechanism can be used to stall an application's instruction issue. As noted previously, we use this mechanism to optimize SMT core throughput rather than to maintain fairness.

Cazorla, et al. [31] study a mechanism to execute a target application at a specified fraction of its IPC in single-threaded mode by throttling the performance of a co-schedule comprising the target application and another application executed on two hardware threads of an SMTSIM simulator. To meet this goal, application performance is throttled by the simulator's allocation of issue bandwidth, fetch bandwidth, functional units, rename registers, and issue queue entries. Similar to [23], the methodology consists of two phases: (1) a sample phase of 60,000 cycles to determine the target application's single-threaded performance, and (2) a subsequent tuning phase of 1.2 million cycles that, every 15,000 cycles, varies the amount of resources allocated to the co-schedule to meet the target application's specified fraction of single-threaded IPC. Using the SPEC CPU2000 benchmarks, the authors were able to meet 10%-90% of single-threaded IPC. Like this work, we characterize an application in terms of its single-threaded mode performance metrics and we throttle an application's access to resources. However, instead of doing this via allocation of issue and fetch bandwidths, functional units, rename registers, and issue queue entries, we do it through allocation of decode cycles via intelligent assignment of hardware thread priorities.

In [33-35] Boneti, et al. studied the use of IBM POWER5 hardware thread priorities to improve the application execution times. In [33] they demonstrate the effectiveness of hardware thread priorities in reducing the execution time of a message-passing program in which tasks wait at an MPI barrier. Like a sequential application, a task is a process that can be scheduled to run on a hardware thread. Using their SIESTA application with a level of parallelism of four executed on two cores, each with two hardware threads, they experimented with assigning hardware thread priorities in the range of 1 to 6 to each of the four hardware threads. Their experiments showed that by assigning hardware thread priority 5 to task that spend more time in the compute phase, as compared to other tasks, and assigning 4 to other tasks barrier wait times can be reduced and, thus, performance improved. For the SIESTA application

this improves performance by 8%. In [34] Boneti, et al. characterize the impact of POWER5 hardware thread priorities on the execution times of crafted microbenchmarks as well as overall processor throughput. Their experiments use microbenchmarks that stress one of the following: fixed-point unit, floating-point unit, L1-cache, L2-cache, and main memory. They execute all possible co-schedules of their microbenchmarks at all available priority pairs. First, for each application in a co-schedule, they analyze the relative difference between the execution time achieved using non-equal priorities and that achieved using default priorities. They conclude that using a priority difference of two is sufficient to attain performance improvements that are only 5% below the maximum improvement that can be attained. They also calculate and analyze the relative improvement in the overall throughput of a co-schedule at non-equal priorities, as compared to that attained at equal priorities. This study indicates that using a priority difference of five can increase overall throughput by as much as 23% over that attained with equal priorities, whereas using a priority difference of two can improve overall throughput by 7.2% over that attained with equal priorities. This dissertation also studies the impact of POWER5 priorities on overall throughput; our pilot study, which is presented in Chapter 4, was published prior to the publication of [34]. The experiments associated with our pilot study also show that a thread priority difference of five has a significant impact on co-schedules comprising applications in the SPEC CPU2000, SPEC CPU2006, Imbench, and stream benchmark suites. In that study, described in Chapter 4, we found that using a priority difference of five improved throughput by as much as 35% over that attained with equal priorities, whereas using a priority difference of two improved throughput by as much as 20% over that attained with equal priorities. In addition, in our implementation on the IBM POWER5, explained in Chapter 6, a priority difference of five improved throughput by as much as 16.42%. Finally, in [35] Boneti, et al. develop an operating system scheduler in Linux that automatically sets hardware thread priorities for co-schedules of tasks of an MPI application. The execution flow of many MPI applications comprises multiple iterations of a compute phase followed by a communication phase. A task is assigned a subset of the iterations. For each such iteration, the scheduler calculates a task's CPU utilization as the ratio of the wall clock time spent in the compute phase to the wall clock time spent waiting for completion of the communication phase. For each task's iterations, the scheduler

keeps track of CPU utilization and assigns one of three thresholds to each task, which indicates if the task has low, medium, or high CPU utilization. In [35] these thresholds are empirically set to less than 65%, between 65% and 85%, and greater than 85%, respectively. If the two tasks that form a co-schedule have different utilization levels, then the task with lower CPU utilization is assigned a higher hardware thread priority than the task with higher CPU utilization. Such priority assignments are meant to reduce the imbalance in CPU utilization between the two tasks and, thus, could cause tasks to reach MPI barriers more or less at the same time, thus, reducing waiting time. The authors introduce two different heuristics to predict the CPU utilization of the next iteration of a task and, thus, to automatically adapt hardware thread priorities during runtime: a uniform heuristic and an adaptive heuristic. The uniform heuristic uses the cumulative CPU utilization of all the past iterations of the two tasks that form the co-schedule to predict the co-schedule's CPU utilization in the next iteration. The uniform heuristic is intended for applications that have uniform behavior in terms of task CPU utilization. The adaptive heuristic predicts a co-schedule's CPU utilization in the next iteration using the sum of weighted values of the CPU utilization of the last iteration and the cumulative sum of all but the last iteration; the weights were determined empirically. The adaptive heuristic is intended for applications with a task CPU utilization that changes over time. Using the adaptive heuristic and the SIESTA application, the scheduler was able to improve execution time by 6% over the default case of running hardware threads with equal priorities, as compared to the 8% improvement that was achieved by empirically setting hardware thread priorities in [33]. Like the research discussed in [33] and [35], we measure an application's utilization of CPU resources, in particular, the critical shareable resources of a core of the POWER5 during each execution time interval. The length of the execution time interval for our POWER5 implementation was determined as one second. In contrast, the CPU utilization metric used in [33] and [35] measures the ratio of computation to communication time for each time interval, where a time interval is defined as an iteration of the SIESTA application.

3.3 SUMMARY

Related research [23, 25-29, 30-35, 57] in SMT has been done to form co-schedules with best throughput and, given a co-schedule, to improve its performance. The research described in [23, 25, 27-29, 32, 57] attempts to form co-schedules that give the best throughput among all possible co-schedules of entries in the job queue. In contrast, given a co-schedule of two sequential applications, we predict the best priority pair that improves overall throughput. As such our work can be used in conjunction with work that forms best co-schedules. Other related work, like ours, has investigated improving the performance of a given co-schedule. The issue of fairness of a co-schedule was investigated in [26, 30], whereas [31] prioritizes the throughput of one application of a co-schedule, and [33] and [35], as well as our work, attempt to improve the overall throughput of a given co-schedule.

Table 3.1: Characteristics of Related Research and our Research

Authors	Improve Through-put via Co-schedule Formation	Improve Through-put of a Given Co-schedule	Provide Fairness to a Given Co-schedule	Guaranteed Through-put for one Application of a Given Co=schedule	ST Mode Application Characterization	SMT Mode Application Characterization	Application Characterization of Entire Execution	Application Characterization of Intervals of Execution
Meswani		√			√			√
Snively [23]	√					√		√
Nakajima [25]	√				√		√	
Grunwald [26]			√			√	√	
Doucette [27]	√				√		√	
Moseley [28]	√					√	√	
McGregor [29]	√					√		√
Fedorova [30]			√		√		√	
Cazorla [31]				√	√			√
Bulpin [32]	√					√		√
Boneti [33]		√				√		√
Boneti [34]		√				√	√	
Boneti [35]		√				√		√
Shepelov [57]	√				√		√	

Table 3.1 enumerates characteristics of our methodology and that of related research. In this table, the first column refers to the research being characterized, while a check (✓) in the remaining columns indicates features of the associated research. A check in column two indicates that the research was used to form best co-schedules, while a check in column three, four, or five indicates that given a co-schedule the research attempted to improve overall throughput, provide fairness, or guarantee throughput to one application of a co-schedule, respectively. A check in column six or seven indicates that the research characterized applications based on their execution in single-threaded mode or SMT mode, respectively. Finally, a check in one or the other of the last two columns indicates that the research characterized applications based on their entire application execution times or based on their execution time intervals, respectively. For example, as shown in this table, our methodology is used to improve the throughput of a given co-schedule using application characterizations of the applications' execution time intervals during their individual executions in single-threaded mode. As shown in this table, while our research shares some features of related research, it differs from each of the related research initiatives in at least one feature.

Table 3.2: Metrics used by Related Research and our IBM POWER5 Implementation

Author	Wall Clock Time	IPC	Stall Cycles	L1 Cache	L2 Cache	L3 Cache	Memory Bus	Integer Unit	FP Unit	TLB	Issue Queue	Number Instructions	Re-use Distance
Meswani					✓			✓	✓	✓			
Snavely [23]		✓		✓	✓								
Nakajima [25]					✓				✓				
Grunwald [26]		✓											
Doucette [27]				✓	✓				✓				
Moseley [28]				✓	✓			✓	✓	✓	✓	✓	
McGregor [29]			✓		✓		✓						
Fedorova [30]					✓								
Cazorla [31]		✓											
Bulpin [32]				✓	✓				✓			✓	
Boneti [33]	✓												
Boneti [34]		✓											
Boneti [35]	✓												
Shepelov [57]													✓

Table 3.2 lists the metrics used to characterize applications in related research as well as in our IBM POWER5 implementation, described in Chapter 6. In this table, the first column refers to the research being described, while a check (✓) in the remaining columns indicates the metrics used by the associated research. A check in columns two through 14 indicates that the research used wall clock time, IPC, number of CPU stall cycles, number of L1 cache accesses, number of L2 cache accesses, number of L3 cache accesses, number of memory bus transactions, integer unit usage, floating-point unit usage, translation lookaside buffer (TLB) usage, issue queue usage, total number of instructions completed, and memory re-use distance metrics, respectively. For example, our implementation on the IBM POWER5 used the following metrics: L2 cache accesses and utilization of the integer unit, floating-point unit, and TLB. Since the metrics used to characterize applications depend on the processor architecture, as shown in Table 3.2, we share many metrics with that of related research and, in particular, we use a subset of the metrics used by [28].

Our best priority pair prediction methodology, which is based on the notion of Shareable Resource Signatures, characterizes the critical SMT core resource utilization of applications executed in single-threaded mode. Only the work of [27] presents an application characterization methodology based on utilization of such resources. Although this work has many parallels to ours, we use signatures to predict the best priority pair, while [27] uses them (called intensity vectors in [27]) to estimate the relative slowdown of an application of a given co-schedule and to form application co-schedules. Our characterization of an application’s utilization of critical shareable SMT core resources, i.e., an application’s Shareable Resource Signature, is similar to the intensity vector described in [27]. Like [27], we characterize an application’s utilization of these resources in single-threaded mode and, as shown in Table 3.2, the resources used in our IBM POWER5 implementation are the L2 cache, TLB, integer unit, and floating-point unit, whereas [27] uses the L1 cache, L2 cache, and floating-point unit. Both the utilization information captured by our signature and the intensity vector of [27] provide estimates of the availability of the targeted resources for the use of the other application of a given co-schedule. Unlike [27], as noted in Table 3.1, we associate a signature with each interval of an application’s execution time; for the IBM POWER5 implementation this interval was one second. This

fine granularity can facilitate the dynamic adaptation of priorities as an application's signature changes. Note however, in this dissertation we have not implemented dynamic adaptation of priorities and leave it as future work.

Only the related research described in [33, 35] investigated the use of IBM POWER5 hardware thread priorities. In their research they use priorities to improve the throughput of MPI applications that employ barrier synchronization. In contrast, in our IBM POWER5 implementation, we use priorities to improve the throughput of a pair of sequential applications. Of course, their methodology also could be used to improve the throughput of co-schedules comprising two sequential applications, which is the goal of our research. The methodology adopted in [33] and [35] uses CPU utilization to predict the settings of hardware thread priorities that reduce the imbalance of CPU utilization among MPI tasks, which can lead to improved performance. Instead, we use finer grain metrics to predict the settings of hardware thread priorities. These are used to estimate the utilization of four critical resources of an IBM POWER5 core. Using finer grain metrics we observe that, for the IBM POWER5, intelligent setting of hardware thread priorities are the most beneficial when applications have higher utilization of the floating-point unit versus the fixed-point unit. We observe that while some applications have a near constant degree of utilization of critical resources there are many applications whose utilization of resources varies during their execution. Using our prediction methodology, which is based on these fine-grain metrics, we show the efficacy of improving IBM POWER5 throughput for co-schedules of applications, each of which can be characterized well by a single signature. Moreover, although we do not demonstrate it in this dissertation, our methodology can be used to adapt hardware thread priorities for applications with multiple signatures.

Note that signatures have been used in other contexts besides the ones described in this section. Snively, et al. [36] predict application performance using machine signatures and application profiles. In that study, the authors define machine signature as rates at which a machine can sustain loads and stores for various memory-access patterns and request sizes. An application profile stores information about load and store patterns and load-store rates of application code.

4 Pilot Simulation Study

To assess the potential merit of using SMT hardware thread priority adaptation to enhance overall processor throughput, we performed a pilot performance study, which is described in Section 4.1. This study is based on simulations performed on the IBM POWER5 trace-driven simulator discussed in Section 4.2. As described in this section, the effectiveness of 11 hardware thread priority pairs is investigated. The simulations were driven by instruction traces that were created using the benchmarks presented in Section 4.3 and the instruction trace generation process described in Section 4.4. The experimental design of the simulation study, including the definition of the workloads, is described in Section 4.5, while the experimental results and conclusions of the study are discussed in Sections 4.6 and 4.7, respectively.

4.1 PERFORMANCE STUDY

The goal of this study is to determine if the IBM POWER5 default (equal) priority pair always yields the best SMT processor throughput (defined below). In other words, the study investigates the potential merit of using non-default priority pairs to enhance SMT processor throughput. For this study we used an IBM POWER5 simulator and partial instruction traces from real applications to drive the simulations. Each application trace co-schedule was run under 11 different priority pairs and a best priority pair, among the 11, was identified. As described in Section 4.4, the trace lengths used in this study are between 17 million and 110 million instructions and, hence, these traces may not be representative of the applications from which they were derived and, accordingly, the results obtained using the traces may not be applicable to the applications. To address this shortcoming, in this dissertation our implementation of the best priority pair prediction methodology, described in Chapter 6, was validated using a real machine and entire application executions.

4.2 IBM POWER5 SIMULATOR AND HARDWARE THREAD PRIORITIES INVESTIGATED

The IBM POWER5 [17] trace-driven simulator was used in this study. It simulates the execution of one core of an IBM POWER5 processor, either in SMT mode or single-threaded mode. In SMT mode

the simulator terminates a simulation when it finishes the processing of one of the two input traces, say Trace_0 . The resultant SMT mode performance data, e.g., the number of instructions executed by each thread of the co-schedule, relates to the instructions of Trace_0 and the instructions of the second trace, Trace_1 , that were processed before the simulation terminated.

Since the goal of this pilot study requires measurement of the throughput of each trace co-schedule, under each priority pair, the performance data associated with the total execution of each co-schedule were required. To collect this information we also used the simulator to simulate, in single-threaded mode, for each co-schedule, the instructions of Trace_1 that were not processed. This simulation was configured to skip the number of instructions of Trace_1 that were already simulated in SMT mode. Accordingly, the reported execution time for each trace co-schedule is the sum of the execution times in the associated SMT-mode and single-threaded mode simulations. For example, assume that Trace_0 consists of 200 instructions and Trace_1 consists of 300 instructions. Further assume that the SMT-mode simulation completes the processing of Trace_0 before Trace_1 and at this point, when the simulation terminates, only 180 instructions of Trace_1 are simulated. To simulate the remaining 120 instructions of Trace_1 , another instance of the simulator in single-threaded mode, with 180 as the parameter that identifies the number of instructions to skip, is used.

In SMT mode, hardware thread priorities can be controlled by software in both user mode and supervisor mode. The priorities that can be set by each are described in Section 2.5 of Chapter 2. The simulated system has a 4GB main memory and a 70GB hard disk, and runs in SMT mode with specified priorities assigned to the two hardware threads of the core. The relevant command-line arguments required to run the simulator in SMT mode are the following:

- **Trace0**: the trace to be executed on Thread_0
- **Trace1**: the trace to be executed on Thread_1
- **num_inst**: the number of architected instructions to simulate before exiting
- **t0_prio**: priority of Thread_0
- **t1_prio**: priority of Thread_1

The relevant command-line arguments required to run the simulator in single-threaded mode to simulate the execution of only the unprocessed instructions of the unfinished trace are the following:

- **Trace:** the unfinished trace to be executed
- **num_inst:** the number of architected instructions to simulate before exiting
- **num_inst_to_skip:** the number of instructions of the unfinished trace to skip before starting the simulation

This study experiments with hardware thread priorities 2 through 7 (not 0 or 1). As described in Chapter 2, when both threads have a hardware thread priority greater than one, the difference in hardware thread priorities determines the ratio of decode cycles allocated to each thread. Accordingly, given priorities 2 through 7, there are 11 unique hardware thread priority pairs that cover all possible differences and, thus, ratios. These are listed in Table 4.1 along with the associated allocation of decode cycles. For example, if the difference in priorities is 3, Thread₀ is permitted to decode 15 instructions, after which Thread₁ is permitted to decode 1. If the difference is -2, Thread₀ is permitted to decode 1 instruction, after which thread₁ is permitted to decode 7. Each thread co-schedule is simulated under all 11 priority pairs.

Table 4.1: Hardware Thread Priority Pairs used in Pilot Simulation Study

Thread ₀ Priority	Thread ₁ Priority	Difference between Thread ₀ and Thread ₁ Priorities	Thread ₀ Decode Cycles	Thread ₁ Decode Cycles
7	2	5	63/64	1/64
7	3	4	31/32	1/32
7	4	3	15/16	1/16
4	2	2	7/8	1/8
4	3	1	3/4	1/4
4	4	0	1/2	1/2
3	4	-1	1/4	3/4
2	4	-2	1/8	7/8
4	7	-3	1/16	15/16
3	7	-4	1/32	31/32
2	7	-5	1/64	63/64

4.3 BENCHMARKS AND TRACES

The IBM POWER5 simulator is driven by POWER5 instruction traces. For this study, the traces were generated from benchmarks in the SPEC CPU2000 [37], SPEC CPU2006 [38], stream2 [39], and lmbench [40] benchmark suites, which are described in Sections 4.3.1, 4.3.2, 4.3.3, and 4.3.4, respectively. The benchmarks in the four suites are purported to be either compute intensive (SPEC CPU2000 and SPEC CPU2006) or memory intensive (stream2 and lmbench). It is common thinking that compute-intensive applications spend a significant amount of execution time performing computations on the processor, i.e., they are CPU-bound. In addition, it also is commonly thought that memory-intensive applications spend a significant amount of execution time accessing memory, i.e., they are memory-bound. Although the meanings of the terms compute- and memory-intensive when applied to an application’s execution behavior are not agreed upon, we use these pre-existing labels for the benchmarks in our discussions. We do not verify the labeled “intensiveness” of the benchmarks.

As described in the following subsections, the compute-intensive benchmark suites are comprised of floating-point and integer benchmarks, referred to as SPEC CPU2000/2006 Floating-Point, and SPEC CPU2000/2006 Integer. The memory-intensive benchmark suites are further divided into benchmarks that stress the on-chip caches and those that stress the L3 cache and main memory. Accordingly, the benchmarks used in the study fall into four application classes: floating-point intensive, integer-intensive, on-chip cache-intensive, and off-chip memory-intensive. This permits the study of (1) *homogeneous trace co-schedules*, where the traces are from two *homogeneous applications*, i.e., both applications in the pair are deemed to stress the same resource from the following list: floating-point units (FPUs), integer units (FPUs), on-chip caches, or off-chip caches and memory; and (2) *heterogeneous trace co-schedules*, where the traces are from two *heterogeneous applications*, i.e., each application is deemed to stress a different resource class from the list of resources stated earlier. For example, a homogeneous trace co-schedule could comprise traces from two different floating-point intensive applications, which are assumed to stress the FPUs, while a heterogeneous trace co-schedule could comprise a trace from a floating-point intensive application and a trace from an off-chip memory-intensive application, which is assumed to stress the L3 cache and main memory.

4.3.1 SPEC CPU2000

The benchmarks that comprise the SPEC CPU2000 suite [37] have working sets that fit in main memory and include both floating-point and integer-intensive applications, i.e., the SPEC CPU2000 Floating-Point (SPEC2000fp) and SPEC CPU2000 Integer (SPEC2000int) benchmarks. To constrain simulation time, a subset of benchmarks were selected from the suite, i.e., those with one input file. Given one input file per benchmark, we captured one trace per benchmark instead of one per benchmark-input pair. Table 4.2 lists the benchmarks used in the simulation study.

4.3.2 SPEC CPU2006

The benchmarks that comprise the SPEC CPU2006 [38] suite include both floating-point and integer applications, i.e., SPEC CPU2006 Floating-Point (SPEC2006fp) and SPEC CPU2006 Integer (SPEC2006int) benchmarks. Due to simulation time constraints we used the SPEC CPU2006 benchmarks that are the closest, in terms of the problems they solve, to the selected SPEC CPU2000 benchmarks. The benchmarks used in the study are listed in Table 4.3.

Table 4.2: SPEC CPU2000 Benchmarks used in Pilot Simulation Study

Name	Type	Language	Category	Description
176.gcc	Integer	C	C programming language compiler	Compiler
181.mcf	Integer	C	Combinatorial optimization	Min cost flow solved using network simplex algorithm
186.crafty	Integer	C	Game playing: chess	Chess program
197.paser	Integer	C	Word processing	Syntax parser
253.perlbnk	Integer	C	PERL programming language	Perl interpreter
254.gap	Integer	C	Group theory, interpreter	Computing in groups
255.vortex	Integer	C	Object-oriented database	Single user transaction OODBMS
300.twolf	Integer	C	Place and route simulator	Global routing placement for microchips
168.wupwise	Floating-point	Fortran	Physics / quantum chromodynamics	Implementation of lattice gauge theory
172.mgrid	Floating-point	Fortran	Multi-grid solver: 3D potential field	Multigrid solver adapted from NAS
177.mesa	Floating-point	C	3-D graphics library	Open gl
179.art	Floating-point	C	Image recognition / neural	Implementation of organic

			networks	neural networks
200.sixtrack	Floating-point	Fortran	High energy nuclear physics accelerator design	Tracks the particle in a particle accelerator to check beam stability
173.applu	Floating-point	Fortran	Parabolic / elliptic partial differential equations	3D PDE solver
178.galgel	Floating-point	Fortran	Computational fluid dynamics	Numerical analysis of oscillatory instability of convection in low-Prandtl-number fluids
183.equake	Floating-point	C	Seismic wave propagation simulation	Simulation of seismic wave propagation in large basins
191.fma3d	Floating-point	Fortran	Finite-element crash simulation	Mechanical response simulation
301.apsi	Floating-point	Fortran	Meteorology: pollutant distribution	Weather prediction

4.3.3 stream2

The stream [39] benchmark suite is designed to stress the memory subsystem. The second version of the suite, called stream2, measures sustained memory bandwidth at all levels of the memory subsystem. The benchmark suite is comprised of four vector kernels (fill, copy, daxpy, and sum), which are described in [39]. The execution of the suite causes each kernel to be executed with 32 different array sizes; the array is an array of floating-point numbers. The 32 array sizes range from the minimum array size to the maximum; both sizes are input parameters, where the default settings are 30 and 120,000,000, respectively. During execution of the suite, the array size is incremented using Equation (4.4), where j ranges from 1 to 32. For any iteration and vector length, the number of operations performed is determined by the maximum array size, i.e., it is the same as the number performed when the array is of maximum size, i.e., numops_{\max} . Thus, if the given array size is smaller than the maximum size, the number of iterations are increased to make the number of operations performed equal to numops_{\max} . For example, if the maximum array size is 200 and one operation is performed per array element, $\text{numops}_{\max} = 200$, then for an array size of 100, 100 operations are performed during one iteration. In order to perform 200 operations, the number of iterations must be increased to two. Note that an array of the maximum size is much larger than the data store of the simulated L3 cache.

$$\text{Array Size} = 10^{1.477 + (\frac{j-1}{31}) * 6.6028} \quad (4.4)$$

For the simulation study, *stream2* is used to generate two different traces, on-chip cache-intensive, which is deemed to stress the on-chip L1 and L2 caches, and off-chip memory-intensive, which is deemed to stress the off-chip L3 cache and main memory. These traces are called *stream2_L2* and *stream2_L3*, respectively.

Table 4.3: SPEC CPU2006 Benchmarks used in Pilot Simulation Study

Name	Type	Language	Category	Description
400.perlbench	Integer	C	Perl programming language compiler	Perl interpreter
403.gcc	Integer	C	C Programming language compiler	Compiler
429.mcf	Integer	C	Combinatorial optimization	Vehicle scheduling using a network simplex algorithm
458.sjeng	Integer	C	AI chess program	Chess program
433.milc	Floating-point	C	Physics / quantum chromodynamics	A gauge-field generating program for lattice gauge-theory programs with dynamical quarks
436.cactusADM	Floating-point	Fortran/C	Physics	PDE solver for Einstein's equations
447.dealII	Floating-point	C++	Finite element analysis	Targeted at adaptive finite elements and error estimation
470.lbm	Floating-point	Fortran	CFD	Implementation of the Lattice-Boltzmann Method to simulate incompressible fluids in 3D

4.3.4 Imbench

Imbench [40] is a suite of portable benchmarks consisting of applications that test the bandwidth of memory operations, tcp, and pipes. The benchmark suite also has applications that test the latency of context switching, networking, file system operations, signals, and system-call overheads. Our study uses the *bw_mem* benchmark, the bandwidth benchmark, which can stress the memory using memory operations. In this benchmark, the number of bytes used is an input parameter. Using the *bcopy* memory operation to stress memory bandwidth, we generated two different traces using *bw_mem*: on-chip cache-intensive, which stresses the on-chip L1 and L2 caches, and off-chip memory-intensive, which stresses the off-chip L3 cache and main memory. These traces are called *lmbench_L2* and *lmbench_L3*, respectively.

4.4. TRACE GENERATION

The mechanism used to collect the traces of the applications used in this study is discussed in Section 4.4.1. In addition, in this section, we briefly describe how the simulator was configured to capture the required performance data for each trace co-schedule.

4.4.1 Instruction Tracing

The Linux tool ITrace [41] was used to capture partial instruction traces of unmodified benchmark binaries in an in-memory buffer. Although the original toolkit restricted the buffer size and, thus, the trace size, to 200 MB per CPU, we were successful in modifying and recompiling the toolkit to use a maximum buffer size of 1GB. The maximum buffer size was determined experimentally – larger buffer sizes crashed the system. This was because, at the time of the study, the toolkit did not allow tracing beyond the size of main memory.

For the purpose of tracing, for each target application, a single instance of the application was executed on one processor of a partition of an IBM eServer pSeries 590 (p590). The p590 has two 1.65 GHz POWER5 processors and 4GB main memory. The tracing toolkit was configured to capture an instruction trace of the running application only. It was determined experimentally that, for any of the benchmarks of interest, after ten seconds of execution time, the 1GB trace buffer is full and a trace of the maximum size is captured. This execution time interval includes the time required to initialize the tracing daemon, instrument the code, and capture the instructions and related information needed to generate instruction trace records. Thus, ITrace was configured to trace an application for 10 seconds. As shown in Table 4.4, instruction traces between 17 million and 110 million instructions were captured.

For each of the SPEC CPU2000 and SPEC CPU2006 benchmarks tracing was performed during what we believe is a benchmark’s steady state. The benchmark is deemed to be in a steady state when its IPC for a 10-second time interval, which is the execution time interval required for ITrace to capture the maximum size trace, is close to the benchmark’s average IPC. The research of [54] also considers average IPC as a metric to estimate how well traces represent actual program execution. This was accomplished by first creating an IPC profile of each benchmark – the profile provides the IPC of each ten-second execution time interval– and computing the average IPC. The tool pmcount [42], which

permits the performance counter events associated with IPC to be counted for each interval, was used to collect the data needed to create IPC profiles and compute average IPC. To reduce the variability of performance counters that may arise due to interference from other processes, the machine was set up with no user task running on the other processors. Given an IPC profile and an average IPC, a candidate execution time interval was selected for tracing, i.e., one with an IPC that is close to the benchmark's average IPC.

To collect the traces associated with the execution of `stream2_L2` and `stream2_L3`, described in Section 4.3.3, first a memory performance profile was generated that depicts memory performance during each of the 32 iterations (one per array size) that comprise each benchmark's execution time. For each iteration, the `pmcount` tool was used to count performance counter events associated with L2-cache, L3-cache, and memory performance. Next, the memory profiles were analyzed to determine the array sizes that were suitable to represent on-chip cache, off-chip cache, and memory stress. The `stream2_L2` trace (on-chip cache stress) was generated during the processing of the array of size 17,609; this iteration had an L2-cache miss rate of less than 5%. The `stream2_L3` trace (off-chip cache and memory stress) was generated during the processing of the array of size 890,194; for this iteration 95% of memory accesses were satisfied by the off-chip L3 cache or main memory additionally, there was an approximately equal distribution of L3 cache and main memory hits.

To collect the trace associated with the execution of the `lmbench_L2` benchmark, described in Section 4.3.4, a memory performance profile was generated for the execution instance that uses a data size of 2MB, which stresses the on-chip L1 and L2 caches. Similarly, for the `lmbench_L3` benchmark, a memory performance profile was generated for the execution instance that uses a data size of 35MB, which stresses the off-chip L3 cache and main memory. The data sizes of 2MB and 35MB were determined empirically. To generate a memory performance profile, the `pmcount` tool is used to count performance counter events associated with L2-cache, L3-cache, and memory performance. The memory profiles were analyzed to verify that the data structure sizes of 2M and 35MB stress the memory system as predicted. The benchmark requires three parameters: the number of iterations to run

(N), the amount of data to use, and the desired memory operation. The benchmarks were invoked as shown below:

```
bw_mem -N 10000 2m bcopy
bw_mem -N 3000 35m bcopy
```

Table 4.4: Number of Instructions in Captured Traces

Application	Application Type	Number of Instructions Captured
176.gcc	SPEC CPU000 Integer	20,615,714
181.mcf	SPEC CPU000 Integer	20,561,493
186.crafty	SPEC CPU000 Integer	20,505,431
197.parser	SPEC CPU000 Integer	20,073,366
253.perlbmk	SPEC CPU000 Integer	26,432,377
254.gap	SPEC CPU000 Integer	17,791,850
255.vortex	SPEC CPU000 Integer	61,799,297
300.twolf	SPEC CPU000 Integer	79,041,831
168.wupwise	SPEC CPU2000 Floating-point	21,564,740
172.mgrid	SPEC CPU2000 Floating-point	21,515,970
173.applu	SPEC CPU2000 Floating-point	21,914,492
177.mesa	SPEC CPU2000 Floating-point	20,961,262
178.galgel	SPEC CPU2000 Floating-point	20,900,191
179.art	SPEC CPU2000 Floating-point	19,485,389
183.equake	SPEC CPU2000 Floating-point	20,332,402
191.fma3d	SPEC CPU2000 Floating-point	20,748,908
200.sixtrack	SPEC CPU2000 Floating-point	21,925,478
301.apsi	SPEC CPU2000 Floating-point	110,936,443
stream2_L2	Stream, on-chip cache-intensive	36,763,902
stream2_L3	Stream, off-chip memory-intensive	36,961,865
400.perlbench	SPEC CPU2006 Integer	79,788,233
403.gcc	SPEC CPU2006 Integer	79,630,421
429.mcf	SPEC CPU2006 Integer	85,453,973
458.sjeng	SPEC CPU2006 Integer	86,993,610
433.milc	SPEC CPU2006 Floating-point	86,371,781
436.cactusADM	SPEC CPU2006 Floating-point	80,026,520
447.dealII	SPEC CPU2006 Floating-point	86,803,925
470.lbm	SPEC CPU2006 Floating-point	94,229,745
lmbench_L2	lmbench, on-chip cache-intensive	71,875,845
lmbench_L3	lmbench, off-chip memory-intensive	60,261,965

4.4.2 Simulator Configuration

The configuration of simulator parameters, passed as command line arguments, for SMT mode are as follows: (1) `Trace0`: the trace to be simulated on Thread₀; (2) `Trace1`: the trace to be simulated

on Thread₁; (3) **num_inst**: the total number of instructions to simulate, which was set to one billion, which is greater than the combined number of instructions of any two traces; (4) **t0_prio**: the desired priority of Thread₀; and (5) **t1_prio**: the desired priority of Thread₁.

As explained in Section 4.2, the simulation terminates when one trace of the co-schedule finishes. The goal of this study was to determine the throughput of co-scheduled traces. Thus, for a trace co-schedule at a given priority pair, the number of instructions of the unfinished trace completed in SMT mode, which is output by the simulator, is used as a parameter for the single-threaded simulation of the unfinished trace. The single-threaded simulation is configured as follows: (1) **Trace**: the unfinished trace to execute in single-threaded mode; (2) **num_inst**: the total number of instructions to simulate, which was set to one billion, which is greater than the length of any trace used in this pilot study; and (3) **num_inst_to_skip**: the number of instructions of the unfinished trace simulated in SMT mode. Accordingly, the aggregate execution time of a co-schedule at a given priority pair is the simulated SMT execution time plus the simulated single-threaded execution time of that portion of the trace of the thread that was not fully processed. For each studied trace co-schedule, 11 simulations were performed in SMT mode with **t0_prio** and **t1_prio** taking on the values of the 11 different priority pairs listed in Table 4.1. Additionally, for each studied trace co-schedule, 11 simulations were performed in single-threaded mode to execute unprocessed instructions of the unfinished trace.

4.5 EXPERIMENTAL DESIGN

Using pairs of homogeneous applications, 117 homogeneous trace co-schedules were formed, and, using pairs of heterogeneous applications, 146 heterogeneous trace co-schedules were formed. The heterogeneous trace co-schedules were formed using traces from the execution of two applications, each of which is deemed to stress a different processor resource (FPU, FXU, on-chip cache, or off-chip cache and memory). For any heterogeneous trace co-schedule used in the study, the two associated applications came from the following pairs of benchmark suites: (1) SPEC2000int and SPEC2000fp, (2) SPEC2000int and stream2, (3) SPEC2000fp and stream2, (4) SPEC2006int and SPEC2006fp, (5) SPEC2006int and lmbench, and (6) SPEC2006fp and lmbench. Accordingly, a total of 263 trace co-

schedules were studied; Table 4.5 shows the number of trace co-schedules associated with the different pairs of benchmark suites.

Table 4.5: Number of Trace Co-schedules associated with Different Benchmark Suite Pairs

Benchmark Suite Pair	Number of Trace Co-schedules	Co-schedule Group
SPEC2000int and SPEC2000int	36	Homogeneous
SPEC2000fp and SPEC2000fp	55	Homogeneous
stream2 and stream2	3	Homogeneous
SPEC2006int and SPEC2006int	10	Homogeneous
SPEC2006fp and SPEC2006fp	10	Homogeneous
lmbench and lmbench	3	Homogeneous
SPEC2000int and stream2	16	Heterogeneous
SPEC2000fp and stream2	20	Heterogeneous
SPEC2000int and SPEC2000fp	80	Heterogeneous
SPEC2006int and lmbench	7	Heterogeneous
SPEC2006fp and lmbench	7	Heterogeneous
SPEC2006int and SPEC2006fp	16	Heterogeneous

To quantify the effect of different hardware thread priorities on co-schedule execution times, we analyzed the simulation data produced by these 263 co-schedules simulated under the 11 different priority pairs. For each trace co-schedule, we identified the best and worst priority pairs, i.e., those that yield the best and worst $IPC_{\text{aggregate}}$, and recorded the $IPC_{\text{aggregate}}$ attained by using the default priority pair, i.e., (4,4). The $IPC_{\text{aggregate}}$ metric is shown in Equation 1.1

4.6 RESULTS

To improve readability, in this section, in particular, in Figure 4.2 and Table 4.6, the following abbreviations for benchmark suites are used: for SPEC2000Int, *Int2000*; for SPEC2000fp, *FP2000*; for SPEC2006int, *Int2006*; and for SPEC2006fp, *FP2006*.

First, we determine the priority pair that yields best core throughput, i.e., $IPC_{\text{aggregate}}$, for each trace co-schedule by simulating the execution of each co-schedule using the 11 different priority pairs. These results are depicted in Figure 4.1, which shows the distribution of the 263 trace co-schedules

across the best priority pairs. In Figure 4.1, the X-axis represents the 11 priority pairs and the Y-axis represents the percentages of the 263 trace co-schedules that achieved best core throughput at these priority pairs. As shown in this figure, *the best priority pair depends on the characteristics of the trace co-schedule; each of the 11 priority pairs yields best $IPC_{aggregate}$ for some set of trace co-schedules*. The default priority pair (4, 4) is the best priority pair for only 18% of the trace co-schedules, hence, our study shows that *the default priority pair does not always yield best core throughput and to attain best core throughput for a set of co-schedules, the full range of priority pairs should be considered*.

Next, we determine if the default priorities yield best throughput for a majority of the co-schedules associated with any of the 12 benchmark suite pairs, which are shown in the first column of Table 4.5. To determine this, Figure 4.2 depicts the distribution of the trace co-schedule set of each of the 12 benchmark suite pairs across the 11 priority pairs. In this figure, the X-axis represents the 11 priority pairs and the Y-axis represents the percentages of the trace co-schedule sets that attain best throughput when executed with each priority pair. The legend on the right of the figure shows the 12 benchmark suite pairs. For example, for the benchmark suite pair (Int2006, lmbench), priority pair (2, 7) is best for 43% of its trace co-schedule set. As shown in Figure 4.2, *the default priority pair is not the best for the entire trace co-schedule set of any one of the 12 benchmark suite pairs. In addition, in only two cases is it best for the majority of a trace co-schedule set, i.e., 66% of (stream2, stream2) and 50% of (Int2006, Int2006), two of the six homogeneous trace co-schedule sets*. The average difference between the core $IPC_{aggregate}$ attained with the default priority pair and that attained with the best priority pair is 3.54%, with a standard deviation of 2.09%. The best and worst differences between the core $IPC_{aggregate}$ attained by executing with the default priority pair as compared to executing with the best priority pairs are 0.63% and 6.69%, respectively.

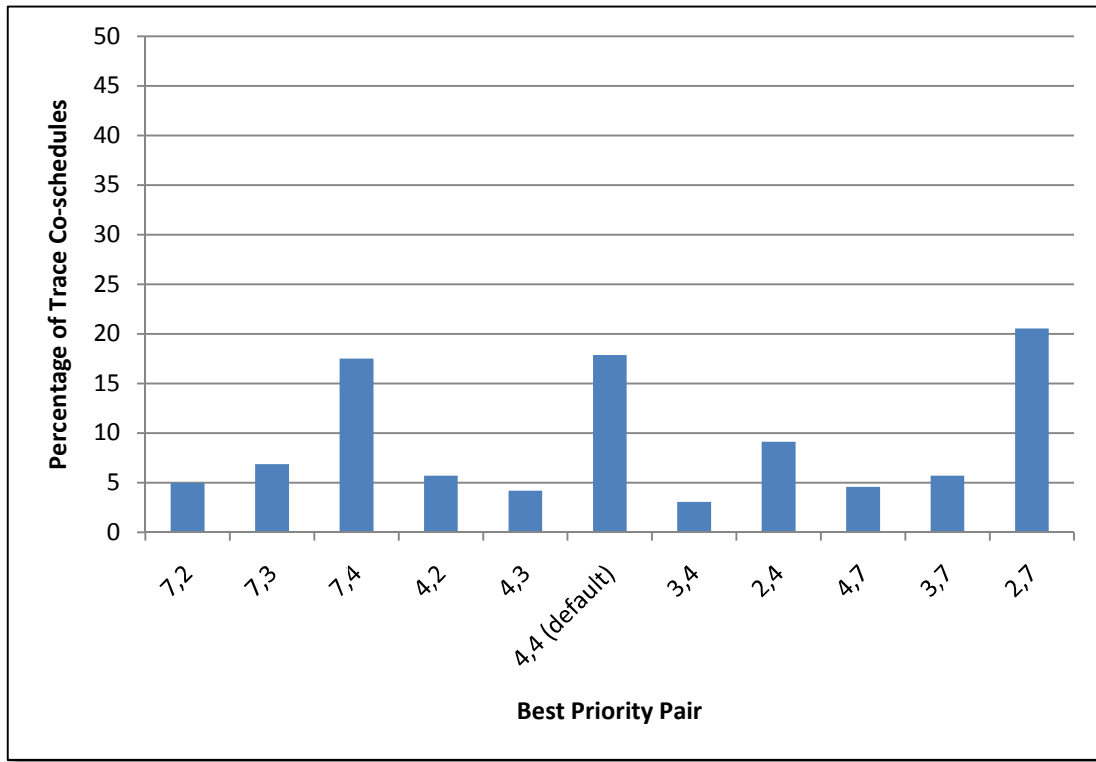


Figure 4.1: Distribution of the 263 Trace Co-schedules w.r.t. Best Priority Pair

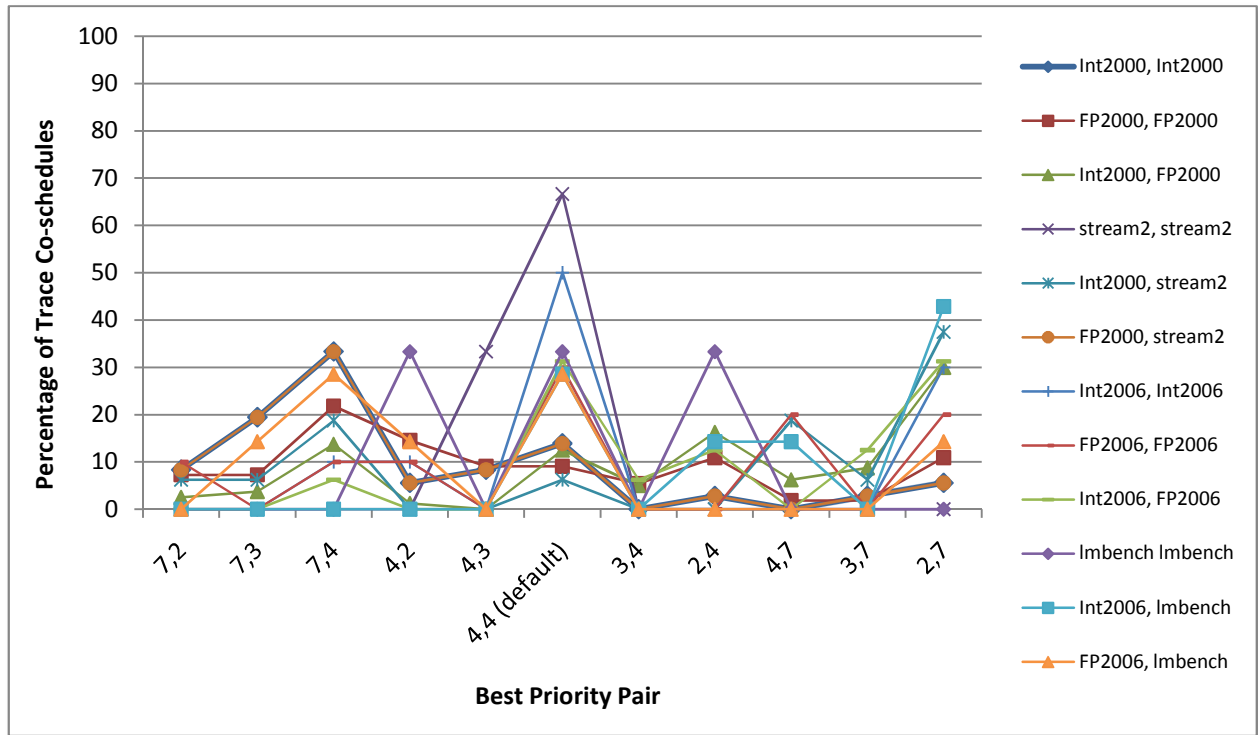


Figure 4.2: Distribution of 263 Trace Co-schedules formed from Int2000, FP2000, Int2006, FP2006, stream2, and Imbench w.r.t. Best Priority Pair

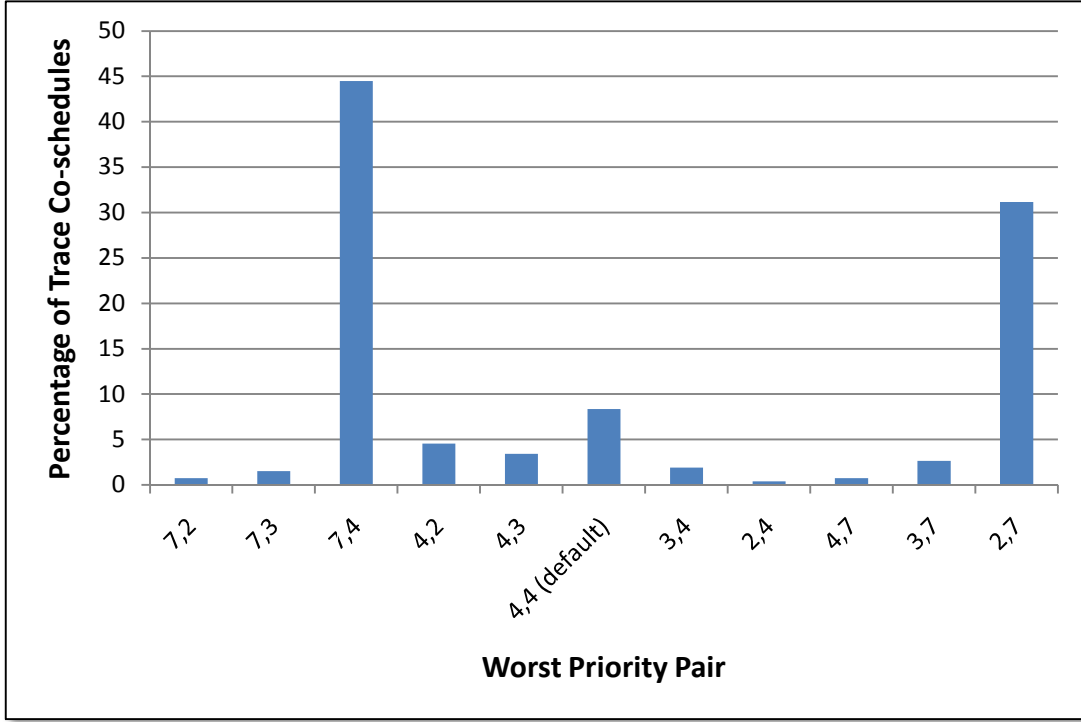


Figure 4.3: Distribution of 263 Trace Co-schedules w.r.t. Worst Priority Pair

Next we determine if any particular priority pairs always result in poor performance or lower performance than that achieved with the default priority pair. The histogram depicted in Figure 4.3 allows us to determine this by plotting the percentage of the total number of trace co-schedules studied (Y-axis) that achieve worst throughput with each of the 11 priority pairs (X-axis). As illustrated in the figure, *worst case performance was achieved at all 11 priority pairs*. The average difference between the best and worst core $IPC_{aggregate}$ is 13.26%, with a standard deviation of 3.61%. The minimum and maximum differences between the core $IPC_{aggregate}$ attained with the worst and best priority pairs are 7.92% and 19.29%, respectively. Note that the default priority pair yields worst throughput for 8.4% of the trace co-schedules.

We analyze the difference between the $IPC_{aggregate}$ attained with the default and best priorities to assess the potential benefit of using non-default priorities. In addition, to assess if the wrong choice of priorities can lead to significant performance degradation we also compare the $IPC_{aggregate}$ attained with the best and worst priorities. Table 4.6 compares the impact of the best, worst, and default priority pairs on core $IPC_{aggregate}$. Each row of the table is associated with a benchmark suite pair. Columns two

through four give the average, minimum, and maximum percentage difference between the default core $IPC_{\text{aggregate}}$, i.e., the core $IPC_{\text{aggregate}}$ achieved with the default priority pair, and the best $IPC_{\text{aggregate}}$, i.e., the $IPC_{\text{aggregate}}$ achieved with the best priority pair. Columns five through seven give the average, minimum, and maximum percentage difference between the worst $IPC_{\text{aggregate}}$, i.e., the $IPC_{\text{aggregate}}$ achieved with the worst priority pair, and the best $IPC_{\text{aggregate}}$.

Table 4.6: Comparisons of Best, Worst, and Default $IPC_{\text{aggregate}}$

Benchmark Suite Pair	$((IPC_{\text{Best}} - IPC_{\text{Default}}) / IPC_{\text{Best}}) * 100$ Average	$((IPC_{\text{Best}} - IPC_{\text{Default}}) / IPC_{\text{Best}}) * 100$ Minimum	$((IPC_{\text{Best}} - IPC_{\text{Default}}) / IPC_{\text{Best}}) * 100$ Maximum	$((IPC_{\text{Best}} - IPC_{\text{Worst}}) / IPC_{\text{Best}}) * 100$ Average	$((IPC_{\text{Best}} - IPC_{\text{Worst}}) / IPC_{\text{Best}}) * 100$ Minimum	$((IPC_{\text{Best}} - IPC_{\text{Worst}}) / IPC_{\text{Best}}) * 100$ Maximum
Int2000, Int2000	5.01%	0.00%	25.52%	11.30%	0.42%	35.87%
FP2000, FP2000	5.09%	0.00%	19.25%	16.68%	1.47%	39.69%
Int2000, FP2000	5.12%	0.00%	19.13%	17.13%	0.59%	45.04%
stream2, stream2	0.63%	0.00%	1.88%	7.92%	3.21%	10.87%
Int2000, stream2	1.47%	0.00%	9.23%	12.01%	2.88%	26.50%
FP2000, stream2	1.22%	0.00%	5.95%	12.22%	1.41%	39.14%
Int2006, Int2006	1.49%	0.00%	4.62%	10.45%	0.66%	30.17%
FP2006, FP2006	6.69%	0.00%	21.26%	19.29%	1.64%	51.09%
Int2006, FP2006	3.48%	0.00%	10.71%	17.69%	5.59%	41.84%
lmbench, lmbench	3.71%	0.00%	6.13%	12.52%	6.13%	21.47%
Int2006, lmbench	2.36%	0.00%	7.07%	9.16%	3.36%	19.40%
FP2006, lmbench	6.24%	0.00%	16.46%	12.78%	4.09%	24.69%

As shown in the second column, the $IPC_{\text{aggregate}}$ achieved with the best priority pair is, on average, 0.63% to 6.69% better than that achieved with the default priority pair. As shown in column five, the difference between the IPCs achieved with the best and the worst priority pairs is, on average, between 7.92% and 19.29%.

4.7 CONCLUSIONS

This study shows that:

- the default setting of the hardware thread priorities on the POWER5 SMT processor yields best throughput for only 18% of the trace co-schedules studied;
- each of the priority pairs achieves best throughput for some subset of the trace co-schedules;

- utilizing the best priority pair can improve throughput, over that achieved using the default priority pair, by as much as 6.69%; and
- worst-case throughput differs from the best-case throughput by as much as 19.29%.

Thus, the choice of priority pair has an impact on SMT $IPC_{\text{aggregate}}$. The best choice can improve $IPC_{\text{aggregate}}$, as compared with that attained by using the default priority settings, while the worst choice can decrease $IPC_{\text{aggregate}}$, as compared with that attained using the default priority settings.

This study indicates that the intelligent setting of hardware thread priorities can enhance core throughput. Note, however, that this study makes assumptions w.r.t. to the resource utilization of the trace co-schedules used in the study. As a result, the effectiveness of a methodology that assigns hardware thread priorities to a co-schedule depending upon the nature of the applications that are to be executed concurrently on an SMT processor is questionable. The traces are relatively short, as compared to the total execution times of the applications. Thus, it is not clear if the captured traces are representative of the assumed intensity of the floating-point, integer, or memory activity of the applications from which they were generated. The study does not (1) show how the compute-intensive applications differ from the memory-intensive ones, (2) characterize the homogeneous applications in terms of the processor resources stressed, or (3) differentiate between the processor resources stressed by pairs of heterogeneous applications. Instead, assumptions are made based on the literature. Nonetheless, the studied co-schedules represent the co-execution of different pairs of applications with different behaviors. In addition, the goal of this study is to understand the impact of the setting of the hardware thread priorities of the POWER5 on SMT core throughput. Hence, we surmise that the results of this study indicate that there is promise in providing a methodology that intelligently assigns hardware thread priorities to co-schedules that will execute concurrently on an SMT processor. Accordingly, this dissertation develops such a methodology.

5 Best Priority Pair Prediction Methodology

The best priority pair prediction methodology described in this chapter is used in Chapter 6 to predict the best priority pair for given co-schedules of two sequential applications and a given SMT processor architecture that supports software-controlled hardware thread priorities. The methodology also can be used to predict best priority pairs, i.e., priority settings of the two hardware threads of an SMT processor core that produce best throughput, for co-schedules of segments of applications. To accomplish this we make use of information that characterizes an application's utilization, when executed in single-threaded mode, of processor core resources that are shareable by hardware threads running concurrently in SMT mode. This utilization information, which is captured periodically, can provide insights into the availability of the shared resources for another application's simultaneous use in SMT mode. This characterization of an application's resource utilization is called its *Shareable Resource Signature* (also referred to as *signature*), which is described in Section 5.1. Given a target SMT processor architecture that supports software-controlled hardware thread priorities, $\text{processor}_{\text{target}}$, our best priority prediction methodology has three distinct phases. Phase 1, called Shareable Resource Signature Generation, described in Section 5.2, is used to create the archetypical application signature for the target SMT processor. Phase 2, called Prediction Framework Development, described in Section 5.3, is used to construct the prediction framework for the target architecture. Finally, Phase 3, called Prediction Validation, described in Section 5.4, is used to assess the accuracy of the framework's best priority pair predictions. The methodology described is iterative in nature – if the desired prediction accuracy is not achieved, then some or all steps of Phases 1 and 2 of the methodology may have to be repeated. The composition of each step of the methodology has improved as a result of the experience we have gained from implementing the methodology for the IBM POWER5 processor, which is described in Chapter 6. In addition, it is likely that it will improve further as we explore different implementations for the POWER5 processor and implement the methodology for different processor architectures.

5.1 INTUITION

Co-scheduled applications executing on the hardware threads of an SMT processor core compete for shared core resources and, as a result, may contend for these resources. We hypothesize that information that accurately characterizes an application's degree of utilization of the set of critical shareable resources of a core, when executed in single-threaded mode, can provide hints about the degree of availability of those resources for another application's concurrent use, when executed in SMT mode. A core's set of critical shareable resources consists of those that have a significant impact on core throughput. Given a signature pair that characterizes either a co-schedule of program segments of two applications or the majority of the execution times of two applications, the best priority pair prediction methodology, presented in the next three sections, can be used to predict the best priority pair for the co-schedule.

5.2 PHASE 1: SHAREABLE RESOURCE SIGNATURE GENERATION

The first phase of our best priority pair prediction methodology is Shareable Resource Signature Generation. The input to this phase is the target SMT processor architecture that supports software-controlled hardware thread priorities, $\text{processor}_{\text{target}}$. The first step of this phase is to identify the *signature-generating applications*, a set of applications that are representative of the classes of applications that are targeted to run on this processor. Given this set of applications, Phase 1 creates the archetypical application signature for $\text{processor}_{\text{target}}$. Before describing a five-step process for accomplishing this, we provide definitions of terminology used in the remainder of this dissertation, i.e., the *Shareable Resource Signature*, which we refer to as *signature*, the *interval length*, which determines the periodicity of signature collection, *signature set*, *time-ordered signature set*, *interval set*, *signature phase*, and *application with a dominating signature*.

5.2.1 Terminology

An application's Shareable Resource Signature is an N -tuple of ordered pairs, $(\langle R_1, U_{R_1} \rangle, \langle R_2, U_{R_2} \rangle, \dots, \langle R_N, U_{R_N} \rangle)$, where

- N is the number of critical shareable core resources identified for processor_{target};
- R_i is an alphabet that represents a critical shareable core resource i , for $i = 1, \dots, N$;
- U_{R_i} is a number $\in \{1, \dots, M_{R_i}\}$ that represents the level of utilization of critical shareable core resource R_i during an interval of the given application's execution time (execution time interval), where M_{R_i} is the number of utilization levels defined for resource R_i .

The length of the execution time interval, referred to as the *interval length*, determines the frequency with which a signature is captured. The smaller the interval length, the finer is the granularity of signature capture and the greater is the accuracy of the resource-usage characterization.

For example, as discussed in Chapter 6, for the IBM POWER5 processor we identify four critical shareable core resources, i.e., the floating-point unit, integer unit, L2 cache, and TLB. We represent them with the symbols F, I, C, and T. If during an execution time interval of length equal to the interval length an application utilizes the four critical resources F, I, C, and T at 28%, 12%, 25%, and 15% of their capacities, respectively, then the signature of the application for the execution time interval is $(\langle F, 28 \rangle, \langle I, 12 \rangle, \langle C, 25 \rangle, \langle T, 15 \rangle)$. In this dissertation, we classify the resource utilization values, 0% to 100%, into an appropriate number of levels. For our POWER5 implementation, we use 10 levels for each of the four resources, which are labeled by integers 1 to 10, where 1 represents a utilization of 0% to 10%, 2 represents a utilization of 11% to 20%, 3 represents a utilization of 21% to 30%, ..., 10 represents a utilization of 91% to 100%. Thus, the signature of the example application for the considered execution time interval is $(\langle F, 3 \rangle, \langle I, 2 \rangle, \langle C, 3 \rangle, \langle T, 2 \rangle)$. Further, we simplify the notation of a signature by denoting this signature by F3I2C3T2. Although we use 10 levels of utilization for each resource in our POWER5 implementation, it is possible that to attain better prediction accuracy or to implement the methodology for another processor, the appropriate number of utilization levels may be different for each of the critical shareable core resources.

The execution time of an application can be considered to be composed of a sequence of consecutive execution time intervals of length equal to the interval length. Let the number of such execution time intervals for the execution time, E , of the example application be T , i.e., E is composed of a time-ordered sequence of execution time intervals, $I_1, I_2, I_3, \dots, I_T$. Let the chronological sequence of signatures corresponding to these intervals be $\langle S_1, S_2, \dots, S_T \rangle$, which is defined as the application's *time-ordered signature set*. The set of signatures $\{S_1, S_2, \dots, S_T\}$ is the application's *signature set* and the set of execution time intervals $\{I_1, I_2, I_3, \dots, I_T\}$ is the application's *interval set*.

If for consecutive execution time intervals $I_i, I_{i+1}, I_{i+2}, \dots, I_j$, where $(1 \leq i \leq j \leq T)$, the corresponding signatures, S_i, S_{i+1}, \dots, S_j , are such that $S_i = S_{i+1} = S_{i+2} = \dots = S_j$, then during its execution time the application is said to have a *signature phase* of S_i that corresponds to the execution time intervals I_i to I_j .

An *application with a dominating signature*, S_i , is an application with a signature set such that the percentage of application execution time that is characterized by signature S_i is greater than or equal to a user-defined threshold percentage of application execution time; determination of this threshold is explained in Section 5.4. For example, in our implementation for the IBM POWER5 the threshold was determined to be 95%. Hence, in our implementation for the IBM POWER5 an application has a dominating signature S_i if 95% or more of the application's execution time is characterized by S_i . Best priority pair prediction for co-schedules of applications with dominating signatures can assume that each of the co-scheduled applications has one signature phase during its entire execution time. For such a co-schedule the predicted best priority pair can be statically assigned at the beginning of the execution of the co-schedule and held constant for the entire execution times of the two applications.

5.2.2 Five-step Process

To generate a Shareable Resource Signature for processor_{target} the following five steps we perform the following five steps. These steps, except when noted, are carried out in Chapter 6 for our POWER5 processor implementation of the methodology.

Step 1:

Identify the set of signature-generating applications. Ideally, these applications should represent the characteristics of applications that will be run on processor_{target}. For example, if compute-intensive applications are intended to run on this processor, then a standard benchmark suite such as SPEC CPU might be used as the set of signature-generating applications.

To identify the set of signature-generating applications:

- a) Determine the classes of applications that are targeted to run on this processor.
- b) Determine the computational characteristics of the classes of applications identified in step a. This can be accomplished by profiling and/or reading published literature.
- c) Select benchmarks that represent the characteristics of the classes of applications identified in step b. Again, like in step b, the characteristics of the selected benchmarks can be ascertained by profiling and/or reading published literature. This set of benchmarks is the set of signature-generating applications.

Step 2:

Identify the set of N critical shareable core resources R_i (also referred to as *critical resources*), i.e., the shareable core resources that have significant impact on the core throughput of the signature-generating applications. A resource is considered to be in this set if contention for this resource by two signature-generating applications that comprise a co-schedule executed in SMT mode results in substantial throughput loss, a metric that is defined by the user.

To identify the set of critical shareable core resources:

- a) Identify the set of *shareable core resources*.
- b) Establish which of the shareable core resources can be measured in terms of utilization – these are called the *monitorable shareable core resources*.

- c) Define, for each monitorable shareable core resource, a metric to measure its utilization by an application executed in single-threaded mode. Note that this metric is resource dependent. For example, for a functional unit, the associated metric might quantify the number of cycles during which it is busy, while for a cache, the associated metric may be the number of accesses.
- d) Validate the set of metrics defined in the previous step. For each metric, write microbenchmarks that stress the resource at pre-defined utilization levels, which we call *resource-stress microbenchmarks*, and validate the accuracy of the estimates provided by the associated metric. The design of the microbenchmarks depends on the given processor_{target} and the monitorable shareable core resources. Hence, we give a detailed explanation of the design of the resource-stress microbenchmarks for the IBM POWER5 in the next chapter in Section 6.2.2.
- e) Identify the set of *substantially used monitorable shareable core resources*. As explained at the beginning of Step 2, the extent of throughput loss that is considered to be substantial is defined by the user. As explained in Chapter 6, in our POWER5 implementation we were targeting throughput improvements greater than or equal to 5%, thus, we considered a 5% loss in throughput as substantial. Hence, the goal is to identify the resources R_i for which contention in SMT mode may result in a 5% core throughput loss. The set of substantially used monitorable shareable core resources can be identified as follows:
 - i) Generate profiles of the single-threaded execution of the signature-generating applications in terms of their utilization of the monitorable shareable core resources.
 - ii) Define the threshold of throughput loss that is considered substantial. As explained earlier, in our POWER5 implementation, we were targeting throughput improvements greater than or equal to 5% and, thus, we considered a 5% core throughput loss to be substantial.

- iii) To determine the threshold of substantial usage, determine the contention for resources that leads to substantial throughput loss which was established in the previous step. However, in our implementation on the IBM POWER5 processor we did not use this methodology and, instead, the threshold of substantial usage was set to the value established as substantial throughput loss in the previous step. Of course, usage does not necessarily translate to throughput loss, for example, the TLB may be used 5% of the time but latencies associated with the resolution of TLB misses may be high and could result in more than a 5% throughput loss. Although not done in this dissertation, a better way to determine substantial resource usage is to consider the latencies associated with resource contention and events such as cache and TLB misses.
- iv) Identify resources that are substantially used by the signature-generating applications. A resource is included in this set if the average utilization of the resource by the set of signature-generating applications is greater than or equal to the threshold value that indicates substantial usage.
- f) Refer to the published literature to identify monitorable shareable core resources that result in substantial throughput loss. If required, modify the set of resources identified in the previous step accordingly to form the set of *critical shareable core resources* (also referred to as *critical resources*).

Step 3:

Determine the interval length and, thus, the frequency for recording an application's signature. For example, if the interval length is one second, then a signature is recorded for each second of the application's execution time. The smaller the interval length, the more accurate is the characterization of the application's resource utilization, but the more pervasive is the associated monitoring, which can perturb the application's execution time. For example, consider the case where an application's L1

instruction cache misses are monitored at each cycle of its execution. The monitoring tool will, for every one cycle of execution, interrupt the application, execute on the processor and output the L1 instruction cache miss count. Assume that during each of the monitoring tool's executions it populates the entire L1 instruction cache and evicts all the instructions of the monitored application. In this case, each time the application executes it will generate L1 instruction cache misses to recoup its working set. As a result, the application's execution time will increase, as compared to the time it would take without performance monitoring. To determine the interval length the rule of thumb that is used in this research is to decrease the monitoring frequency until there is less than or equal to 1% perturbation of the execution time of each application in the set of signature-generating applications.

The interval length can be identified as follows:

- a) Determine the base execution time of signature-generating applications, i.e., the execution time without monitoring; this execution time is referred as $Total_Cyc_{Base}$.
- b) Determine the maximum interval length. The interval length cannot be larger than the execution time of the shortest running signature-generating application. Hence, the execution time of the shortest running signature-generating application is used as the maximum interval length.
- c) Determine the minimum interval length. The monitoring tool determines the smallest granularity at which signature-generating applications can be monitored. The perturbation of application execution time increases with finer granularities of monitoring. The percentage of perturbation depends on the tool being used and, thus, the minimum interval length is dependent on the monitoring tool.
- d) Construct an empty set of potential interval lengths, L . Add the maximum and minimum interval lengths to this set.
- e) Follow the steps below to add interval lengths to L .
 - i) Set the value of Q to the maximum interval length.
 - ii) Calculate $J = Q / 10$.
 - iii) If J is less than or equal to the minimum interval length, then go to step f.

- iv) Add the interval length J to set L . Go to step ii.
- f) For each interval length k in set L do the following:
 - i) Set the monitoring frequency equal to the interval length k .
 - ii) Determine the resultant execution times of the signature-generating applications with monitoring enabled; this execution time is referred as $Total_Cyc_k$.
 - iii) Calculate the resultant monitoring perturbation as follows: $(|Total_Cyc_{Base} - Total_Cyc_k|) / Total_Cyc_{Base} * 100$.
- g) Determine the subset of evaluated interval lengths in set L that result in less than 1% perturbation; these are the set of candidate interval lengths K .
- h) Identify the smallest interval length in set K ; this is I , the interval length that determines the periodicity at which signatures are captured.

As shown in step e, we divide the maximum interval length, i.e., the execution time of the shortest running signature-generating application by 10 to calculate interval lengths that are candidates for I . As a result, the number of intervals that are evaluated are limited and this may lead to the selection of a suboptimal value of I . Modification of step e to increase the number of candidate intervals can address this limitation.

Step 4:

For each resource R_i identified in Step 2, define its utilization levels, i.e., $U_{Ri} \in \{1, \dots, M_{Ri}M\}$. The goal of this step is to determine the number of utilization levels for each resource, i.e., $M_{Ri}M$. Optimally $M_{Ri} \forall R_i$ should characterize applications in such a way that only applications that have significantly different resource requirements are characterized with different signatures. For example, consider two applications, Application_B and Application_C, each co-scheduled with another Application_A. If Application_B and Application_C do not have significantly different resource requirements then they should have the same best priority pair when each are co-scheduled with Application_A. However if

Application_B and Application_C have significantly different resource requirements then they should have different best priority pairs when they are co-scheduled with Application_A. Hence, optimally, signature-generating applications with significantly different resource requirements should be characterized by different signatures and vice versa.

One way to determine the number of utilization levels is to use the following algorithm:

- a) Define the number of utilization levels M_{R_i} for each resource R_i .
- b) Using definitions of utilization levels of step a, determine the signatures of the signature-generating applications.
- c) Run co-schedules of all application-pairs of signature-generating applications at all available priority pairs and determine the best priority pairs.
- d) Determine if two applications from the set of signature-generating applications with the same signature have different best priority pairs when co-scheduled with the same application. If such an application-pair is found, then increase M_{R_i} for one or more resources and go to step b.

The above algorithm may be intractable to implement if there is a large set of signature-generating applications. For example, in our POWER5 implementation there are 149 application-input-data combinations and a total of 22,201 possible co-schedules. For a given priority pair, a co-schedule can execute between two and 3600 seconds. Since we evaluate 11 priority pairs, it may take between 488,422 and 879,159,600 seconds to execute the 22,201 co-schedules. Hence, instead of the above algorithm, we use the algorithm outlined below to define the number of utilization levels M_{R_i} for each resource R_i :

- a) For each signature-generating application, record, at the interval length established in Step 3, the critical shareable core resource usage of the single-threaded executions of the signature-generating applications.
- b) Analyze the resource utilization histograms of the N critical resources and define resource utilizations levels by following the steps below:

- i) Identify the minimum and maximum resource usage for each of the N critical shareable core resources.
- ii) Define the same number of utilization levels $M_{R_i} \forall R_i$ such that it includes the minimum and maximum resource usage for each of the N critical resources. In our POWER5 implementation, the minimum utilization was 0% and maximum was 85% among all the four resources. Instead of covering only range between 0% and 85%, we included the entire possible range, i.e., between 0% and 100%.
- iii) Determine if this definition sufficiently differentiates signature-generating applications belonging to different application classes. In our POWER5 implementation we determined the efficacy of this definition by comparing the distribution of the percentage of execution time intervals of an application class characterized by a signature. If the percentage of execution time intervals of two or more application classes are characterized by the same set of signatures, then refine the number of utilization levels and repeat step iii.

For example, we first implemented the above algorithm with $M_{R_i} = 2$, i.e., two utilization levels per resource R_i , i.e., $0\% \leq U_{R_i} \leq 50\%$ and $51\% \leq U_{R_i} \leq 100\%$. Using this definition we found that 99% of execution time intervals had the same signature. However, from previous experiments we knew that many of the application-pairs have different best priority pairs. Thus, we next used ten levels per resource, $M_{R_i}=10$, such that level 1 is associated with utilization between 0% and 10%, 2 with utilization between 11% and 20%, etc. Using 10 levels of utilization for each resource R_i the throughput attained with our best priority predictions were on average 2.63% below those obtained using the actual best priority pairs.

As a result of using the same number of levels M_{R_i} for each resource R_i , for resources that do not span the entire range of utilization, applications will not have all the levels for each resource defined by M_{R_i} . For example, in our POWER5 implementation, which uses 10 levels of utilization per resource R_i , it was observed that the TLB utilization of the signature-generating applications spans only four

utilization levels. Furthermore, for a given utilization level, the percentage of execution time intervals that use each of the N resources at the given level may be different. For example, in our POWER5 implementation, 99% of the execution time intervals had level-one TLB utilization, whereas only 45% of the execution time intervals had level-one FXU utilization. Thus, the distribution of execution time intervals across the different levels for each resource may not be uniform. To make the distribution uniform, a possible solution is to define for each R_i an M_{R_i} that uniformly divides utilization between the minimum and maximum utilization, e.g., in the case of the POWER5's TLB define the number of utilization levels to be 10 such that 10 levels equally cover utilization from 0% to 40%. While this may distribute execution time intervals uniformly for each resource R_i it may also increase the number of signatures that characterize applications. An explosion in the number of observed signatures may make it intractable to implement the methodology. Moreover, in our implementation only 45 out of the 10,000 possible signatures are sufficient to characterize the set of signature-generating applications. Hence, it is not clear if increasing the number of signatures will actually increase the accuracy of characterization and, thus, the best priority pair prediction. We believe that a better solution to define utilization levels may become evident to us with further use of the best priority pair prediction methodology. Since the implementation presented in Chapter 6 is a proof of concept we did not try other definitions of utilization levels to address these limitations.

Step 5:

Define the Shareable Resource Signature, i.e., the archetypical signature, for processor_{target} and build a database of the time-ordered signature sets of the signature-generating applications; these are the representative signatures used in the next phase of the methodology.

This step can be carried out as follows:

- a) Establish the Shareable Resource Signature for processor_{target} and simplify the notation from $(\langle R_1, U_{R_1} U \rangle, \langle R_2, U_{R_2} \rangle, \dots, \langle R_N, U_{R_N} \rangle)$ to $R_1 U_{R_1} U R_2 U_{R_2} U \dots R_N U_{R_N} U$. This notation can be further simplified by representing each critical resource R_i with a distinct alphabet.

- b) Create the time-ordered signature set for each signature-generating application using the signature representation established in step a.
- c) Store the time-ordered signature sets of all the signature-generating in the signature database.

5.3 PHASE 2: PREDICTION FRAMEWORK DEVELOPMENT

The second phase of our best priority pair prediction methodology is Prediction Framework Development, during which the prediction framework for $\text{processor}_{\text{target}}$ is constructed. The input to this phase is (1) the time-ordered signature sets of the signature-generating applications, which are stored in the signature database, and (2) the resource-stress microbenchmarks created in Phase 1. Given a signature pair that characterizes either a co-schedule of signature phases of two applications or the majority of the execution times of two applications, the framework can be used to predict best priority pairs for the co-schedule. The development of the framework, illustrated in Figure 5.1, has as its first step the creation of what are called signature microbenchmarks, which are built using the resource-stress microbenchmarks created in Phase 1. During the second step a prediction table is built that is used to look up the best priority pair for a given co-schedule; the two steps are described below.

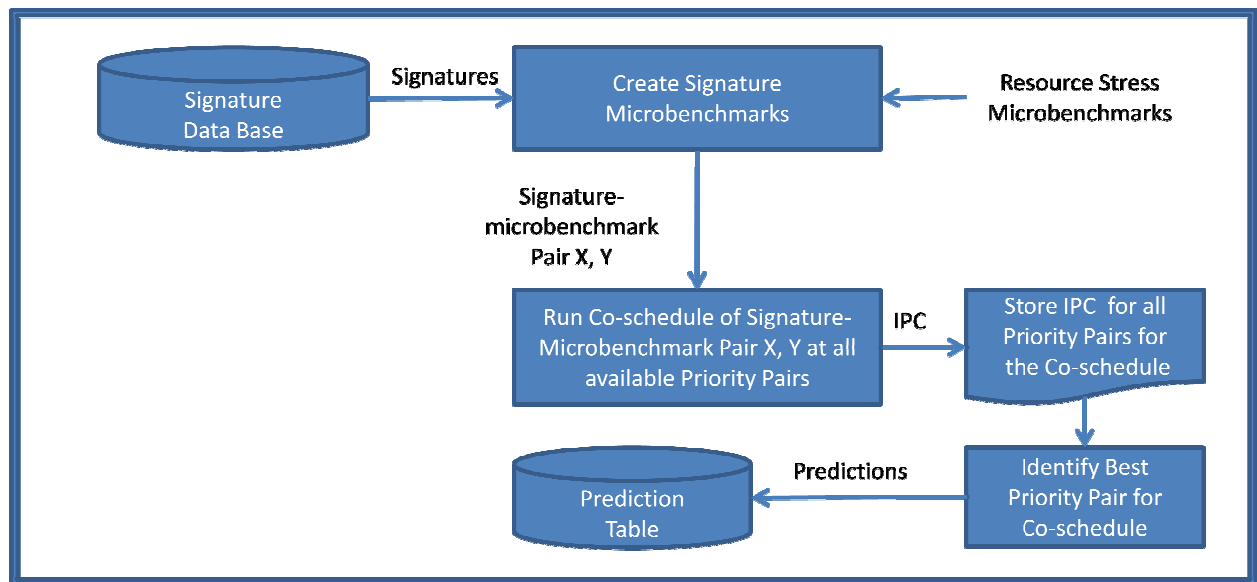


Figure 5.1: Prediction via Signature Microbenchmarks

To construct the prediction framework for processor_{target} the following two steps are followed:

Step 1:

For every distinct signature in the signature database created in Phase 1, create a signature microbenchmark that utilizes the set of critical resources at the utilization levels defined by the signature. These microbenchmarks are built by using the resource-stress microbenchmarks from Phase 1 as templates.

A signature microbenchmark for a specific signature can be created as follows. First combine the N resource-stress microbenchmarks created in Phase 1 – these were created to validate the metrics used to estimate resource utilization – into one microbenchmark. Next, fine-tune this microbenchmark to achieve, for every execution time interval, the defined utilization of the N critical resources. The design of these microbenchmarks depends on the given processor_{target} and the critical resources. Hence, we give a detailed explanation of their design for the IBM POWER5 in the next chapter in Section 6.3.1.

Step 2:

Using the signature microbenchmarks a table is built to predict the best priority pair for every pair of signatures. If prediction is required for a co-schedule comprising an application with a signature that is not stored in the prediction table, then the default priority is output. In the Future Work section of Chapter 7 we discuss a method that may be able to predict priority settings for applications of this kind.

Creation of the prediction table can be accomplished as follows:

- a) Form co-schedules of all pairs of signature microbenchmarks.
- b) Execute all co-schedules under every possible unique priority pair, collecting $IPC_{aggregate}$ for each co-schedule.
- c) For each co-schedule record the best priority pair and store it in the prediction table.

A prediction table is used to predict the best priority pair for a given co-schedule of signature phases of two applications or the majority of the execution times of two applications as follows:

1. For each component of the given co-schedule determine its phase or dominating signature. (Again, the signature is captured while executing the co-schedule component in single-threaded mode).
2. Using the signature pair (X, Y) look up in the prediction table the predicted best priority pair.
3. If the signature pair is in the table, use the predicted best priority pair for the co-schedule; else, use the equal (default) priority pair.

5.4 PHASE 3: PREDICTION VALIDATION

The third and final phase of our best priority pair prediction methodology is the Prediction Validation phase, which assesses the accuracy of the predictions made using the framework constructed in Phase 2. The input to this phase is the prediction table generated in Phase 2. Given this input, a subset of the signature-generating applications, i.e., *target applications*, are chosen to evaluate the accuracy of the best priority pair prediction methodology. Note that the signature microbenchmarks are used to populate the prediction table, whereas the signature-generating applications are candidates for validation.

To validate and assess the accuracy of the best priority prediction methodology, the following three steps can be followed:

Step 1:

Define a threshold percentage of execution time that identifies a dominating signature for an application. As defined in Section 5.2, if the signature represents at least the threshold percentage of the application's execution time, then it is said to be a dominating signature. For a co-schedule comprising two applications with dominating signatures, the predicted priority pair can be statically set at the beginning of the co-schedule's execution and remain unchanged for the entire execution time. Hence, in this case, the implementation of mechanisms to accomplish this and the validation of the accuracy of the

prediction methodology are relatively simple. Note, however, that our predictions also can be used for applications without dominating signatures. For such applications, a mechanism must be developed that is able to identify changes in co-scheduled signature phases and dynamically adapt priorities. Such a mechanism will be relatively more complicated to implement. We discuss this topic in more depth in the future work section of Chapter 7.

The threshold for dominating signatures can be identified as follows:

- a) Set the threshold percentage of application execution time for a dominating signature to 75%.
- b) Using this threshold, identify target applications with a dominating signature.
- c) Form co-schedules using a subset of applications identified in step b.
- d) Identify the accuracy of the best priority pair prediction for the co-schedules formed in step c as follows:
 - [1].Execute the co-schedule using all possible distinct priority pairs and record the $IPC_{aggregate}$.
 - [2].Identify the best priority pair and record its $IPC_{aggregate}$ (BT).
 - [3].Look up the predicted best priority pair in the prediction table and record its $IPC_{aggregate}$ (PT).
 - [4].Compute the accuracy of the prediction as follows: $((PT - BT) / BT) * 100$.
- e) If the desired prediction accuracy is not reached, increase the threshold for dominating signature by 10% and go to step b; else the threshold percentage for a dominating signature has been defined.

Step 2:

Once the dominating signature execution time threshold is chosen, identify applications for validation, *target applications*, as follows:

- a) Identify the *dominating application set*, a subset of signature-generating applications that have dominating signatures.

- b) Choose one target application per distinct dominating signature from the set of dominating applications. The formed set of target applications is used for validation.

Step 3:

Evaluate the accuracy of prediction for each co-schedule comprising target applications by comparing $IPC_{aggregate}$ for the predicted best priority pair and $IPC_{aggregate}$ for the actual best priority pair. This can be accomplished by creating the set of co-schedules that comprise each distinct pair of target applications and then following the procedure below for each co-schedule:

- a) Execute the co-schedule at all possible distinct priority pairs and record the $IPC_{aggregate}$ for each priority pair.
- b) Identify the best priority pair and record $IPC_{aggregate}$ for the best priority pair (BT).
- c) Look up the predicted best priority pair in the prediction table and Record $IPC_{aggregate}$ for the predicted best priority pair (PT).
- d) Compute the accuracy of the prediction as follows: $((PT - BT) / BT) * 100$.

Calculate the average accuracy for all co-schedules and if the desired accuracy is not reached, then some or all steps of Phases 1 and 2 of the methodology have to be repeated.

The workflow of our three-phase methodology is illustrated in Figure 5.2. As shown in this figure, the prediction accuracy is evaluated and if found to be below the user-defined accuracy, then some or all steps of Phases 1 and 2 of the methodology have to be repeated. Thus, our methodology is an iterative process. The proposed implementation of each step of the methodology may improve with as we explore different implementations for the POWER5 processor and implement the methodology for different processor architectures. The next chapter describes an implementation of the methodology for the IBM POWER5 processor.

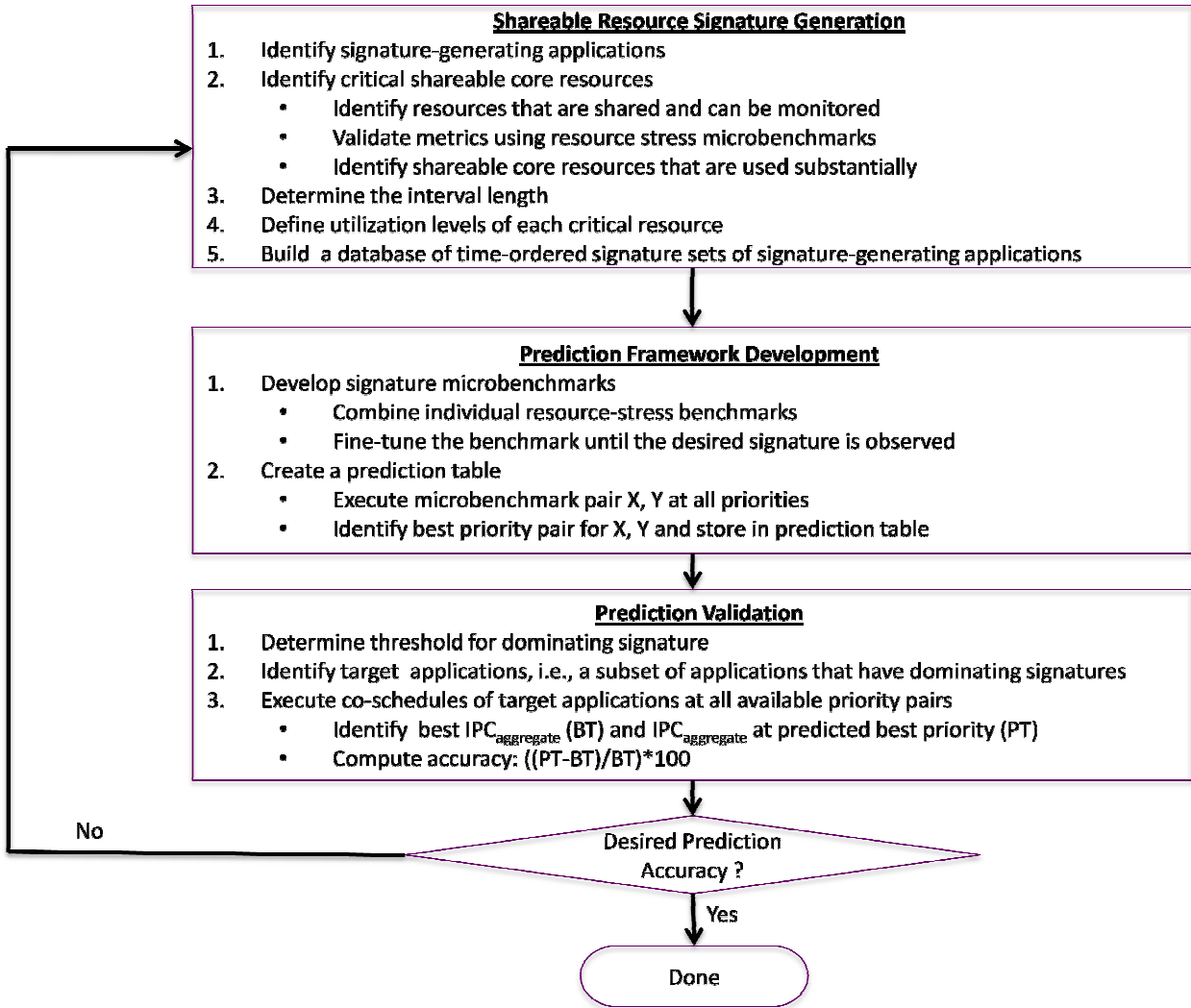


Figure 5.2: Best Priority Pair Prediction Methodology

6 IBM POWER5 Implementation

To assess the merit of our best priority pair prediction methodology, described in the previous chapter, this chapter presents an implementation of the methodology for the IBM POWER5 processor. This implementation evaluates six out of eight hardware thread priorities, described in Section 6.1, available to the operating system on the IBM POWER5 processor. Section 6.1 also discusses the experimental platform used in the implementation. Sections 6.2, 6.3, and 6.4 present our POWER5 implementation of Phases 1, 2, and 3 of the best priority pair methodology. Finally, Sections 6.5 and 6.6 discuss the lessons learned from this implementation and conclusions, respectively.

6.1 EXPERIMENTAL ENVIRONMENT

The experiments in this implementation were performed on an IBM p550 machine [49] with the Linux operating system [19]. The setup of this machine is described in Section 6.1.1. The hardware thread priorities evaluated in this implementation are described in Section 6.1.2.

6.1.1 Experimental Platform

An IBM p550 machine [49] was used for our POWER implementation. As shown in Figure 6.1, the p550 has two dual chip modules (DCM), each containing one dual-core POWER5 chip and an L3 cache chip. The two cores of a POWER5 chip on a DCM share an on-chip level-two (L2) cache and an off-chip L3 cache. All the cores share 32GB of main memory; the POWER5 core is described in detail in Section 2.5 and in [10].

The p550 machine was installed with the Open Suse Linux operating system [50] and runs Linux kernel 2.6.16.21-0.25 [19]. The stock Linux kernel uses hardware thread priorities to improve kernel performance. Hence, to prevent the kernel from interfering with our hardware thread priority settings, the hardware thread priority setting calls were removed from the kernel. The kernel modifications are described in Appendix D.

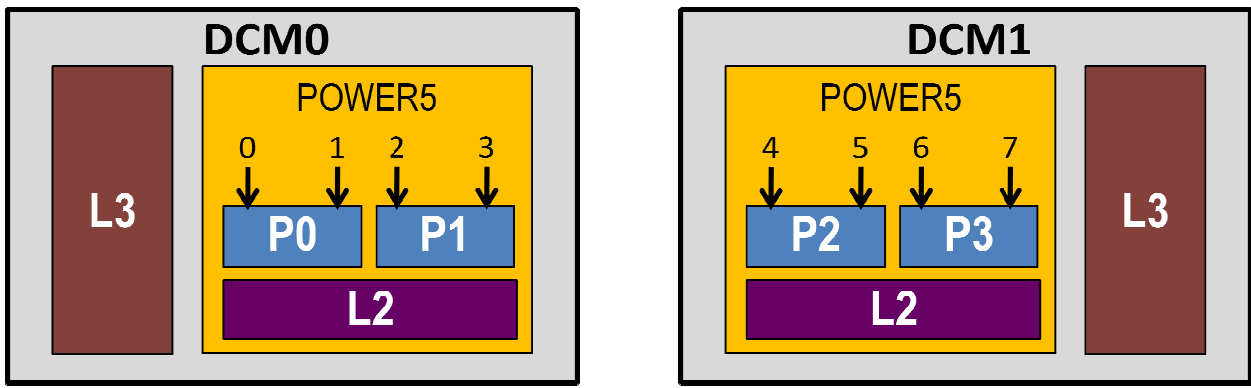


Figure 6.1: Two-DCM p550 Configuration

SMT was enabled on the system's processors and, thus, there are two cores per DCM and a total of eight hardware threads (0-7) available for process execution, threads 0-3 on the POWER5 chip on DCM0 and threads 4-7 on the POWER5 chip on DCM1. To reduce interference from operating system activity, the kernel processes were bound to threads 0-3. Furthermore, to be able to accurately profile the two hardware threads of one core, i.e., core P2, hardware threads 6 and 7 of core P3 were turned off. In this way, core P2 and hardware threads 4 and 5, which were used in our study, have full access to the resources of DCM1. SMT-related experiments were performed on threads 4 and 5, while single-threaded experiments were performed on thread 4, with no user activity on thread 5.

6.1.2 IBM POWER5 Hardware Thread Priorities

The IBM POWER5 implements hardware thread priorities, which control the instruction decode rate of the two threads of a core. The default priority settings (equal priorities) allows the two threads of a core to decode at the same rate, alternating turns every cycle, while unequal priority pairs allow one thread to decode at a greater rate than the other. As described in Section 2.5 in detail, the POWER5 has eight hardware thread priority levels. The hypervisor can assign all eight; the operating system can assign 1 through 6; and user applications can assign 2 through 4. A thread with priority 0 is not allowed to execute instructions. If both threads of a core have priority 1 then each thread is allowed to start up to five instructions every 64 cycles; this setting can potentially be used for power savings. Lower priorities

can be used by the operating system, for example, when one thread has no work to do, when one thread is waiting on a lock, or for power savings. Also, as our research and that of others show, adapting the decode rate can be used to control contention by two hardware threads for a core's shareable resources and, thus, improve utilization and throughput.

Let us denote the two threads of a core as Thread₀ and Thread₁. If Thread₀ has priority 1 and Thread₁ has a priority greater than 1, Thread₁ is assigned all the shared core resources, and Thread₀ gets what remains unused. If both threads have priorities greater than 1 the number of decode cycles allocated to the threads are shown in Table 6.1. If Thread₀ has priority A and Thread₁ has priority B and A is less than or equal to B then during each $2^{(A-B+1)}$ consecutive decode cycles, then one decode cycle is allocated to Thread₀, while all the remaining ones are allocated to Thread₁. For example, if Thread₀ has priority 4, and Thread₁ has priority 6, then Thread₀ gets one cycle out of every eight and Thread₁ gets the remaining seven cycles. Given that (1) the difference in the hardware thread priorities of Thread₀ and Thread₁ determines the allocation of decode cycles if both threads have priorities greater than 1, (2) only priority levels 1-6 can be assigned by the operating system, and (3) priority pair (1, 1) is used for power savings, our study uses only the 11 priority pairs listed in Table 6.1. Additional details about the IBM POWER5 hardware thread priorities are given in [4] and in Section 2.5.

Table 6.1: Thread Priority Pairs Considered in this Study

Thread ₀ Priority (A)	Thread ₁ Priority (B)	Priority Difference (A – B)	Thread ₀ Decode Cycle Share	Thread ₁ Decode Cycle Share
1	6	-5	Thread₀ gets resources not used by Thread₁	Thread₁ gets all shared core resources
2	6	-4	1/32	31/32
3	6	-3	1/16	15/16
4	6	-2	1/8	7/8
5	6	-1	1/4	3/4
6	6	0	1/2	1/2
6	5	1	3/4	1/4
6	4	2	7/8	1/8
6	3	3	15/16	1/16
6	2	4	31/32	1/32
6	1	5	Thread₀ gets all shared core resources	Thread₁ gets resources not used by Thread₀

6.2 PHASE 1: SHAREABLE RESOURCE SIGNATURE GENERATION

This section describes the implementation of the first phase of our methodology, which was described in Section 5.2 of Chapter 5. This phase, which consists of five steps, generates the archetypical signature for the IBM POWER5 processor. The first step of the phase, described in Section 6.2.1, identifies the set of signature-generating applications that are used in this implementation. The second step, described in Section 6.2.2, discusses how the four POWER5 critical shareable core resources (critical resources) were identified. This section also presents the resource-stress benchmarks that were used to validate the metrics that were employed to estimate utilization of the four critical resources. Section 6.2.3 describes the third step of this phase, which determines the interval length for this implementation. The fourth step, described in Section 6.2.4, determines the number of utilization levels for each of the four critical resources. Finally, the fifth step, described in Section 6.2.5, creates the POWER5's Shareable Resource Signature and builds the signature database, which stores the time-ordered signature sets of the signature-generating applications used in this implementation. Section 6.2.5 also summarizes the signatures that characterize the signature-generating applications.

6.2.1 Step 1: Identification of Signature-Generating Applications

The IBM POWER5 is a general-purpose processor and is used to run a large array of application classes. The application classes that may be targeted to run on this processor include but are not restricted to the following: business intelligence, transaction processing, scientific, high performance computing, compute-intensive, and file servers [63].

We chose a relatively small subset of application classes for this implementation and study the benchmarks from the following suites: SPEC CPU2006 [38], NAS NPB [52], and PETSc KSP [53]. The SPEC CPU2006 benchmarks represent characteristics of compute-intensive applications, while the NAS NPB and PETSc KSP benchmarks represent characteristics of computational fluid dynamics applications and scientific applications that utilize linear solvers, respectively. These benchmarks are described below.

SPEC CPU2006, a CPU-intensive benchmark suite described in detail in [38], stresses a system's processor, memory subsystem, and compiler. Although the entire suite contains 29 benchmarks, for the study we randomly selected a set of seven integer benchmarks from CINT2006: 429.mcf, 445.gobmk, 458.sjeng, 462.libquantum, 471.omnetpp, 473.astar, and 483.xlancbnmk, and 13 floating-point benchmarks from CFP2006: 410.bwaves, 416.gamess, 433.milc, 434.zeusmp, 435.gromacs, 437.leslie3d, 444.namd, 447.dealII, 450.soplex, 453.povray, 454.calculix, 459.gemsFDTD, and 470.lbm. Each benchmark was executed with the reference data set.

NAS NPB3.2 sequential, a benchmark suite available in sequential and parallel versions and described in detail in [52], is used for performance evaluation of parallel computers. It consists of three benchmarks: bt-mz, lu-mz, and sp-mz, which mimic the computational and communication aspects of large-scale computational fluid dynamics applications. The workloads were executed with data sets A, B, and C, which have memory requirements of 50 MB, 200 MB, and 0.8 GB, respectively; data sets bigger than C could not be run on our experimental platform.

PETSc KSP library is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. To create workloads for our study, sequential versions of each of the 10 PETSc KSP solvers, which are listed below, with one of 12 input data set, which are listed below, is embedded in a simple C program. The program is passed the KSP routine and the input data set as input parameters via the command line. A program instance calls only one routine, which executes on one input data set. The 12 data sets, which are listed below, were derived from real applications and are available for download with the PETSc libraries, which are described in detail in [53]. The source code of the C program is given in Appendix C.

- The PETSc KSP library is a suite of Krylov subspace iterative methods and preconditioners to solve linear equations. The following functions are used with their default preconditioners: bcgs, bicg, cgs, chebychev, cr, gmres, lsqr, Richardson, tcqmr, and tfqmr.

- The data sets used are: arco1, arco3, arco4, arco6, cfd1.10, cfd2.10, medium, poisson1, poisson2, poisson3, small, and tiny.

The benchmarks used in our study have execution times in the range of one through 4,000 seconds. The average POWER5 run time of the PETSc KSP solvers is 40 seconds; for the SPEC CPU2006 benchmarks, it is 1,800 seconds; and for the NAS NPB benchmarks, 120 seconds for data set A, 480 seconds for data set B, and 2200 seconds for data set C. Thus, the workloads represent a mix of short- and long-running applications.

6.2.2 Step 2: Identification of Critical Shareable Core Resources

As described in Section 5.2, to determine the critical shareable core resources, first, we determine the set of resources on a POWER5 core that can be shared by the hardware threads in SMT mode. As shown by the non-shaded resources in Figure 6.2 and also shown in Figure 2.5, most of the resources are shared with the exception of the program counters, instruction buffers, return stack, store queue, and group completion stage.

In this implementation we use hardware performance counters to monitor the usage of resources on a POWER5 core. Hardware performance counters are on-chip hardware that permits detailed low-level monitoring of performance-related events that occur on a core. Compared to software profilers, the use of hardware counters contributes much less perturbation to application execution time. We employ the following monitoring tools to facilitate the use hardware performance counters: (1) pmcount [42], which was used to capture performance counter data for the execution of the SPEC CPU2006 and NAS NPB3.2 sequential benchmarks, and (2) PAPI [51], which was used to capture performance counter data for the execution of the PETSc KSP library. The pmcount tool can be used to attach to a running process and does not require source code modification, whereas the PAPI tool provides an API to instrument source code. We used PAPI for the PETSc KSP library, which were programmed by us, to skip the profiling of initial portions of the code that wait for the user to input the name of the KSP solver and

data set to be used for the given program instance. In contrast, the SPEC CPU2006 and NAS NPB3.2 benchmarks do not require user input and, hence, we used Pmcount to profile them.

Using pmcount or PAPI, we identify the subset of shareable core resources for which utilization can be monitored by hardware performance counter events, i.e., monitorable shareable core resources. The set of monitorable shareable core resources, highlighted in Figure 6.2, were identified as follows: the floating-point units (FPUs), fixed-point units (FXUs), and entire cache hierarchy in terms of instructions and data, as well as address translation resources. Additionally, there are counter events that can be used to know when the shared issue queues or the shared register mappers are full. However, in these cases, we can neither estimate the number of issue queue entries that are used when the issue queues are not full, nor the number of register mappers used when they are not full. Hence, we identify critical shareable core resources from the set of monitorable shareable core resources highlighted in Figure 6.2

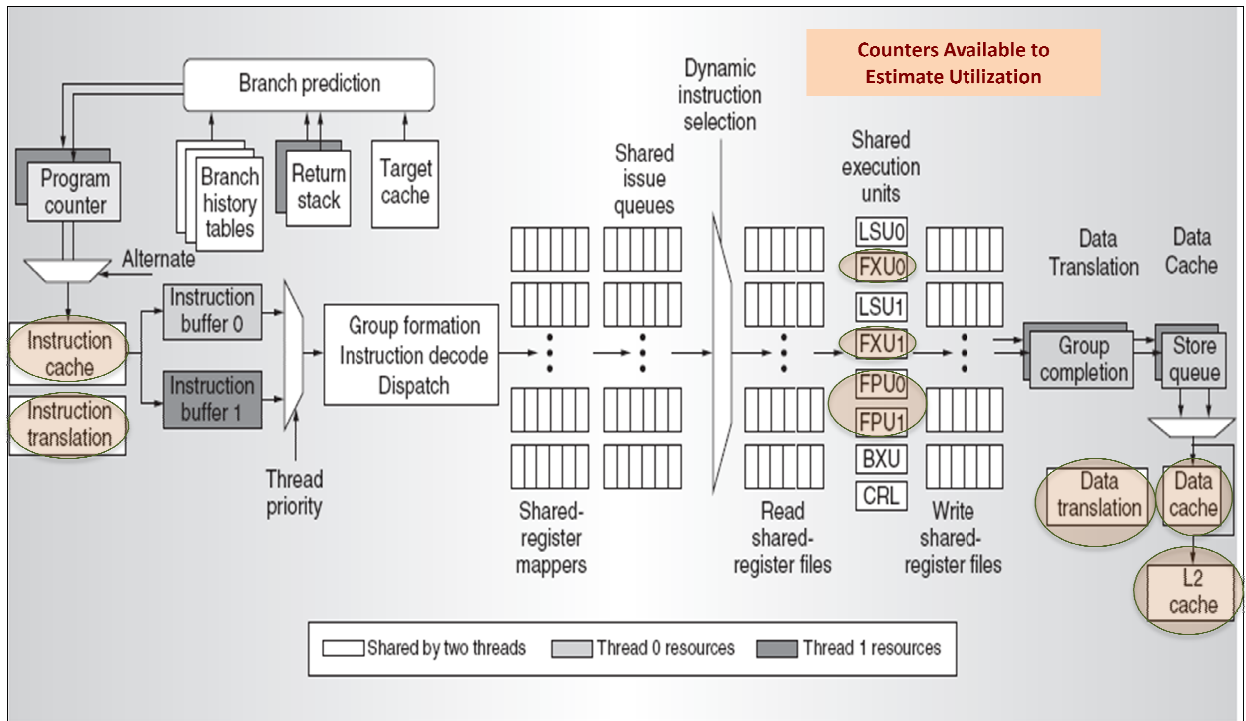


Figure 6.2: POWER5 Shareable Core Resources that can be Monitored by Performance Counters

Rather than use the signature-generating applications and POWER5 performance counters to identify the set of critical shareable core resources (critical resources), as proposed in Section 5.2, we use results of our pilot study, described in Chapter 4. This step of Phase 1 of our methodology was implemented in the initial stages of our research, thus, it does not follow the suggested implementation described in Chapter 5. The suggested implementation is the result of further experimentation that indicates that the application coverage of the best priority pair prediction methodology is dependent upon the workloads used to determine the critical resources. Accordingly, in future implementations we propose to use the signature-generating applications to implement this step. Given the prediction accuracy of this implementation, described in Section 6.4, for this implementation we chose not to implement this step of Phase 1.

The pilot study, which evaluated the efficacy of using non-default priority settings, was conducted using a trace-driven POWER5 simulator and partial instruction traces of SPEC CPU2000, SPEC CPU2006, Imbench, and Stream2 benchmarks. The results of this study include resource utilization data for each trace when executed on the simulator in single-threaded mode.

As described in Section 5.2, we identify the set of resources as those that experienced at least 5% utilization by one or more benchmark suites. The simulation data was analyzed for time spent using the FPU, FXU, as well as the time spent accessing the entire cache and TLB hierarchy. The simulation data, presented in Appendix H, indicates that, for all the simulations, TLB misses generated by instruction fetches and L1 I-cache misses account for an average of less than 5% of the total execution cycles of each benchmark suite. In contrast, the other candidate resources are associated with an average of at least 5% of the total execution time of one or more benchmark suites. Note that while waiting for a cache miss, TLB miss, or busy functional unit, the processor may overlap execution with other instructions and, thus, hide the associated latency. Hence, the time spent waiting for a busy resource only indicates the worst-case impact on throughput. As a result of analysis of our simulation study, we classified all monitorable shareable resources except the I-cache and the hardware associated with the ITLB as the POWER5's set of critical resources. This set comprises the FPUs, FXUs, and the remainder

of the cache hierarchy, including the hardware associated with the DTLB. Next, as explained in Section 5.2, we analyze the POWER5 architecture to prune the set of resources identified by the simulation data.

Applications that have relatively high FPU (FXU) utilization are sensitive to FPU (FXU) performance. If two such applications are co-scheduled, it is likely that both will experience performance degradation, as compared to when each executes in isolation (in single-threaded mode). A co-schedule comprised of two applications that have relatively high utilization of the L2 cache, for data accesses, can lead to expensive off-chip L3 cache and main memory accesses. One other study [27] also points to the L2 cache as a source of contention on SMT processors. Similarly, a co-schedule comprised of two applications that have relatively high utilization of the TLB for data address translation can lead to costly main memory accesses.

Thus, based on an analysis of our initial simulations and our understanding of the POWER5 architecture, we determine that the following four resources are the critical resources for the POWER5: 1) floating-point unit (FPU), 2) fixed-point unit (FXU), 3) L2 cache for data accesses, and 4) TLB for data address translations. Hence, four critical resources, $N=4$, are used to form a signature. Note that with the exception of the TLB, the resources we identified are also used in [34] in which the authors develop benchmarks that stress one of these three resources in addition to the main memory and L1 cache. This work shows that for co-schedules of these benchmarks non-equal priority pairs can improve throughput over that achieved with default (equal) priorities by as much as 23%. We also studied the impact of POWER5 priorities in our pilot study (described in Chapter 4), which was performed before [34] was published. This study showed that non-equal priorities can improve throughput over that achieved with equal priorities by as much as 35%.

Critical Resource Utilization Metrics:

Next, given the four critical shareable core resources (critical resources), we develop and validate metrics to estimate their utilization. The utilization metrics and the methods used to realize them on the IBM POWER5 using hardware performance counter events are described below.

FPU Utilization, U_{FPU} : The two floating-point units (FPUs) of a POWER5 core execute short- and long-latency floating-point (FP) instructions, which have 64-bit operands. FP instructions are executed in the order in which they enter the pipeline. When a long-latency FP instruction (divide or square root) enters the pipeline, the FPU is not available (does not allow other instructions to enter) for either 26 (FP divide) cycles, as shown in Figure 6.4, or 30 (FP square root) cycles. In contrast, as shown in Figure 6.3, the execution of a short-latency floating-point instruction (add, multiply, subtract, or add-multiply) is pipelined. After the six-stage pipeline is full of short-latency instructions, the execution of one short-latency instruction is completed every cycle. A long-latency instruction can enter the FPU while the execution of one or more short-latency instructions is in progress. Given this implementation, an FPU is utilized and not available when the execution of either an FP divide or square root instruction is in progress. In addition, even though a short-latency instruction takes six cycles to execute, when the FPU is available, such an instruction uses only one “slot” in the execution pipeline – the other five are available. Accordingly, the utilization of the two FPUs of a POWER5 core, U_{FPU} , during an application’s interval length can be measured by the following metric:

$$U_{FPU} = (FPU_Cycles_Utilized / (Total_Cycles * 2)) * 100,$$

where *Total_Cycles* is the number of cycles required to execute the program and *FPU_Cycles_Utilized* = $(Num_FDIVInstrs * FDIV_Latency) + (Num_FSQRTInstrs * FSQRT_Latency) + Num_ShortFPInstrs$, where *Num_FDIVInstrs*, *Num_FSQRTInstrs*, and *Num_ShortFPInstrs* are the number of FDIV, FSQRT, and short-latency floating-point instructions executed by the program; and *FDIV_Latency* and *FSQRT_Latency* are the number of cycles required to execute the floating-point divide and square root instructions.

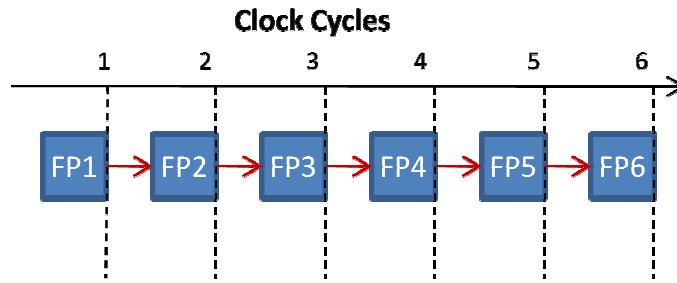


Figure 6.3: Six-cycle Pipelined Execution of Short Latency
Floating-point Instructions on a POWER5 FPU

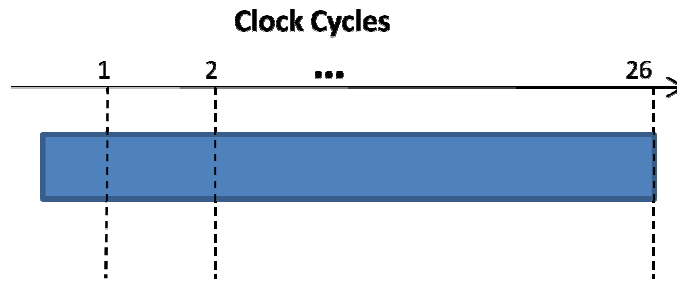


Figure 6.4: Number of Cycles the Issue slot is not Available during the Execution
of an FDIV Instruction on a POWER5 FPU

We know the latencies of FDIV and FSQRT instructions: $FDIV_Latency = 26$ and $FSQRT_Latency = 30$. To obtain values for the remaining terms of the U_{FPU} equation using POWER5 performance counters, we use the following performance counter events:

- RUN_CYC: number of cycles to execute the program
- FPU0_FIN: number of cycles that FPU0 produces a result, a short- or long-latency floating-point instruction
- FPU1_FIN: number of cycles that FPU1 produces a result, a short- or long-latency floating-point instruction
- FDIV: number of FDIV instructions executed in both FPU0 and FPU1
- FSQRT: number of FSQRT instructions executed in both FPU0 and FPU1

FPU0_FIN and FPU1_FIN measure the number of cycles a result was produced by FPU0 and FPU1, respectively. Every instruction that is issued to a unit will produce a result after it executes (although it may not be committed because of a mispredicted branch) and, thus, these counters also reflect the number of cycles the FPU was issued a floating-point instruction. Thus, the total number of floating-point instructions issued = FPU0_FIN + FPU1_FIN.

Counters FDIV and FSQRT measure the total number of floating-point divide instructions and floating-point square root instructions executed on the processor. Since all instructions that are issued, are executed (but may not be committed due to a mispredicted branch), we use these counters to measure the number of FDIV and FSQRT instructions issued to the FPUs, thus,

$$\begin{aligned} \text{Num_FDIVInstrs} &= \text{FDIV} \text{ and} \\ \text{Num_FSQRTInstrs} &= \text{FSQRT}. \end{aligned}$$

Given the total number of floating-point instructions issued, which is given by (FPU0_FIN + FPU1_FIN) and the total number of long-latency instructions issued, which is given by (FDIV + FSQRT), the number of short-latency floating-point instructions issued can be calculated as follows:

$$\text{Num_ShortFPInstrs} = (\text{FPU0_FIN} + \text{FPU1_FIN}) - (\text{FDIV} + \text{FSQRT}).$$

Using these metrics, U_{FPU} is computed as follows:

$$U_{\text{FPU}} = (\text{FPU_Cycles_Utilized} / (\text{RUN_CYC} * 2)) * 100,$$

where $\text{FPU_Cycles_Utilized} =$

$$(\text{FDIV} * 26) + (\text{FSQRT} * 30) + ((\text{FPU0_FIN} + \text{FPU1_FIN}) - (\text{FDIV} + \text{FSQRT})).$$

FXU Utilization, U_{FXU} : The two fixed-point units (FXUs) of a POWER5 core execute all instructions in one cycle. Fixed-point (FX) instructions are executed in the order in which they are issued. When an instruction is executing, the FXU is not available for issue for only one cycle. Given this implementation, an FXU is utilized and not available for only one cycle while executing an instruction. Accordingly, the utilization of the two FXUs of a POWER5 core, U_{FXU} , during an application's interval length can be measured by the following metric:

$$U_{\text{FXU}} = (\text{FXU_Cycles_Utilized} / (\text{Total_Cycles} * 2)) * 100,$$

where *Total_Cycles* is the number of cycles required to execute the program and *FXU_Cycles_Utilized* = *Num_FXInstrs*, where *Num_FXInstrs* is the number of fixed-point instructions executed by the program.

To obtain values for the remaining terms of the U_{FXU} equation using POWER5 performance counters, we use the following performance counter events:

- RUN_CYC: number of cycles to execute the program
- FXU0_FIN: number of cycles FXU0 produces a result
- FXU1_FIN: number of cycles FXU1 produces a result

FXU0_FIN and FXU1_FIN measure the number of cycles a result was produced by FXU0 and FXU1, respectively. Every instruction that is issued to a unit will produce a result after execution (but may not be committed due to a branch misprediction) and, thus, these counters also reflect the number of cycles the FXU was issued an instruction. Thus,

$$\text{Num_FXInstrs} = \text{FXU0_FIN} + \text{FXU1_FIN}.$$

Using these events, U_{FXU} is computed as follows:

$$U_{FXU} = (FXU_Cycles_Utilized / (RUN_CYC * 2)) * 100,$$

where $FXU_Cycles_Utilized = FXU0_FIN + FXU1_FIN$.

L2 Cache Utilization, $U_{L2Cache}$: A POWER5 processor core has a two-way, 64 KB, L1 instruction cache and a four-way, 32KB, L1 data cache per core, whereas the two cores share a unified, 10-way, 1.9MB, L2 cache and a 12-way, 36MB, L3 cache, which serves as a victim cache for the L2 cache. The L1 caches are indexed by virtual addresses, whereas the L2 and L3 caches are indexed by physical addresses. The cache line size of the L1 and L2 caches is 128 bytes, whereas the line size of the L3 cache is 256 bytes. The L1 data cache is a write-through cache and does not allocate a line on a write miss. The L2 cache is inclusive of the L1 caches and implements a write-back policy and allocates a line on a write miss. The two cores also share a unified, L3 cache, which implements a write-back policy. The L3 cache is used as a victim cache and it is not inclusive of the L2 cache. Thus, evicted lines from the L2 cache are stored in the L3 cache. Both the L2 and L3 caches implement consistency using a snoopy protocol and consistency is maintained for all cache levels. In addition, all cache levels use the LRU replacement policy.

In addition to the caches, the POWER5 processor has hardware pre-fetching for instructions and data. An L1 cache miss generates a lookup in the pre-fetch buffers and if the referenced instruction/data is not found, a lookup in the L2 cache commences. If not found in the L2 cache, then the miss is resolved in the off-chip L3 cache, a remote L3 cache, or main memory. Since our experimental machine, described previously, has two dual-chip modules, each of which has a POWER5 chip and L3 cache, the miss also may be resolved in a remote cache, i.e., the L3 cache of the other module. As noted in [43], L2 cache misses are expensive to resolve and may take hundreds of cycles.

A line of a set in the L2 cache is not available if it is utilized; similarly, a set in the L2 cache is not available if all lines of the given set are utilized. If all lines of all sets of the L2 cache are occupied

then the L2 cache is fully utilized. The capacity of a cache (number of lines) that is utilized indicates the utilization and availability of the L2 cache.

To estimate L2 cache utilization, we use the L1 data cache miss rate. Misses in the L1 data cache may get resolved in the L2 cache or in lower levels of the memory hierarchy. Since the L2 cache is inclusive of the L1, miss data are stored in both the L1 data cache and the L2 cache. Thus, an L1 data cache miss results in an increase in L2 cache utilization (even if it replaces a line in the cache, which is likely). Thus, our metric assumes that higher L1 data-cache miss rates result in higher utilization of the L2 cache. Accordingly, the utilization of the L2 cache, $U_{L2Cache}$, is computed using the L1 data cache miss. It is calculated as follows:

$$U_{L2Cache} = (Num_L1DCache_misses / Num_L1DCache_Refs) * 100,$$

where $Num_L1DCache_misses$ and $Num_L1DCache_Refs$ are the number of load (read) and store (write) L1 data cache misses and the number of load and store references to the L1 data cache, respectively.

To obtain values for the other terms of the $U_{L2Cache}$ equation using POWER5 performance counters, we use the following performance counter events:

- ST_REF_L1: number of stores
- LD_REF_L1: number of loads
- ST_MISS_L1: number of L1 cache write misses
- LD_MISS_L1: number of L1 cache read misses

Using these performance counter events,

$$Num_L1DCache_misses = ST_MISS_L1 + LD_MISS_L1 \text{ and}$$

$$Num_L1DCache_Refs = ST_REF_L1 + LD_REF_L1.$$

Accordingly, $U_{L2Cache}$ is computed as follows:

$$U_{L2Cache} = ((ST_MISS_L1 + LD_MISS_L1) / (ST_REF_L1 + LD_REF_L1)) * 100.$$

TLB Utilization, U_{TLB} : Each core of a POWER5 chip has 128-entry ERATs, one for instruction-address translation (i-ERAT) and one for data-address translation (d-ERAT). The i-ERAT is two-way set associative with a FIFO replacement policy. The d-ERAT is fully associative with an LRU replacement policy. In addition, the core also has a unified 1024-entry, four-way set-associative translation lookaside buffer (TLB) to resolve ERAT misses. Address translations are first looked up in an ERAT and on a miss they are looked up in a TLB; misses in a TLB are resolved in the memory hierarchy.

An entry of a set in a TLB is not available if it is utilized; similarly, a set in a TLB is not available if all entries of the given set are utilized. If all entries of all sets of a TLB are occupied then the TLB is fully utilized. The number of entries in the TLB that are utilized indicates the utilization and availability of the TLB.

To estimate TLB utilization, we use the d-ERAT miss rate. d-ERAT misses are resolved in the TLB or in lower levels of the memory hierarchy. Since the TLB is inclusive of the ERATs, d-ERAT miss data are stored in both the d-ERAT and TLB. Thus, a d-ERAT miss results in an increase in TLB utilization and higher d-ERAT miss rates result in higher TLB utilization. Accordingly, our metric for estimating TLB utilization, U_{TLB} , uses the miss rate in the d-ERAT and is calculated as follows:

$$U_{TLB} = (Num_DERAT_misses / Num_DERAT_Refs) * 100,$$

where Num_DERAT_misses and Num_DERAT_Refs are the number of data-address translations that missed in the d-ERAT and the number of data-address translations references to the d-ERAT, respectively.

To obtain values for the other terms of the U_{TLB} equation using POWER5 performance counters, we use the following performance counter events:

- ST_REF_L1: number of stores
- LD_REF_L1: number of loads
- LSU_DERAT_MISS: number of misses in the data ERAT (all such misses access the TLB)

Thus, using the above performance counter events,

$$Num_DERAT_misses = LSU_DERAT_MISS \text{ and}$$

$$Num_DERAT_Refs = ST_REF_L1 + LD_REF_L1.$$

Using these metrics, U_{TLB} is computed as follows:

$$U_{TLB} = (LSU_DERAT_MISS / (ST_REF_L1 + LD_REF_L1)) * 100.$$

Validation of Metrics:

In order to validate the methods used to realize the metrics to estimate an application's utilization of the POWER5's critical resources during an interval length, resource-stress microbenchmarks were designed. Each benchmark, which is associated with one critical resource, the targeted resource, is designed to maximize the utilization of that resource. To accomplish this, each benchmark is designed to maximize the number of instructions that utilize its targeted resource and to minimize the number of instructions that do not use it, i.e., that use other microarchitectural resources. Note that 100% utilization of a resource requires that the resource be used 100% of the time, i.e., it must be used every cycle of execution time; for most resources, this is impossible. Nonetheless, a resource-stress microbenchmark is designed to approach 100% utilization of the targeted resource and, thus, the total number of cycles needed to execute the program is highly dominated by cycles used to execute instructions that utilize the targeted resource.

To ascertain the accuracy of the utilization metric for a critical resource, we predict the execution time of a co-schedule comprising two instances of the resource's stress microbenchmark executed on the two cores of a POWER5 processor in SMT mode. Then the predicted execution time is compared to the observed execution time of the co-schedule executed in SMT mode. If the observed execution time can be verified to be within 5% of the predicted execution time, then we declare that the method used to realize the metric is valid.

The following sections describe the design of the four POWER5 resource-stress microbenchmarks and the experiments conducted to validate the method used to realize the corresponding utilization metrics; the complete source code of the microbenchmarks is presented in Appendix A.

Floating-point Unit (FPU) Utilization Metric:

As shown in Figure 6.5, the main loop of the FPU-stress microbenchmark contains no load or store operations and, hence, there are no memory accesses to fetch data. To implement this benchmark, the following steps can be followed to maximize FPU utilization and minimize utilization of other core resources:

1. Determine the number of programmable floating-point registers, N . To maximize the floating-point instruction issue rate to the FPUs, the loop body is sequential code that is a sequence of floating-point instructions that do not have data dependences (without branches). Figure 6.5 shows such a loop; it contains N floating-point instructions that use different floating-point (FP) registers, i.e., the first instruction uses FP register 1, the second uses FP register 2, ..., the N^{th} uses FP register N . Hence, there are N instructions without data dependences.
2. Initialize the loop unrolling factor, X , to 1. Loop unrolling decreases the number of branch instructions and reduces the number of cycles associated with loop control code, however, it results in increased code size and can result in register spills.

3. If the code size is greater than half the capacity of the instruction cache, decrease the unrolling factor, X , by one and repeat step 3. Code that fits in half the L1 instruction cache permits a co-schedule comprising two copies of the microbenchmark to execute in SMT mode without experiencing L1 instruction cache misses, except for compulsory misses.
4. Run the loop and detect register spills (L1 data cache misses). If there are any, decrease the unrolling factor, X , by one and repeat step 4. Using more registers than physically available may lead to register spills, which results in data being written back to memory due to a lack of mapped registers and, potentially, L1 data cache misses. This step along with the previous one also ensure that TLB misses, except compulsory misses, are avoided as well.
5. Run the loop and determine FPU utilization. If it is not 100%, then increase the unrolling factor, X , by one and go to step 3.
6. Set the number of loop iterations, max . The number of loop iterations should be set so that the time it takes to execute the specified number of iterations is sufficiently larger than the time spent servicing compulsory L1 cache misses associated with the first loop iteration. Accordingly, the execution time of the subsequent iterations of the loop should be dominated by intrinsic instruction execution.


```

Loop:

FP instruction1 using register FP1
FP instruction2 using register FP2
...
FP instructionN using register FPN

Above code copied X times making sure
that the code size does not exceed half
the size of L1 instruction cache

count++;
branch to loop if count<max

```

Figure 6.5: Main Loop of FPU-Stress Microbenchmark

Next, we implemented the FPU-stress microbenchmark on the POWER5; the source code is provided in Appendix A. In order to avoid the effects of undesirable compiler optimizations we used the powerpc assembly language to write the loop depicted in Figure 6.5. The implementation is discussed below:

1. The POWER5 has 32 programmable floating-point registers, i.e., $N = 32$. Thus, the loop body of the microbenchmark contains 32 floating-point instructions, where each instruction is of the form: `fadd R_i , R_i , R_i` , where $i = 1, 2, \dots, 32$.
2. The loop unrolling factor, X , is 4. This resulted in a code size of 15KB, which is less than half the size of the 32KB L1 instruction cache, i.e., less than 16KB. As shown below, execution of the main loop of the microbenchmark generates no L1 data cache accesses and, hence, it avoids register spilling. Execution of one iteration of the loop results in the execution of 160 floating-point add instructions.
3. The execution time of one loop iteration is 1 millisecond. The number of iterations, max , is set to 40 billion to achieve an execution time of 130 seconds, which is dominated by the execution of the last $max-1$ iterations.

We predicted that this microbenchmark's FPU utilization is 99.9%, i.e., $U_{FPU} = 99.9\%$. For our implementation of the microbenchmark, we verified that no loop execution cycles for the last $max-1$

iterations are associated with cache and TLB misses. In addition to counting the performance counter events needed to calculate U_{FPU} , using pmcount and the following events to measure load on the caches and TLBs: LD_REF_L1, ST_REF_L1, LSU_DERAT_MISS, DTLB_MISS, DATA_FROM_L3, DATA_FROM_LMEM, DATA_FROM_RMEM, INST_CMPL, ITLB_MISS, INST_FROM_L1, INST_FROM_L2, INST_FROM_L3, INST_FROM_LMEM, and INST_FROM_RMEM. These performance event counts indicate the following:

- Of the $1.51 * 10^{12}$ instructions fetched, 99.99% were fetched from the L1 instruction cache, 0.0004% were fetched from the L2, unified cache, and a negligible amount were fetched from the L3 cache. There were no instructions fetched from main memory. This indicates that a minimal amount of the program's execution cycles are associated with L1 instruction cache misses.
- Less than 0.001% of the $1.51 * 10^{12}$ instructions executed are load and stores, which are used to initialize variables and perform other bookkeeping operations. These instructions were resolved in the L1 and L2 caches. This indicates that a minimal amount of the program's execution cycles are associated with L1 data cache misses.
- With respect to address translation, there were 0.002% misses in the L1 ERAT, and out of these ERAT misses 0.37% missed the TLB. Although these numbers are very small, we were expecting no ERAT misses since the instruction code size is less than a page long and there is no memory allocated for data. The larger than expected number of ERAT misses could be attributed to the pmcount tool that was used to obtain the event counts.

To validate the method used to measure FPU utilization, two instances of the microbenchmark were executed concurrently on the two hardware threads of one POWER5 processor core with the same hardware thread priority in SMT mode. Since equal priorities cause the processor to allocate 50% of the decode cycles to each thread, if the threads execute the same set of instructions, then we expect each

thread to get a 50% share of each FPU. Consequently, we predict that the co-schedule execution time will be double that of the benchmark when executed in single-threaded mode.

The experiment shows that the total number of cycles (RUN_CYC) required to execute the co-schedule, i.e., two instances of the FPU-stress microbenchmark, is double that required for the execution of the benchmark in single-threaded mode. This indicates that the method that we propose to use to measure FPU utilization is valid.

Validation of Fixed-point Unit (FXU) Metric:

As shown in Figure 6.6, the main loop of the FXU-stress microbenchmark contains no load or store operations and, hence, there are no memory accesses to fetch data. To implement this benchmark, the following steps, which are almost identical to those followed to implement the FPU-stress microbenchmark, can be followed to maximize FXU utilization and minimize utilization of other core resources:

1. Determine the number of programmable floating-point registers, N . To maximize the fixed-point instruction issue rate to the FXUs, the loop body is sequential code that is a sequence of fixed-point instructions that do not have data dependences (without branches). Figure 6.6 shows such a loop; it contains N fixed-point instructions that use different fixed-point (FX) registers, i.e., the first instruction uses FX register 1, the second uses FX register 2, ..., the N^{th} uses FX register N . Hence, there are N instructions without data dependences.
2. Initialize the loop unrolling factor, X , to 1. Loop unrolling decreases the number of branch instructions and reduces the number of cycles associated with loop control code, however, it results in increased code size and can result in register spills.
3. If the code size is greater than half the capacity of the instruction cache, decrease the unrolling factor, X , by one and repeat step 3. Code that fits in half the L1 instruction cache permits a co-schedule comprising two copies of the microbenchmark to execute in

SMT mode without experiencing L1 instruction cache misses, except for compulsory misses.

4. Run the loop and detect register spills (L1 data cache misses). If there are any, decrease the unrolling factor, X , by one and repeat step 4. Using more registers than physically available may lead to register spills, which results in data being written back to memory due to a lack of mapped registers and, potentially, L1 data cache misses. This step along with the previous one also ensure that TLB misses, except compulsory misses, are avoided as well.
5. Run the loop and determine FXU utilization. If it is not 100%, then increase the unrolling factor, X , by one and go to step 3.
6. Set the number of loop iterations, max . The number of loop iterations should be set so that the time it takes to execute the specified number of iterations is sufficiently larger than the time spent servicing compulsory L1 cache misses associated with the first loop iteration. Accordingly, the execution time of the subsequent iterations of the loop should be dominated by intrinsic instruction execution.

Loop:

FP instruction1 using register FP1
FP instruction2 using register FP2
...
FP instructionN using register FPN

Above code copied X times making sure
that the code size does not exceed half
the size of L1 instruction cache

count++;
branch to loop if count<max

Figure 6.6: Main Loop of FXU-Stress Microbenchmark

Next, we implemented the FXU-stress microbenchmark on the POWER5; the source code is provided in Appendix A. In order to avoid the effects of undesirable compiler optimizations we used the powerpc assembly language to write the loop depicted in Figure 6.6. The implementation is discussed below:

1. The POWER5 has 32 programmable fixed-point registers, i.e., $N = 32$. Thus, the loop body of the microbenchmark contains 32 floating-point instructions, where each instruction is of the form: add R_i, R_i, R_i , where $i = 1, 2, \dots, 32$.
2. The loop unrolling factor, X , is 6. This resulted in a code size of 2.5KB, which is less than half the size of the 32KB L1 instruction cache, i.e., less than 16KB. As shown below, execution of the main loop of the microbenchmark generates no L1 data cache accesses and, hence, it avoids register spilling. Execution of one iteration of the loop results in the execution of 224 fixed-point add instructions.
3. The execution time of one loop iteration is 1 millisecond. The number of iterations, max , is set to 40 billion to achieve an execution time of 130 seconds, which is dominated by the execution of the last $max-1$ iterations.

We predicted that this microbenchmark's FPU utilization is 70%, i.e., $U_{FXU} = 70\%$. For our implementation of the microbenchmark, we verified that no loop execution cycles for the last $max-1$ iterations are associated with cache and TLB misses. In addition to counting the performance counter events needed to calculate U_{FXU} , using pmcount and the following events to measure load on the caches and TLBs: LD_REF_L1, ST_REF_L1, LSU_DERAT_MISS, DTLB_MISS, DATA_FROM_L3, DATA_FROM_LMEM, DATA_FROM_RMEM, INST_CMPL, ITLB_MISS, INST_FROM_L1, INST_FROM_L2, INST_FROM_L3, INST_FROM_LMEM, and INST_FROM_RMEM. These performance event counts indicated the following:

- Of the $2.17 * 10^{12}$ instructions fetched, 99.99% were fetched from the L1 instruction cache, 0.0005% were fetched from the L2, unified cache, and a negligible amount were fetched from the L3 cache. There were no instructions fetched from main memory. This indicates that a minimal amount of the program's execution cycles are associated with L1 instruction cache misses.
- Less than 0.001% of the $2.17 * 10^{12}$ instructions executed are load and stores, which are used to initialize variables and perform other bookkeeping operations. These instructions were resolved in the L1 and L2 caches. This indicates that a minimal amount of the program's execution cycles are associated with L1 data cache misses.
- With respect to address translation, there were 0.00036% misses in the L1 ERAT, and out of these ERAT misses 0.4% missed the TLB. Although these numbers are very small, we were expecting no ERAT misses since the instruction code size is less than a page long and there is no memory allocated for data. The larger than expected number of ERAT misses could be attributed to the pmcount tool that was used to obtain the event counts.

To validate the method used to measure FXU utilization, two instances of the microbenchmark were executed concurrently on the two hardware threads of one POWER5 processor core with the same hardware thread priority in SMT mode. Since equal priorities cause the processor to allocate 50% of the decode cycles to each thread, if the threads execute the same set of instructions, then we expect each thread to get a 50% share of each FXU, instead of the 70% utilized when running alone in single-threaded mode. Consequently, we predict that the co-schedule execution time will be 140% of the execution time of the benchmark when executed in single-threaded mode.

The experiment shows that the total number of cycles (RUN_CYC) required to execute the co-schedule, i.e., two instances of the FXU-stress microbenchmark, is, indeed, 140% of that required for the execution of the benchmark in single-threaded mode. This indicates that the method that we propose to use to measure FXU utilization is valid.

Validation of L2 Cache Metric:

In order to stress the L2 cache, the main loop of the L2 cache-stress microbenchmark, depicted in Figure 6.7, accesses a one-dimensional integer array with a stride such that all accesses miss the L1 data cache. To implement this benchmark the array size and the access stride must ensure that L2-cache utilization is maximized and that utilization of all other microarchitectural resources are minimized. To implement this benchmark, the following steps can be followed:

1. Initialize the array access stride, *stride_Cache*, to one cache line. As shown in Figure 6.7, during each iteration of the main loop of the microbenchmark, the array is accessed starting with the first element with a stride equal to *stride_Cache*.
2. Create an integer array of size equal to L2 cache, i.e., *array_size* = size of L2 cache. To maximize L2 cache utilization and to ensure that during a loop iteration all L1 data cache accesses are misses, select an array size such that during an iteration all L1 data cache lines are evicted at least once and the full capacity of the L2 cache is utilized.
3. Execute an iteration of the outer *for* loop and count the number of L1 data cache misses generated. If the iteration has 100% L1 data cache misses, then go to step 5. As shown in Figure 6.7, an iteration of the inner *while* loop accesses the array, starting with the first element, with a stride equal to *stride_Cache*. For example, Figure 6.8 depicts the access pattern with a stride of one cache line. In this figure, assuming that 32 array elements fit in one cache line, a stride of 32 array elements results in accessing only one element of each line and, as a result, each access should result in a miss.
4. Empirically adjust the values of *stride_Cache* and *array size*, and go to step 3. The presence of hardware pre-fetching may reduce the number of expected L1 data cache misses and, hence, it may be necessary to adjust the values of *stride_Cache* and *array size* to achieve 100% L2 cache utilization.
5. Set the number of loop iterations, *max*. The number of loop iterations should be set so that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 instruction cache misses for the first loop iteration. Thus, the execution

time of the benchmark should be dominated by the execution of the loop, i.e., the satisfaction of L2 cache misses, for subsequent iterations.

```

/***** MAIN LOOP BODY *****/
For(j=0;j<max;j++)
    While not end of array do:
        one memory access at distance of stride_Cache
    done

```

Figure 6.7: Main Loop of L2 Cache-Stress Microbenchmark

Next, we implemented the L2 cache-stress microbenchmark on the POWER5; the source code is provided in Appendix A. The implementation is discussed below:

1. *stride_Cache* was set to three cache lines (384 bytes), i.e., 96 four-byte integer array elements. This resulted in a 99.78% L2 cache utilization.
2. *array_size* was set to the size of the L2 cache, i.e., 1,920KB.
3. The execution time of one loop iteration is 10 milliseconds. The number of iterations, *max*, was set to 1,825,000, which results in an execution time of 100 seconds. In this way, the execution time is dominated by the execution time associated with the last *max*-1 iterations.

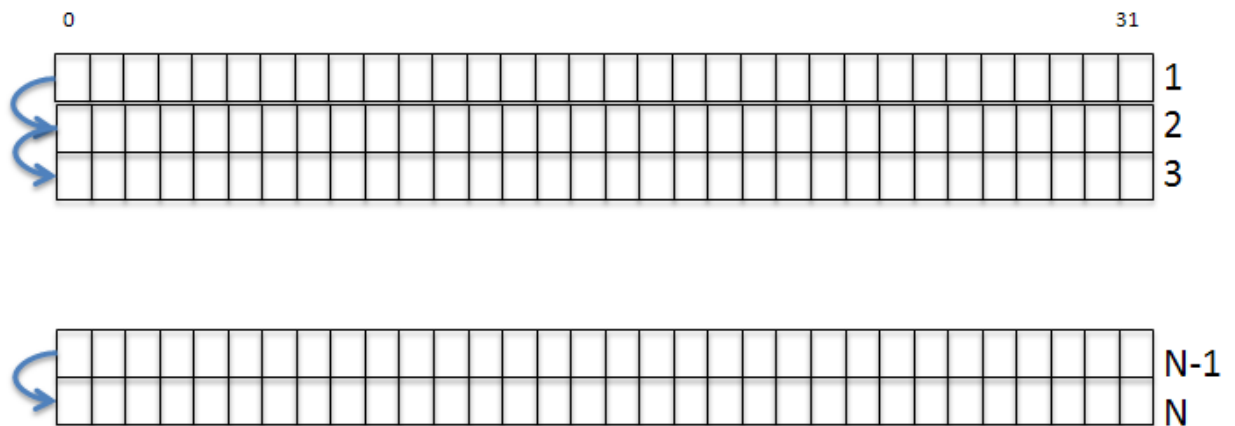


Figure 6.8: Accessing an Integer Array with a Stride of One Cache Line

We predicted that this microbenchmark's L2-cache utilization is 99.79%, i.e., $U_{L2Cache} = 99.79\%$. For our implementation of the microbenchmark, we verified that no loop execution cycles for the last max-1 iterations are associated with instruction cache and TLB misses, and FPU and FXU utilization. In addition to counting the performance counter events needed to calculate $U_{L2Cache}$, using pmcount and the following events to measure load on the caches and TLBs: LD_REF_L1, ST_REF_L1, LSU_DERAT_MISS, DTLB_MISS, DATA_FROM_L3, DATA_FROM_LMEM, DATA_FROM_RMEM, INST_CMPL, ITLB_MISS, INST_FROM_L1, INST_FROM_L2, INST_FROM_L3, INST_FROM_LMEM, and INST_FROM_RMEM. These performance event counts indicate the following:

- Of the $4.9 * 10^{11}$ instructions fetched, 99.99% were fetched from the L1 instruction cache, 0.0005% were fetched from the L2, unified cache, and a negligible amount were fetched from the L3 cache. There were no instructions fetched from main memory. This indicates that a minimal amount of the program's execution cycles are associated with L1 instruction cache misses.
- With respect to address translation, there was a 1% miss rate for the L1 ERAT and out of these ERAT misses none missed the TLB.
- The microbenchmark has only 0.5% utilization of the FXUs and 0% utilization of the FPUs. Thus, it puts very little stress on the functional units and, as a result, there should be very little contention for these units.

To validate the method used to measure L2-cache utilization, two instances of the microbenchmark were executed concurrently on the two hardware threads of one POWER5 processor core with the same hardware thread priority in SMT mode. Since equal priorities cause the processor to allocate 50% of the decode cycles to each thread, if the threads execute the same set of instructions, then we expect each thread to get a 50% share of the L2 cache. We expected that L2-cache contention would have a significant impact on the co-schedule's execution time but, unfortunately, we could not predict

the expected execution time because it depends on the L2 cache-miss resolution sites – the more deep in the memory hierarchy an L2-cache miss is serviced, the more severe the penalty.

The experiment shows that the total number of cycles (RUN_CYC) required to execute the co-schedule, i.e., two instances of the L2 cache-stress microbenchmark, is 150% of that required for the execution of the benchmark in single-threaded mode. 50% does present a significant increase in execution time and, thus, this indicates that the proposed method to measure L2-cache utilization is valid. Since we did not predict the exact increment in time due to the variability of the miss resolution site, it does not necessarily mean it is the most accurate realization of the L2 cache utilization metric.

Validation of TLB Metric:

In order to stress the TLB, the main loop of the TLB-stress microbenchmark, depicted in Figure 6.9, accesses a one-dimensional integer array with a stride such that all data address translations miss the d-ERAT. To implement this benchmark the array size and the access stride must ensure that TLB utilization is maximized and that utilization of all other microarchitectural resources are minimized. To implement this benchmark, the following steps can be followed:

1. Initialize the array access stride, *stride_TLB*, to one page. As shown in Figure 6.9, during each iteration of the main loop of the microbenchmark, the array is accessed starting with the first element with a stride equal to *stride_TLB*.
2. Create an integer array of size equal to $2 * (\text{Num_Page_Entries in d-ERAT}) * \text{Page_Size}$, i.e., $\text{array_size} = 2 * (\text{Num_Page_Entries in d-ERAT}) * \text{Page_Size}$. To maximize TLB utilization and to ensure that during a loop iteration all d-ERAT accesses are misses, select an array size such that during an iteration all d-ERAT entries are evicted at least once and the full capacity of the TLB is utilized.
3. Execute an iteration of the outer *for* loop and count the number of d-ERAT misses generated. If the iteration generates 100% d-ERAT misses, then go to step 5. As shown in Figure 6.8, an iteration of the inner *while* loop accesses the array, starting with the first

element, with a stride equal to *stride_TLB*. A d-ERAT miss is expected for every array access since each access references a different page.

4. Empirically adjust the values of *stride_TLB* and *array_size*, and go to step 3. The presence of hardware pre-fetching may reduce the number of expected d-ERAT misses and, hence, it may be necessary to adjust the values of *stride_TLB* and *array_size* to achieve 100% TLB utilization.
5. Set the number of loop iterations, *max*. The number of loop iterations should be set so that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 instruction cache misses for the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the loop, i.e., the satisfaction of TLB misses, for subsequent iterations.

```

/***** MAIN LOOP BODY *****/
For(j=0;j<max;j++)
    While not end of array do:
        one memory access at distance of stride_TLB
    done

```

Figure 6.9: Main Loop of TLB-Stress Microbenchmark

Next, we implemented the TLB-stress microbenchmark on the POWER5; the source code is provided in Appendix A. The implementation is discussed below:

1. *Page_Size* was set to 4KB, the page size in our experimental environment. Since 1,024 integers can be stored in a 4KB page, *stride_TLB* is 1,024.
2. *Array_size* was set to 4,096KB (1,024 4KB pages) because the POWER5 can store 1,024 entries.
3. The execution time of one loop iteration is 10 milliseconds. The number of iterations, *max*, was set to 400,000, which results in an execution time of 100 seconds. In this way, the execution time is dominated by the execution time associated with the last *max*-1 iterations.

We predicted that this microbenchmark's TLB utilization is 98.6%, i.e., $U_{TLB} = 98.6\%$. For our implementation of the microbenchmark, we verified that no loop execution cycles for the last max-1 iterations are associated with instruction cache and FPU and FXU utilization. In addition to counting the performance counter events needed to calculate U_{TLB} , using pmcount and the following events to measure load on the caches and TLBs: LD_REF_L1, ST_REF_L1, LSU_DERAT_MISS, DTLB_MISS, DATA_FROM_L3, DATA_FROM_LMEM, DATA_FROM_RMEM, INST_CMPL, ITLB_MISS, INST_FROM_L1, INST_FROM_L2, INST_FROM_L3, INST_FROM_LMEM, and INST_FROM_RMEM. These performance event counts indicate the following:

- Of the $4.9 * 10^{11}$ instructions fetched, 99.99% were fetched from the L1 instruction cache, 0.002% were fetched from the L2, unified cache, and a negligible amount were fetched from the L3 cache. There were no instructions fetched from main memory. This indicates that a minimal amount of the program's execution cycles are associated with L1 instruction cache misses.
- This microbenchmark stores to array locations and, hence, there is pressure on the caches for data. The L1 data cache has a 99.9% miss rate and nearly 99.9% of those misses are resolved in the L3 cache. A negligible fraction of the misses are resolved in the L2 cache and main memory.
- The benchmark utilizes only 1% of the FXUs and 0% of the FPUs. Thus, it puts very little stress on the functional units and, as a result, there should be very little contention for these units.

To validate the method used to measure TLB utilization, two instances of the microbenchmark were executed concurrently on the two hardware threads of one POWER5 processor core with the same hardware thread priority in SMT mode. Since equal priorities cause the processor to allocate 50% of the decode cycles to each thread, if the threads execute the same set of instructions, then we expect each thread to get a 50% share of the TLB. We expected that TLB contention would have a significant impact

on the co-schedule's execution time but, unfortunately, we could not predict the expected execution time because it depends on the TLB-miss resolution sites – the more deep in the memory hierarchy a TLB miss is serviced, the more severe the penalty.

The experiment shows that the total number of cycles (RUN_CYC) required to execute the co-schedule, i.e., two instances of the TLB-stress microbenchmark, is twice that required for the execution of the benchmark in single-threaded mode. 100% does present a significant increase in execution time and, thus, this indicates that the proposed method to measure TLB utilization is valid. Since we did not predict the exact increment in time due to the variability of the miss resolution site, it does not necessarily mean it is the most accurate realization of the TLB utilization metric.

6.2.3 Step 3: Determination of the Interval Length

In general, the smallest possible interval length will provide the most accurate characterization, the smaller the interval, the greater is the accuracy of resource utilization characterization. However, the perturbation associated with monitoring performance must be taken into account. This perturbation, which is caused by events such as cache and TLB pollution, may change application behavior and increase application execution time.

As described in Section 5.2, the goal of this step is to identify the smallest interval length such that there is less than or equal to an average of 1% perturbation of execution time of the signature-generating applications. To determine the interval length, we experimented with different interval lengths using performance counters to quantify the effect of this perturbation on the execution times of the signature-generating applications used in this implementation.

The time interval length cannot be bigger than the execution time of the shortest running signature-generating application. As shown in Appendix E, some of the PETSc KSP solvers had execution times of one second and, hence, the maximum interval length considered for monitoring was set to one second. In addition, the minimum interval length is restricted by the performance counter tools. For our tools, the smallest interval that can be used is restricted to the length of the CPU scheduling quantum allocated to a task by the operating system. In our experimental environment this

value is 0.01 second and, hence, the minimum interval length is set to 0.01 second. Given the maximum and minimum interval lengths, our goal was to find interval lengths within this range that could be used to profile applications but experience less than or equal to a 1% increase in execution times. If all interval lengths resulted in greater than 1% perturbation, the one that resulted in the smallest average perturbation would be selected. The interval lengths evaluated were selected by first choosing an interval length of one second and then selecting the others so that each subsequent choice represented a decrease in the previously selected interval length by a factor of 10, up to 0.01 second. Thus, we experimented with three different interval lengths: 1 second, 0.1 second, and 0.01 second. We did not try different decrements, such as reducing the interval length by half instead of by a factor of 10; we may do so in future implementations.

First, we count the total number of cycles required to execute each signature-generating application. To accomplish this, we use hardware performance counters to monitor the RUN_CYC event using counter group 5, which counts total run cycles. The counters are started at the beginning of application execution and stopped when the application finishes execution. At the end of application execution the total number of cycles required to execute the entire application is reported. This gives us the base-case execution time of each application. Note, however, that the run time does include the perturbation associated with counting total run cycles, which cannot be avoided.

To capture the execution time of each application with profiling, we count the total cycles required to run the program when monitored at one of the three interval lengths. Here too RUN_CYC is used to capture application execution time. To count the number of cycles required to execute an application at a specific interval length, the counters are programmed to report RUN_CYC counts at the specified interval length. For example, for an interval length of one second, every second the counters are stopped and the counts for this one-second interval are reported; then the counters are restarted for the next interval. Note that restarting the counters resets RUN_CYC to zero and, hence, the counts are not cumulative. At the end of application execution we sum the RUN_CYC counts of all the intervals to obtain the total cycles required to execute the application.

To calculate the execution-time perturbation for each application, we denote as the base-case execution time in number of cycles of the application as $\text{Total_Cyc}_{\text{Base}}$ and the execution times with intervals of 1 second, .1 second, and .01 second in number of cycles as $\text{Total_Cyc}_{T=1}$, $\text{Total_Cyc}_{T=0.1}$, and $\text{Total_Cyc}_{T=0.01}$, respectively. Next, for each application we compute the following:

- $\text{Difference}_{T=1} = (| \text{Total_Cyc}_{T=1} - \text{Total_Cyc}_{\text{Base}} | / \text{Total_Cyc}_{\text{Base}}) * 100$
- $\text{Difference}_{T=0.1} = (| \text{Total_Cyc}_{T=0.1} - \text{Total_Cyc}_{\text{Base}} | / \text{Total_Cyc}_{\text{Base}}) * 100$
- $\text{Difference}_{T=0.01} = (| \text{Total_Cyc}_{T=0.01} - \text{Total_Cyc}_{\text{Base}} | / \text{Total_Cyc}_{\text{Base}}) * 100$

In our experiments we profiled all signature-generating applications using the three interval lengths and calculated $\text{Difference}_{T=1}$, $\text{Difference}_{T=0.1}$, and $\text{Difference}_{T=0.01}$. The data, provided in Appendix E, shows that none of the chosen interval lengths resulted in perturbation of less than or equal to 1% for all applications. Hence, we selected the interval length that gave the least average perturbation across all signature-generating applications. From the data shown in Appendix E, the average $\text{Difference}_{T=1} = 1.26\%$, $\text{Difference}_{T=0.1} = 11.13\%$, and $\text{Difference}_{T=0.01} = 7.7\%$. Hence, we fixed the interval length to one second.

6.2.4 Step 4: Determination of Resource Utilization Levels

Now that we have identified the POWER5's critical resources, i.e., floating-point unit, fixed-point unit, L2 cache, and TLB, and its interval length, next we define the utilization levels of the four critical resources. To do this, we first measure, using the methods described in Section 6.2.1, the utilization of these resources by the signature-generating applications. Since the IBM POWER5 performance-monitoring unit (PMU) restricts the number of hardware counters that can be monitored concurrently to six and the types of events that can be monitored concurrently as a group, each application must be executed six times. This number of executions is required because the methods used to realize the metrics employed six different groups of counter events, only one of which can be counted at a time. The groups employed are 43, 44, 78, 79, 80, and 92. All the groups collect RUN_CYC and instructions completed. The groups that count the events that are used to realize the utilization metrics described in Section 6.2.1 are identified below.

- Group 43 collects LD_MISS_L1 and LD_REF_L1.
- Group 44 collects LSU_DERAT_MISS, ST_REF_L1, and ST_MISS_L2.
- Group 78 collects FPU_FDIV.
- Group 79 collects FPU_FSQRT.
- Group 80 collects FPU0_FIN and FPU1_FIN.
- Group 92 collects FXU0_FIN and FXU1_FIN.

Hence, U_{FPU} is calculated using groups 78, 79, and 80, U_{FXU} is calculated using group 92, and $U_{L2Cache}$ and U_{TLB} is calculated using groups 43 and 44.

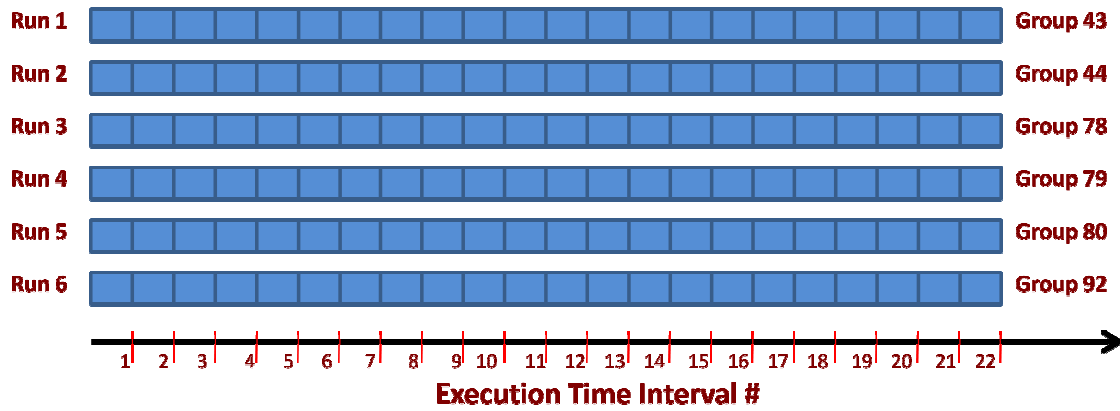


Figure 6.10: Performance Event Groups used to Measure Resource Utilization

As illustrated in Figure 6.10, during each run of an application, a different group of performance events are counted. To count the required performance events for each application, the application is run six times in a row to obtain counts of the six different relevant performance event groups. Another application is not run until all six runs of a given application are completed. This ensures that all but the first run of the application executes with similar states of the cache and TLB, i.e., that left by the previous run of the application. To ensure that first run also gets a similar machine state, before this run we first run the application with performance counter group 5 but without recording performance counter data, thus, warming up the machine state. Group 5 was chosen at random and, in fact, we could have done the warm-up run with any counter group. To obtain resource utilization information for the four critical resources of this implementation during a particular execution time interval, utilization

information for the same execution time interval of the six different runs is captured. For example, resource utilization information for the four critical shareable core resources for execution time interval 1 is obtained using utilization information for execution time interval 1 from the six different runs.

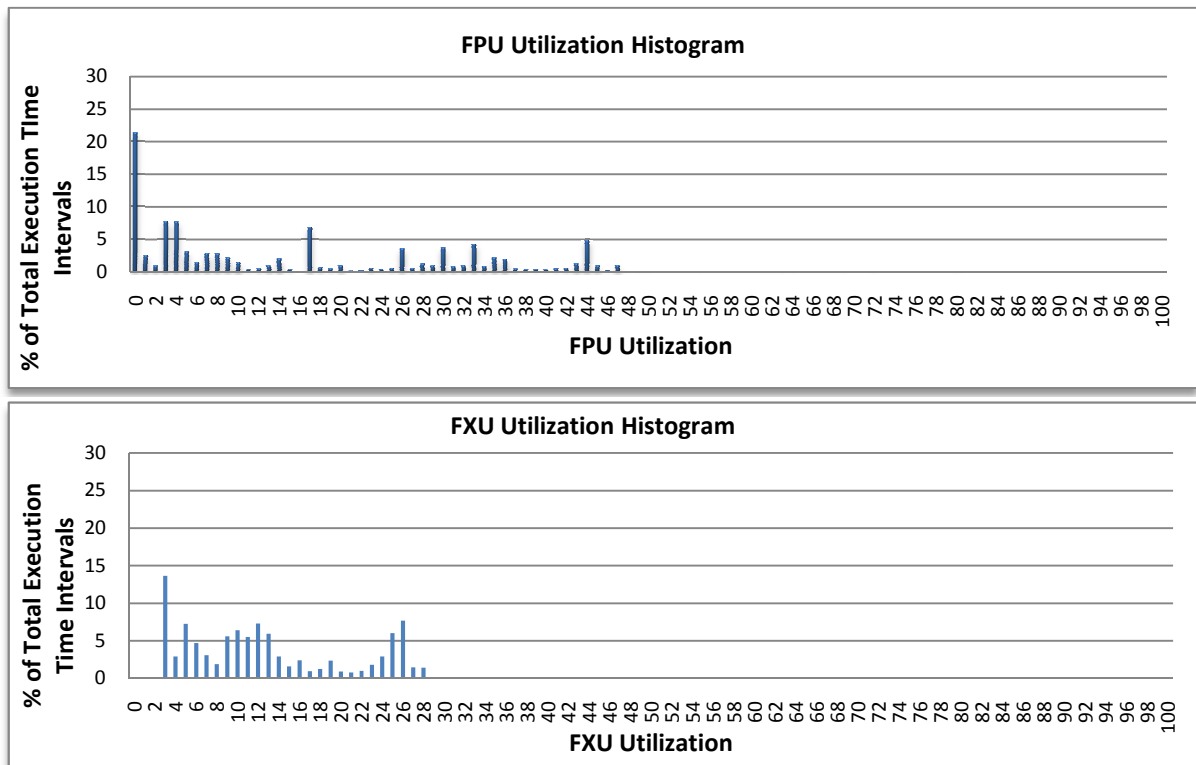
Next, as described in Section 5.2, the resource utilization histograms of each of the four resources are analyzed in order to define the number of utilization levels for each of the four resources. The resultant histogram for each of the four critical resources is shown in Figure 6.11. In this figure, the X-axis represents the utilization of the given resource and the Y-axis represents the percentage of total execution-time intervals. For example, analyzing the graph for TLB utilization in Figure 6.11, it can be seen that 14% of the execution-time intervals have TLB utilization TLB at level one. The minimum resource utilization is 0% for the TLB and the maximum is 85% for the L2 cache.

As described in Section 5.2, given the histograms we next attempt to define utilization levels such that the definition sufficiently differentiates the signature-generating applications belonging to different application classes. Instead of defining levels to cover the utilization range between 0% and 85%, we covered the entire range, i.e., between 0% and 100%. The first attempt defined two levels of utilization for each resource, i.e., $0\% \leq U_{R_i} \leq 50\%$ and $51\% \leq U_{R_i} \leq 100\%$. As shown in Appendix F, in this case, only three distinct signatures were found in the signature-generating applications. Moreover, 99.15% of the execution time intervals had the same signature. Thus, 99% of the execution-time intervals would have the same best priority pair. In contrast, the simulation study, described in Chapter 4, showed that at most 18% of co-schedules got their best throughput at a given priority pair.

Next, the definition of utilization levels was changed to use 10 levels of utilization per resource, such that $0\% \leq U_{R_i} \leq 10\%$ represents one of 10 levels of utilization defined as ranges of utilization, e.g., 1 is associated with utilization between 0% and 10%, 2 with utilization between 11% and 20%, etc; data is shown in Appendix F for this definition. Using 10 levels results in 45 distinct signatures that characterize the signature-generating applications. In this case, it was observed that no more than 15% of total execution-time intervals have the same signature.

Using 10 levels, the distribution of the time intervals of a benchmark suite across the 45 signatures is analyzed. Figure 6.12 presents the distribution data for the four suites. In this figure the

legend is as follows: fp2006 represents SPEC CPU2006 floating-point applications, int2006 represents SPEC CPU2006 integer applications, KSP represents PETSc KSP applications, and NASNPB represents NAS NPB applications. The X-axis represents the signature and the Y-axis represents the percentage of the total execution-time intervals. For example, nearly 66% of the executiontime intervals of the PETSc KSP suite have the signature F1I1C1T1. As shown in this figure, the four benchmark suites have different distributions;. Thus, in this implementation we use 10 levels per resource as defined above, i.e., $0\% \leq \quad \leq 10\%$ represents one of 10 levels of utilization defined as ranges of utilization, e.g., 1 is associated with utilization between 0% and 10%, 2 with utilization between 11% and 20%, etc. The accuracy of our predictions will reflect the efficacy of the choice of utilization levels. As mentioned in Section 5.2, we could have experimented with many different definitions of utilization levels, which may have improved prediction accuracy, however, we did not do this as the implementation is a proof of concept of our methodology.



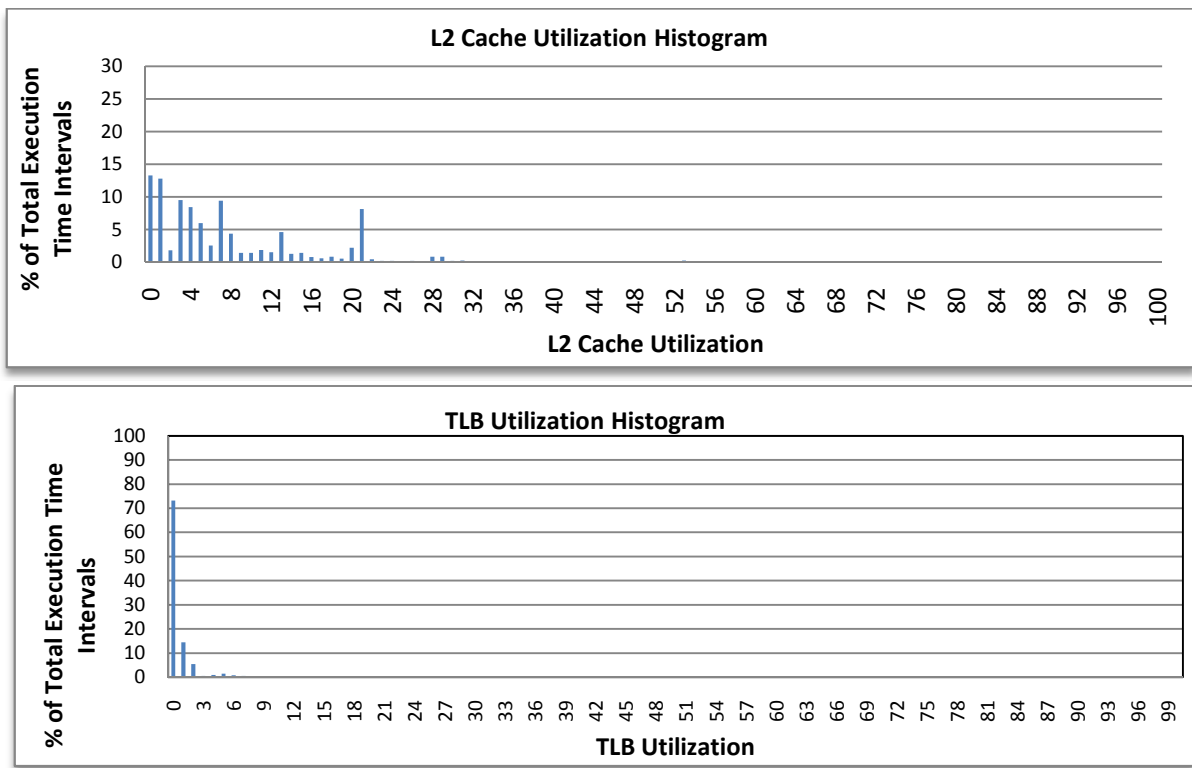


Figure 6.11: Resource Utilization Histogram of Critical Resources

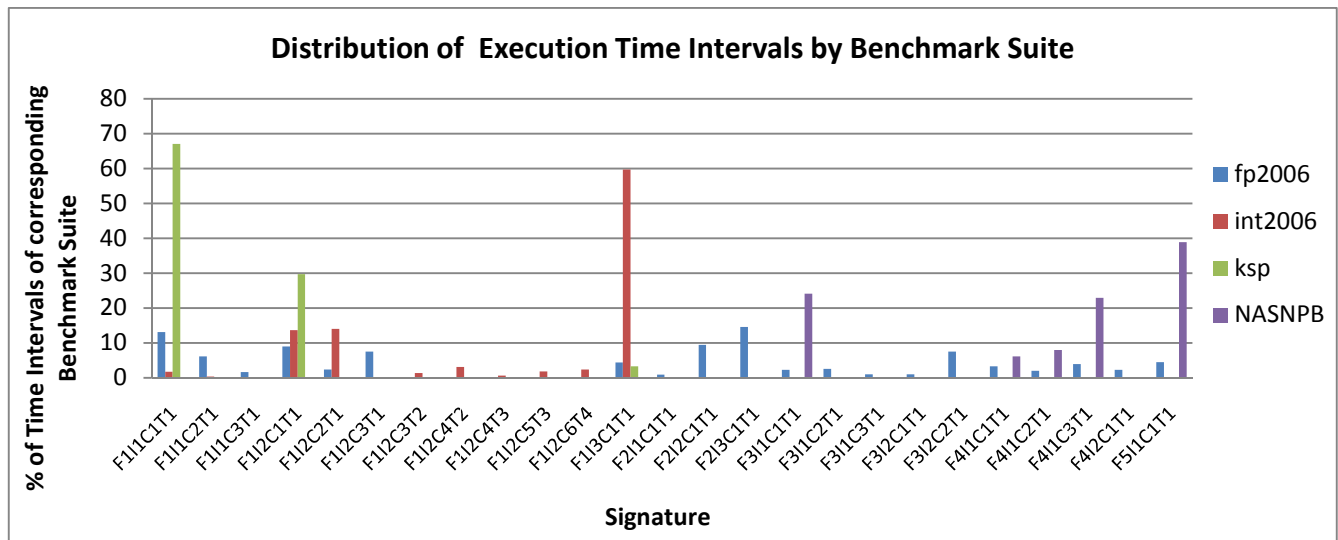


Figure 6.12: Distribution of Execution Time Intervals by Benchmark Suite
Across different Signatures for 10 Levels of Utilization

6.2.5 Step 5: Construction of Signature Database

Having defined the utilization levels, we now choose the representation of a signature. As described in Section 5.2, we define a representation for the four critical resources in our POWER5 implementation. We represent critical resources FPU, FXU, L2 cache, and TLB by symbols F, I, C, and T, respectively.

Given the above representation, if during an execution time interval an application utilizes the four critical resources F, I, C, and T at 28%, 12%, 25%, and 15% of their capacities, respectively, then the signature of the application for the execution time interval is denoted as F3I2C3T2.

To complete step five, described in Section 5.2, we store in a database the time-ordered signature sets of all signature-generating applications using the representation presented above. The number of distinct signatures corresponding to the 10 levels of utilization associated with each of the four critical resources is $(10)^4$ or 10,000. As shown in Table 6.2, 45 out of the 10,000 possible signatures characterize all signature-generating applications. In Table 6.2 the first column lists the name of the signature and the second column lists the percentage of the execution-time intervals of the signature-generating applications with the associated signature. As shown in this table, the ranges of utilization of the four resources are as follows: FPU utilization ranges from 0% to 80%, FXU utilization ranges from 0% to 40%, L2 cache utilization ranges from 0% to 90%, and TLB utilization ranges from levels 0% to 40%.

Table 6.2: Signatures associated with Signature-Generating Applications

Signature	% of Execution-Time Intervals with this Signature
F1I3C1T1	15.74
F1I1C1T1	13.17
F1I2C1T1	10.69
F5I1C1T1	8.98
F2I3C1T1	7.43
F4I1C3T1	5.95
F3I1C1T1	5.33
F2I2C1T1	4.81
F1I2C2T1	4.26
F1I2C3T1	3.85
F3I2C2T1	3.82
F1I1C2T1	3.18
F4I1C1T1	2.77
F4I1C2T1	2.40

F3I1C2T1	1.29
F4I2C1T1	1.15
F1I1C3T1	0.81
F1I2C4T2	0.68
F1I2C6T4	0.52
F3I2C1T1	0.50
F3I1C3T1	0.48
F2I1C1T1	0.47
F1I2C5T3	0.39
F1I2C3T2	0.28
F1I2C4T3	0.13
F6I1C2T1	0.11
F1I3C2T1	0.10
F5I2C2T1	0.10
F1I4C1T1	0.09
F7I1C2T1	0.09
F5I1C2T1	0.07
F1I1C7T4	0.07
F2I1C2T1	0.07
F4I2C2T1	0.07
F2I2C2T1	0.05
F1I2C5T4	0.04
F1I2C6T3	0.02
F1I1C6T4	0.01
F1I1C9T5	0.01
F5I2C1T1	0.01
F7I2C1T1	0.01
F7I2C2T1	0.01
F6I2C1T1	0.00
F8I2C2T1	0.00
F1I2C5T2	0.00

Analyzing the resource utilization of the signature-generating applications, presented in Appendix F, the following observations are made:

- As shown in Appendix F, the NAS NPB3.2 benchmarks have signatures with FPU utilization levels between three and five (20%-50%) and FXU utilization level one (0%-10%). These benchmarks are classified by [52] as floating-point intensive and, hence, the signatures confirm the computational characteristic of this benchmark.
- Data in Appendix F shows that the SPEC CPU2006 integer benchmarks have signatures with FPU utilization level one (0%-10%) and FXU levels between one and three (0%-30%). These benchmarks are classified [38] as integer intensive and our signatures confirm this computational characteristic.

- As shown in Appendix F, the SPEC CPU2006 floating-point benchmarks have signatures with FPU utilization levels between one and eight (0%-80%) and FXU utilization between levels one and four (0%-40%). These benchmarks are classified [38] as floating-point intensive, however, our data, shown in Appendix F, shows that utilization of the FXU is between levels two and three for 66% of the execution time intervals. Hence, for these benchmarks rather than relying on the classification in [38] we use our methods to measure utilization, described in Section 6.2.2, to detect integer and floating-point utilization.
- As shown in Appendix F, the PETSc KSP library routines have FPU utilization level one (0%-10%) and for the FXU, 67% of the execution-time intervals in their signature sets have level-one (0%-10%) utilization, 29.7% have level two (11%-20%), and 3.3% have level three (21%-30%). These benchmarks are classified [53] as floating-point intensive, however, our data, presented in Appendix F, shows that they have FPU utilization level one (0%-10%). Hence, for these benchmarks rather than relying on the classification in [53], we use our methods to measure utilization, described in Section 6.2.2, to detect integer and floating-point utilization.

6.3 PHASE 2: PREDICTION FRAMEWORK DEVELOPMENT

In this phase we generate the prediction framework that will be used to predict the best priority pair of a co-schedule of two signature phases or two applications that each have a dominating signature. In the first step, as explained in Section 5.2, we explain the design of the signature microbenchmarks that are used to generate the best priority pair predictions; this step is described in Section 6.3.1. In Section 6.3.2 we explain the process followed to generate the prediction table using the signature microbenchmarks.

6.3.1 Step 1: Creation of Signature Microbenchmarks

As explained in Section 5.2, the second phase of our methodology generates the prediction table that will be used to predict the best priority pair for a co-schedule of signature phases or two applications

each with a dominating signature. To generate the predictions, we create signature microbenchmarks; we explain the design process below.

Given a signature $F_x I_y C_z T_w$, where x, y, z, t, w are the utilization levels of the FPU, FXU, L2 cache, and TLB, respectively, we create a signature microbenchmark that utilizes, at each execution-time interval, resources F, I, C, and T at levels x, y, z , and w , respectively. To create a microbenchmark that stresses the four resources simultaneously at the desired utilization levels per execution-time interval, we first create microbenchmarks that stress only one of the four resources at the desired utilization level per execution-time interval. Hence, we create four microbenchmarks that each stresses, at every execution-time interval, the FPU at level X utilization, the FXU at level Y utilization, the L2 Cache at level Z utilization, and the TLB at level W utilization, respectively. These microbenchmarks are created by modifying the corresponding resource-stress benchmarks described in Section 6.2.2. Finally, to create the signature microbenchmark $F_x I_y C_z T_w$, we combine the four benchmarks such that the signature microbenchmark stresses the four critical resources F, I, C, and T at levels x, y, z , and w , respectively. Since the combination of the corresponding resource-stress benchmarks may interfere with each other, the signature microbenchmark may have to be fine-tuned in order to be characterized by the desired signature.

Microbenchmark Design for Level X FPU Utilization, F_x :

Figure 6.13 shows the design of a microbenchmark that stresses the FPU at utilization level X . For every loop iteration that runs in n cycles, the microbenchmark should use the FPUs between $(X - 1) / n$ to X / n cycles and stall issues to FPUs between $(10 - X - 1) / n$ to $(10 - X) / n$ cycles. For example, if a loop iteration runs for 10 cycles, the microbenchmark to stress the FPU at level-three utilization (20%-30%) should, for each loop iteration, use FPUs during 2/10 to 3/10 of the cycles and should stall issues for 7/10 to 8/10 of the cycles. To create such a microbenchmark the steps shown below can be followed:

1. Given FPU utilization level X , set initial values of $A = 2 * X$, $B = 2 * (10 - X)$, and $C = 1$.

2. Execute A short-latency independent floating-point instructions. The A instructions will use $A/2$ issue slots on the FPUs.
3. Next execute B dependent instructions that are dependent on a predecessor instruction at a distance of C instructions. The POWER5 FPU pipelines the execution of short latency floating-point instruction, and as a result we expect that B instructions will each add $6 - C$ stall cycles to the FPUs, giving a total of $B * (6 - C)$ cycles with no issue.
4. If FPU utilization level X is achieved go to step 6.
5. Empirically adjust the values of A , B , and C , and repeat steps 2, 3, and 4. The presence of loop control code and branch instructions may add cycles to the execution of the loop body and thus, A , B , and C may have to be tweaked to achieve FPU utilization level X .
6. Fix the number of loop iterations, max . The loop iterations should be fixed such that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 misses for the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the loop for subsequent iterations.

Loop:

A independent short latency FP instructions
 B dependent instruction each one dependent on a predecessor at a distance of C instructions

count++;
branch to loop if count < max

Figure 6.13: Utilization Level X FPU Stress Microbenchmark Main Loop: Fx

Microbenchmark Design for Level Y FXU Utilization, I_Y :

Figure 6.14 shows the design of a microbenchmark that stresses the FXU at utilization level Y . For every loop iteration that runs in n cycles, the microbenchmark should use the FXUs between $(Y - 1) / n$ to Y / n cycles and stall issues to FXUs between $(10 - Y - 1) / n$ to $(10 - Y) / n$ cycles. For example, if a loop iteration runs for 10 cycles, the microbenchmark to stress the FXU with level-three utilization

(20%-30%) should, for each loop iteration, use the FXUs during 2/10 to 3/10 of the cycles and should stall issues for 7 / 10 to 8 / 10 of the cycles. To create such a microbenchmark the steps shown below can be followed:

1. Given FPU utilization level Y , set initial values of $D=2*Y$, and $E=2*(10-Y)$.
2. Execute D independent fixed-point instructions. The D instructions will use $D/2$ issue slots on the FXUs.
3. Next, execute E *noop* instructions. These instructions add $E/2$ bubbles (stalls) on the two FXUs, without changing the machine state.
4. If the loop body achieves level Y FXU utilization, go to step 6.
5. Empirically adjust the values of D and E and go to step 2. The presence of loop control code and branch instructions may add cycles to the execution time of the loop body and, thus, the number of independent, D , and *noop*, E , instructions may to have be adjusted.
6. Fix the number of loop iterations, max . The number of loop iterations should be set such that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 misses associated with the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the last $max-1$ iterations.

Loop:

D independent FX instruction
E *noop* instructions

count++;
branch to loop if count<max

Figure 6.14: Utilization Level Y FXU Stress Microbenchmark Main Loop: I_y

Microbenchmark Design with Level Z L2 Cache Utilization, C_z :

Figure 6.15 shows the main loop of a microbenchmark that stresses the L2 cache at utilization level W . During each loop iteration if the benchmark has n memory accesses, then the microbenchmark's access pattern should result in between $(Z - 1) / n$ to (Z / n) L1 data cache misses and

between $(10 - Z - 1) / n$ to $(10 - Z) / n$ L1 data cache hits, resulting in a $(Z - 1) / n$ to (Z / n) L2 cache utilization. For example, if a loop iteration has 10 memory accesses, the microbenchmarks access pattern should stress the L2 cache with level-three utilization (20%-30%) and for each loop iteration, should result in between 2/10 to 3/10 L1 data cache misses, and between 7/10 to 8/10 L1 data cache hits.

To create such a microbenchmark the steps shown below can be followed:

1. Initialize *stride_Cache* = one cache line, $F = 10 - Z$, and $G = Z$, *array_size* = 2*Capacity of the L1 data cache. As shown in Figure 6.14, in each iteration of the loop, the array is accessed from first element to the last element with F sequential memory accesses to one cache line followed by G memory accesses at distance of *stride_Cache*. In order to have the same L1 data cache miss rate in each iteration, we pick the array size such that during an iteration all L1 data cache lines are evicted at least once.
2. Execute F sequential memory accesses from the same cache line in the L1 data cache. These F memory accesses will result in F L1 data cache hits.
3. Execute G memory accesses at *stride_Cache* distance. These G memory accesses will result in G L1 data cache misses.
4. If the loop body achieves level Z L2 cache utilization, go to step 6.
5. Empirically adjust the values of *stride_Cache*, F , G and *array size* and go to step 2. The presence of hardware pre-fetching may result in reducing the number of expected L1 data cache misses and, hence, we empirically adjust values of *stride_Cache*, F , G and *array size* so as to achieve level Z L2 cache utilization.
6. Fix the number of loop iterations, *max*. The number of loop iterations should be set so that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 misses associated with the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the last *max*-1 iterations.

```

/***** MAIN LOOP BODY *****/
For(j=0;j<max;j++)
    While not end of array do:
        F sequential memory access to the same cache line
        G memory access at distance of stride_Cache
    done

```

Figure 6.15: Utilization Level Z L2 Cache Stress Microbenchmark Main Loop: Cz

Microbenchmark Design with Level W TLB Utilization, T_w :

Figure 6.16 shows the main loop of a microbenchmark that stresses the TLB at utilization level W. During each loop iteration if the benchmark has n memory accesses, then the microbenchmark's access pattern should result in between $(Z - 1) / n$ to Z / n d-ERAT misses and between $(10 - Z - 1) / n$ to $(10 - Z) / n$ L1 d-ERAT hits, resulting in $(Z - 1) / n$ to (Z / n) TLB utilization. For example, if a loop iteration has 10 memory accesses, the microbenchmarks' access pattern should stress the TLB with level-three utilization (20%-30%) and for every loop iteration, should result in between $2 / 10$ to $3 / 10$ d-ERAT misses, and between $7 / 10$ to $8 / 10$ d-ERAT hits. To create such a microbenchmark the steps shown below can be followed:

1. Initialize $stride_TLB$ = one page size, $F = 10 - Z$, and $G = Z$, $array_size = 2 * (\text{Num_Page_Entries in d-ERAT}) * \text{Page_Size}$. As shown in Figure 6.15, during an iteration of the loop, the array is accessed from the first element to the last with M sequential memory accesses to a page followed by N memory accesses at distance of $stride_TLB$. In order to have the same d-ERAT miss rate for each iteration, we pick the array size such that during an iteration all d-ERAT entries are evicted at least once.
2. Execute M sequential memory accesses from the same page. These F memory accesses will result in F d-ERAT hits.
3. Execute N memory accesses at $stride_TLB$ distance. These N memory accesses will result in N d-ERAT misses.

4. If the loop body achieves level W TLB utilization, go to step 6.
5. Empirically adjust the values of *stride_TLB*, *M*, *N* and *array size* and go to step 2. The presence of hardware pre-fetching may result in reducing the number of expected d-ERAT misses and, hence, we empirically adjust values of *stride_TLB*, *M*, *N* and *array size* so as to achieve level W TLB utilization.
6. Fix the number of loop iterations, *max*. The number of loop iterations should be set such that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 misses for the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the last *max*-1 iterations.

```

/***** MAIN LOOP BODY *****/
For(j=0;j<max;j++)
    While not end of array do:
        M sequential memory access to the same page
        N memory access at distance of stride_TLB
    done

```

Figure 6.16: Utilization Level W TLB Stress Microbenchmark Main Loop: Tw

Design of Signature Microbenchmark FxIyCzTw:

Now that we have created the benchmarks that stress the four resources one at a time, the next step is to combine these benchmarks to create the signature microbenchmark FxIyCzTw, which is shown in Figure 6.17. Once the individual stress benchmarks are combined, a few more steps have to be followed to get the desired signature microbenchmark; these steps are shown below:

1. Initialize *stride_Cache*, *stride_TLB*, *A*, *B*, *C*, *D*, *E*, *F*, *G*, *M*, and *N* from the corresponding values of the stress benchmark implementation for each of the four resources.

2. Initialize $array_size = 2 * (\text{Num_Page_Entries in d-ERAT}) * \text{Page_Size}$. As shown in Figure 6.16, in each iteration of the loop, the same array is accessed from first element to end of array with F accesses to one cache line, G memory accesses at distance of $stride_Cache$, M sequential memory accesses to a page followed by N memory accesses at distance of $stride_TLB$. In order to have the same L1 data cache and d-ERAT miss rate in each iteration, we pick the array size such that during an iteration all L1 data cache and d-ERAT entries are evicted at least once.
3. Initialize loop unrolling factor, $P = 1$. Loop unrolling takes into account the number of cycles added to program execution in the program loop. The number of cycles to run the loop effects the FXU and FPU utilization metrics, described in Section 6.2.2, as they are calculated per cycle, whereas the number of memory accesses affects L2 cache and TLB metrics, also described in Section 6.2.2, as they are calculated per memory access.
4. Run the loop and evaluate if the desired Signature is achieved, go the last step.
5. Empirically adjust $stride_Cache$, $stride_TLB$, A , B , C , D , E , F , G , M , N , and P and go to step 4.
6. Fix the number of loop iterations, max . The number of loop iterations should be set such that the time it takes to execute them is sufficiently larger than the time spent servicing compulsory L1 misses for the first loop iteration. Thus, the execution time of the benchmark should be dominated by the execution of the last $max-1$ iterations.
7. If the desired signature is not observed in all execution time intervals of the signature microbenchmark, go to step 4.

```

/***** MAIN LOOP BODY *****/
For(j=0;j<max;j++)
    While not end of array do:
        /*** FPU instructions ***/
        A independent short latency FP instructions
        B dependent instruction each one dependent on a predecessor at a distance of C instructions

        /*** FXU instructions ***/
        D independent FX instruction
        E noop instructions

        The FPU and FXU instructions maybe unrolled P times to account for added cycles of L2 cache
and TLB.

        /*** L2 Cache instructions ***/
        F sequential memory access to the same cache line
        G memory access at distance of stride_Cache

        /*** TLB instructions ***/
        M sequential memory access to the same page
        N memory access at distance of stride_TLB

    done

```

Figure 6.17: Signature Microbenchmark Main Loop: FxIyCzTw

Although we could have implemented the microbenchmarks for all the 45 signatures generated in Phase 1, we restrict our implementations to the set of signatures that are the minimum set of signatures that characterize 95% of total aggregate execution time of the signature-generating applications. We call this set the set of *predictable signatures*. We limited our predictions to these signatures because the time required to create a microbenchmark corresponding to all the identified signatures could be significantly large; in some cases a benchmark took as much as two days to create. Thus, best priority pair prediction is restricted to a co-schedule of signature phases or applications each with a dominating signature, where the signatures are in the set of predictable signatures. We identified the set of predictable signatures as follows:

1. Calculate the total number of distinct signatures in the signature database.
2. For each distinct signature, record its number of total occurrences and compute its frequency of occurrence across the aggregate execution time of the signature-generating applications.
3. Sort the signatures in descending order of their frequency of occurrence; this data is shown in Table 6.2 for our POWER5 implementation.

4. Identify the smallest set of signatures for which their frequency of occurrence adds up to 95% of the aggregate execution time of the set of signature-generating applications. This is the set of predictable signatures; Table 6.3 shows the set of predictable signatures for our POWER5 implementation.

As shown in Table 6.3, 17 signatures were sufficient to characterize 95.6% of the aggregate execution time of signature-generating applications. For each of the 17 predictable signatures we created signature microbenchmarks, which are presented in Appendix B. Using these signature microbenchmarks, we can generate best priority pair predictions for co-schedules comprising components with signatures in the set of predictable signatures, shown in Table 6.3. On the other hand, at this time, the methodology predicts equal priorities co-schedules that do not meet these specifications.

Table 6.3: Predictable Signature Set

Signature	% of Execution Time Intervals with this Signature
F1I3C1T1	15.74
F1I1C1T1	13.17
F1I2C1T1	10.69
F5I1C1T1	8.98
F2I3C1T1	7.43
F4I1C3T1	5.95
F3I1C1T1	5.33
F2I2C1T1	4.81
F1I2C2T1	4.26
F1I2C3T1	3.85
F3I2C2T1	3.82
F1I1C2T1	3.18
F4I1C1T1	2.77
F4I1C2T1	2.40
F3I1C2T1	1.29
F4I2C1T1	1.15
F1I1C3T1	0.81

6.3.2 Step 2: Generate the Prediction Table

Given the 17 signature microbenchmarks corresponding to the 17 predictable signatures identified in the previous step, we ran all possible signature-microbenchmark pairs in SMT mode under the 11 priority pairs. For each of the 289 signature-microbenchmark pairs (X, Y) we analyzed the $IPC_{\text{aggregate}}$ of the 11 priority pairs to determine the best priority pair. In the prediction table, presented in Appendix G for this POWER5 implementation, the best priority pair for microbenchmark pair (X, Y) is stored as the prediction for the best priority pair of a co-schedule of application threads with signatures X and Y.

The prediction table can be used as follows:

1. For each component of the given co-schedule determine its phase or dominating signature. (Again, the signature is captured while executing the co-schedule component in single-threaded mode).
2. Using the signature pair (X, Y) look up in the prediction table the predicted best priority pair. Thus, to use the prediction table both signatures of the pair have to be one of the 17 predictable signatures.
3. If the signature pair is in the table, use the predicted best priority pair for the co-schedule; else, use the equal (default) priority pair.

6.4 PHASE 3: PREDICTION VALIDATION

Given the prediction table, this phase of the methodology assesses the accuracy of our POWER5 implementation. As described in Section 5.4, the third phase of our methodology first determines an execution-time threshold that is used to identify applications with a dominating signature; this step is described in Section 6.4.1. Once this has been done, the second step, presented in Section 6.4.2, identifies target applications that are used to evaluate the prediction accuracy. The results of the evaluation of prediction accuracy are reported in Section 6.4.3.

6.4.1 Step 1: Determination of Execution-Time Threshold for a Dominating Signature

Our validation is restricted to co-schedules comprising applications with signatures that are dominating signatures and are one of the 17 predictable signatures shown in Table 6.3. Since the prediction table was constructed using only 17 signature microbenchmarks, i.e., 17 microbenchmarks each of which have one of these signatures as a dominating signature, we cannot predict for co-schedules comprising applications with different signatures – for these we use equal priorities (default). In future work we discuss how we could extend our methodology to predict for signatures that are not used to create the prediction table.

An application with a dominating signature mirrors an application with a single signature phase. But as the literature indicates [54-56], the execution of a scientific application is comprised of a finite number of phases. Currently we are not able to dynamically adapt priority settings to experiment with such applications, and this is the reason why we validate with applications that have a dominating signature. For co-schedules of applications with dominating signatures that are part of the set of predictable signatures, the predicted best priority pair is set statically at the beginning of the execution of the co-schedule for the entire execution times of the two applications. In order to dynamically apply the predictions for applications with multiple signature phases, we need to implement mechanisms for identifying the beginning and end of signature phases. Note that we are investigating the correlation between phases of execution and signature phases, and hypothesize that a phase of execution corresponds to a phase of a signature. As discussed in Chapter 7, future work includes dynamic priority adaptation as an application changes signature phase.

To identify the threshold for a dominating signature, we empirically evaluate three different threshold values: 75%, 85%, and 95%. For each threshold value, we ran co-schedules comprising target applications and evaluated the prediction accuracy as described in Step 3 of our methodology, presented in Section 5.4. The prediction accuracy data for 75% and 85% are presented in Appendix G. As shown there, these co-schedules when executed with the best priority pair associated with their dominating signatures experience an 8.90% and 6.23% throughput loss, respectively, as compared to executing their sets of intervals with their actual best priority pairs. As shown in Section 6.4.3, a 95% threshold results

in a 2.63% throughput loss. Hence, for this implementation an application has a dominating signature, S_i , if S_i characterizes 95% of its execution time.

6.4.2 Step 2: Identification of Target Applications

We identified the signature-generating applications that have a dominating signature that is in the set of 17 predictable signatures. As shown in Appendix G, of the 17 predictable signatures only eight are dominating signatures, and these eight are the dominating signatures for 34 of the 149 benchmark-input-data combinations. It takes between 24 to 48 hours to run a co-schedule at the 11 priority pairs, hence, to run 598 distinct application co-schedules comprising the 34 applications with dominating signatures it would take between 598 to 1,196 days. Thus, we selected eight of the 34 applications for the validation phase of this implementation.

Table 6.4: Applications used to Evaluate Accuracy of POWER5 Prediction Methodology

Signature	Benchmark Suite	Benchmark	Data Set
F1I1C1T1	PETSc KSP	chebychev	cfid.1.10
F1I2C1T1	PETSc KSP	gmres	arco4
F1I3C1T1	PETSc KSP	lsqr	cfid.1.10
F4I1C2T1	NAS NPB	lu-mz.A	A
F5I1C1T1	NAS NPB	bt-mz.A	A
F5I1C1T1	NAS NPB	bt-mz.B	B
F1I3C1T1	SPEC CPU2006	462.libquantum	ref
F3I2C2T1	SPEC CPU2006	437.leslie3d	ref

Table 6.4 lists the eight applications that were used to evaluate the accuracy of our POWER5 prediction methodology. These applications are from the PETSc KSP, NAS NPB, and SPEC CPU 2006 benchmark suites. Each of the applications has a dominating signature that is part of the set of 17 predictable signatures. For each application, which is named in column 3, column 1 gives its dominating signature, column 2 indicates the benchmark suite to which it belongs, and column 4 specifies its input

data. Using these eight applications we formed and executed co-schedules with applications that belong to the same benchmark suite. We also formed and executed co-schedules of PETSc KSP with two out of the three NAS benchmarks, bt-mz.A, and lu-mz.A. Thus, a total of 21 co-schedules were used to test the accuracy of our predictions. Since each co-schedule takes between 24 to 48 hours to complete, the 21 co-schedules complete in about 40 days. Due to long running times we did not experiment with more co-schedules and, thus, our validation is restricted to the 21 co-schedules.

6.4.3 Step 3: Evaluation of Prediction Accuracy

For each of the 21 co-schedules, $IPC_{\text{aggregate}}$ at the best priority pair, default priority pair (equal priority pair) and the predicted best priority pair are identified. Next, for each co-schedule, the $IPC_{\text{aggregate}}$ attained using the predicted best and default priority pairs are compared to that attained with the best priority pair. In addition, the $IPC_{\text{aggregate}}$ attained using the default priority pair is compared to that attained with the predicted best priority pair. The results are presented in Table 6.5, where the first column lists the application suite pair to which the co-scheduled applications belong, and the second (fourth) and the third (fifth) columns list the first (second) application of the co-schedule and its signature, respectively. The sixth and seventh columns of the table list the best and predicted best priority pairs. The last three columns compare the $IPC_{\text{aggregate}}$ at the predicted, default, and best priority pairs. The following abbreviations are used in these columns.

- PT: $IPC_{\text{aggregate}}$ at the predicted best priority pair
- DT: $IPC_{\text{aggregate}}$ at the default priority pair
- BT: $IPC_{\text{aggregate}}$ at the best priority pair

Using these abbreviations, the values in the last three columns are computed as indicated below:

1) Comparison of $IPC_{\text{aggregate}}$ at default and best priority pairs:

$$((DT - BT) / BT) * 100 \%$$

2) Comparison of $IPC_{\text{aggregate}}$ at predicted and best priority pairs:

$$((PT - BT) / BT) * 100 \%$$

3) Comparison of $IPC_{\text{aggregate}}$ at predicted best and default priority pairs:

$$((PT - DT) / DT) * 100 \%$$

Table 6.5: Prediction Accuracy Results

Benchmark Suite Pair	App_X	Signature of App_X	App_Y	Signature of App_Y	Best Priority Pair (x, y)	Predicted Best Priority Pair (x, y)	$((DT - BT) / BT) * 100 \%$	$((PT - BT) / BT) * 100 \%$	$((PT - DT) / DT) * 100 \%$
KSP, KSP	Chebychev	F111C1T1	chebychev	F111C1T1	(6, 6)	(6, 6)	0.00	0.00	0.00
KSP, KSP	Chebychev	F111C1T1	gmres	F112C1T1	(6, 5)	(6, 6)	-0.74	-0.74	0.00
KSP, KSP	Chebychev	F111C1T1	lsqr	F113C1T1	(6, 6)	(6, 6)	0.00	0.00	0.00
KSP, KSP	Gmres	F112C1T1	gmres	F112C1T1	(6, 5)	(6, 6)	-1.58	-1.58	0.00
KSP, KSP	Gmres	F112C1T1	lsqr	F113C1T1	(6, 6)	(6, 6)	0.00	0.00	0.00
KSP, KSP	Lsqr	F113C1T1	lsqr	F113C1T1	(6, 6)	(6, 6)	0.00	0.00	0.00
SPEC, SPEC	462.libquantum	F113C1T1	462.libquantum	F113C1T1	(6, 6)	(6, 6)	0.00	0.00	0.00
SPEC, SPEC	462.libquantum	F113C1T1	437.leslie3d	F312C2T1	(6, 6)	(6, 5)	0.00	-7.46	-7.46
SPEC, SPEC	437.leslie3d	F312C2T1	437.leslie3d	F312C2T1	(6, 1)	(6, 2)	-6.35	-0.76	5.97
NAS, NAS	bt-mz.A	F511C1T1	bt-mz.A	F511C1T1	(6, 2)	(6, 1)	-12.91	-4.10	10.12
NAS, NAS	bt-mz.A	F511C1T1	lu-mz.A	F411C2T1	(6, 6)	(4, 6)	0.00	-20.05	-20.05
NAS, NAS	bt-mz.A	F511C1T1	bt-mz.B	F511C1T1	(6, 1)	(6, 1)	-13.95	0.00	16.21
NAS, NAS	lu-mz.A	F411C2T1	lu-mz.A	F411C2T1	(6, 1)	(6, 1)	-14.10	0.00	16.42
NAS, NAS	lu-mz.A	F411C2T1	bt-mz.B	F511C1T1	(6, 1)	(6, 4)	-10.05	-3.51	7.27
NAS, NAS	bt-mz.B	F511C1T1	bt-mz.B	F511C1T1	(6, 1)	(6, 1)	-13.24	0.00	15.26
KSP, NAS	Chebychev	F111C1T1	bt-mz.A	F511C1T1	(6, 5)	(6, 1)	-1.09	-0.51	0.59
KSP, NAS	Chebychev	F111C1T1	bt-mz.B	F511C1T1	(6, 1)	(6, 1)	-5.33	0.00	5.63
KSP, NAS	Gmres	F112C1T1	bt-mz.A	F511C1T1	(6, 5)	(6, 6)	-1.75	-1.75	0.00
KSP, NAS	Gmres	F112C1T1	bt-mz.B	F511C1T1	(6, 1)	(6, 6)	-5.13	-5.13	0.00
KSP, NAS	Lsqr	F113C1T1	bt-mz.A	F511C1T1	(6, 1)	(6, 6)	-3.94	-3.94	0.00
KSP, NAS	Lsqr	F113C1T1	bt-mz.B	F511C1T1	(6, 2)	(6, 6)	-5.74	-5.74	0.00

We analyzed the data presented in Table 6.5 to determine the accuracy of this implementation of the priority prediction methodology and make the following observations:

- **Comparison of $IPC_{\text{aggregate}}$ at default and best priorities:** The throughput attained using the default priorities is best for only seven of the 21 co-schedules. The co-schedules that attain best throughput at default priorities are two co-schedules of chebychev paired with itself and lsqr, two co-schedules of lsqr paired with itself and gmres, two co-schedules of 462.libquantum paired with itself and 437.leslie3d, and the co-schedule (bt-mz.A, lu-mz.A). For the remaining 14, the throughput attained using the default priorities is between 0.74% and 14.10% below that attained using the best priority pair. For only five of these 14 co-schedules do the default priorities yield a throughput loss of less than 5%. Furthermore, for four of the co-schedules the throughput loss associated with executing at default priorities

rather than best priorities is between 5% and 10%, and for the other five the loss is greater than 10%. The five co-schedules with a throughput loss greater than 10% are five of the six co-schedules from the benchmark suite pair (NAS, NAS) (see Table 6.5). The outlying co-schedule of this set of six is the pair (bt-mz.A, lu-mz.A), for which the default priorities yield the best throughput.

- **Comparison of $IPC_{\text{aggregate}}$ at predicted best and best priorities:** The throughput attained using the predicted best priorities is best for nine of the 21 co-schedules. For the remaining 12 the throughput at the predicted best priorities is between 0.51% and 20.05% below that obtained using the best priority pair. For only four of these co-schedules do the predicted best priorities yield a throughput loss greater than 5%. Furthermore, only one co-schedule yields a throughput loss greater than 10% – the co-schedule (bt-mz.A, lu-mz.A) experiences a throughput loss of 20.05% as compared to that achieved using the best priority pair – and, as described below, this result is questionable.
- **Comparison of $IPC_{\text{aggregate}}$ at predicted best and default priorities:** The throughput attained using the predicted best priorities is equal to that at default priorities for 11 out of 21 co-schedules. For eight of the remaining 10 co-schedules the predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For only one of these eight co-schedules does the predicted priority pair yield a throughput improvement of less than 5% – the co-schedule (chebychev, bt-mz.A) experiences an improvement of 0.59%. Furthermore, for three of the co-schedules the throughput improvement associated with executing with the predicted priority pair, rather than the default priorities, is between 5% and 10%, and for the other four the improvement is greater than 10%. Only for two of the remaining 10 co-schedules is the throughput achieved by executing with default priorities better than that achieved by executing with the predicted priority pair – co-schedules (462.libquantum, 437.leslie3d) and (bt-mz.A, lu-mz.A) experience throughputs of 7.46% and 20.05%, respectively, lower than that achieved with default priorities. As described below, the performance of these two co-schedules was further

investigated – the performance associated with the latter is questionable and that associated with the former appears to be due to the fact that although 462.libquantum has a dominating signature, it has two other signatures and all three signatures differ in terms of FXU utilization.

Our analysis of the co-schedule (bt-mz.A, lu-mz.A), which experienced 20.05% lower throughput at the predicted best priorities, as compared to the default priorities indicates the following. Originally we had executed co-schedule (bt-mz.A, lu-mz.A) on hardware threads (4, 5), i.e., bt-mz.A on hardware thread 4, and lu-mz.A on hardware thread 5. To determine if the mapping of hardware threads may have any bearing on this result, we also executed lu-mz.A on hardware thread 4 and bt-mz.A on hardware thread 5. In this case, we found that the best priority pair was not (6, 6) but rather (6, 2) and, furthermore, the predicted best priority pair provides 3.56% higher throughput than the default priorities. This result comes as a surprise to us, as the processor architecture specifies that the mapping of applications threads to hardware threads of the same core does not make a difference in terms of performance. Nonetheless, it appears that the processor core may be favoring one thread over the other. Accordingly, our future work includes investigation of the role of application-to-hardware thread mapping in terms of identification of best priority pair.

Analysis of the co-schedule (462.libquantum, 437.leslie3d), which experienced 7.46% lower throughput at predicted priorities, as compared to default priorities indicates that the application 462.libquantum has signature F1I3C1T1 for 98.5% of its execution time. In the last 1.5% of its execution time, the application has two different signatures: F1I2C1T1 and F1I4C1T1. The second application, 437.leslie3d, has one signature for its entire execution time. Only one other application from Table 6.4, lsqr, did not have one signature for its entire execution. In this case, lsqr had one signature for 95% of its execution time: F1I3C1T1, and 5% of its time had signature F1I2C1T1. Upon examining the utilization of the FXU by 462.libquantum and lsqr we observe that: (1) for 462.libquantum, the average utilization of the FXU during its F1I3C1T1 signature phase is 25%, whereas during its F1I2C1T1 and F1I4C1T1 signature phases the average FXU utilization is 15% and 33%, respectively and, (2) for lsqr,

the FXU utilization for its F1I3C1T1 signature phase is 22%, whereas it is 20% for its F1I2C1T1 signature phase. In this case, i.e., for lsqr, FXU utilization does not change significantly when its signature changes. In contrast, the differences in FXU utilization during the three signature phases of 462.libquantum are greater, ranging from 15% to 33%. This may have resulted in throughput loss experienced with predicted priorities. Thus, in retrospect, to better assess the accuracy of our implemented prediction methodology, we should have only considered target applications that have a dominating signature for the entire execution time or that have a dominating signature and additional multiple signatures that do not differ much in terms of FXU utilization.

We also analyzed the results presented in Table 6.5 to determine if there are certain applications that benefit most from the use of predicted best priorities. For seven of the eight co-schedules comprising applications for which the utilization of the floating-point unit exceeds that of the fixed-point unit by 10% or more, the predicted priority pairs, as compared to the default priorities, yield a throughput improvement between 5.97% and 16.42%. The seven co-schedules are: (437.leslie3d, 437.leslie3d), (bt-mz.A, bt-mz.A), (bt-mz.A, bt-mz.B), (lu-mz.A, lu-mz.A), (lu-mz.A, bt-mz.B), (bt-mz.B, bt-mz.B), and (chebychev, bt-mz.B). The outlying co-schedule of this set of eight is the co-schedule (bt-mz.A, lu-mz.A), which experienced a throughput loss of 20.05%, which, as described above, is questionable. This result indicates that the methodology for predicting best priority pairs is most applicable to applications for which floating-point unit utilization dominates that of the fixed-point unit by at least 10%.

6.5 LESSONS LEARNED

In this implementation we developed methodologies for the IBM POWER5 processor that attempt to achieve the goals outlined for each step of the three-phase methodology described in Chapter 5. During the course of this implementation several improvements became evident. Below we list the lessons learned:

- The goal of selecting the interval length is to limit the average perturbation of application execution time to less than or equal to 1%. The bigger the interval length, the smaller is the

perturbation of application execution time. The choice of the maximum interval length that can be evaluated for performance monitoring is limited to the execution time of the shortest running application from the set of signature-generating applications. In our implementation the shortest running application ran for one second. Hence, the maximum interval length we could consider was one second. In our implementation it turned out that the maximum interval length of one second resulted in an average perturbation of 1.25% and was close to our target maximum perturbation. However, this may not always be the case. Hence, the lesson learned from this experience is that in order to use a larger maximum interval length for monitoring, longer running applications should be used to form the set of signature-generating applications.

- The goal of choosing the set of critical resources is to identify the set of shareable core resources that have a significant impact on the throughput of the signature-generating applications. In this implementation we used utilization information gathered during our pilot study, described in Chapter 4. This pilot study is a simulation study that used applications that are in application classes than those represented by our signature-generating applications. Nonetheless, we used the set of critical resources that had significant impact on the throughput of the partial application traces used in the simulation study to characterize the set of signature-generating applications. While it turned out that this set of critical resources led to good prediction accuracy in our implementation, this may not always be the case. Hence, the lesson learned as a result of the implementation of this step is that signature-generating applications should be used to determine critical resources.

6.6 CONCLUSIONS

We showed the potential merit of our best priority prediction methodology that is based on Shareable Resource Signatures. This was done by implementing our methodology for an IBM POWER5 processor and characterizing signatures of SPEC CPU2006, NAS NPB, and PETSc KSP benchmarks. Using 21 co-schedules of applications chosen from the above benchmarks, our predictions show that for

nine of the 21 co-schedules predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For 11 co-schedules both the default and predicted priorities yield equal throughput. For only one co-schedule throughput at predicted priorities is 7.46% lower than that achieved with default priorities.

The comparison of throughputs achieved using the predicted best and default priority pairs shows that on the IBM POWER5 it is beneficial to use hardware thread priority settings to improve the throughput of the SMT core for applications that have utilization of the floating-point unit that is at least 10% higher than that of the fixed-point unit. For eight such application pairs, throughput improvements are between 0.59% and 16.42%.

Although our implementation shows merit, it may be improved in several ways, and these are explored in future work in the next chapter.

7 Conclusions and Future Work

Contention for shared processor core resources among applications executing on the hardware threads of an SMT processor can limit achievable throughput. Hardware thread priorities can be used to reduce this resource contention and, thus, potentially improve throughput. However, given an SMT processor with multiple hardware threads, the best priority tuple, i.e., the priorities for a given co-schedule that yields the best throughput, depends on the characteristics of the co-scheduled application threads (henceforth called applications).

To characterize applications for this purpose, we propose a concept called Shareable Resource Signature (signature) corresponding to each specified interval length of the execution of the application in single-threaded mode. This characterization generates a time-ordered signature set for each application. Assume a signature pair that characterizes either a co-schedule of signature phases of two applications or the majority of the execution times of two applications. In these cases, we propose a methodology that predicts the best priority pair. Our methodology can be used only for processors that allow software to set hardware thread priorities. Hence, at present it can be used only on the IBM POWER5 and IBM POWER6 processors.

We presented an implementation of this methodology for the IBM POWER5 processor and demonstrated the efficacy of our methodology to improve throughput for applications with a dominating signature. We showed that, for the IBM POWER5, non-default hardware thread priority pairs should be used, in particular, to improve throughput of applications that have utilization of the floating-point unit 10% or higher than that of the fixed point unit. The proposed methodology makes significant contributions towards application characterization that prove useful in terms of determining hardware thread priorities that can enhance processor core throughput. The contributions made in this dissertation are described in Section 7.1, and Section 7.2 describes future work.

7.1 CONTRIBUTIONS

We presented a three-phase methodology to predict the best priority pair for a co-schedule executed on an SMT processor that supports software-controlled hardware thread priorities, and

implemented and evaluated the methodology using an IBM POWER5 processor. The research questions that were raised and answered in this dissertation are as follows:

1. Can judicious setting of hardware thread priorities be used to improve the throughput of an SMT processor?
2. How can application characteristics be used to predict the best priority pair for a given co-schedule comprising two applications?

While answering these questions, this dissertation made the following major contributions:

2. Demonstrated that the judicious setting of hardware thread priorities can be used to improve SMT processor throughput.
 - a. Using an IBM POWER5 simulator and 263 partial application trace-pairs, we showed that the default priority pair (equal priorities) does not provide the best throughput for 82% of the application trace-pairs.
 - b. For the application trace-pairs studied, the improvement in throughput, as measured by $IPC_{\text{aggregate}}$, achieved by the best priority pair, as compared to the default priority pair, is between 0.00% and 25.52%. The worst-case throughput differs from the best case by as little as 0.42% and as much as 51.09%. Thus, the choice of priority pair can have a significant impact on throughput and the wrong choice can lead to significant throughput loss.
3. Developed the concept of a Shareable Resource Signature. For an application executing in single threaded mode and for a specified execution time interval, the shareable resource signature characterizes the application's utilization of critical shareable core resources.
4. Developed a three-phase methodology that, given a signature pair that characterizes either a co-schedule of signature phases of two applications or the majority of the execution times of two applications, predicts the best priority pair for the co-schedule.

5. Implemented the methodology for the an IBM POWER5 processor, which shows the following:
 - a. 17 of 10,000 possible signatures are sufficient to characterize 95.6% of the execution times of a set of applications that consists of 20 SPEC CPU2006 benchmarks (1 data input), three NAS NPB benchmarks (3 data inputs), and 10 PETSc KSP solvers (12 data inputs). The cgs and lsqr PETSc KSP solvers have signatures that are independent of input data, while one of three NAS NPB benchmark (bt-mz) has a signature that is independent of the input data.
 - b. For 21 co-schedules of applications, each with a signature that characterizes 95% of its execution time, our validation study shows the following:
 - i. *Predicted best priorities yield higher throughput than default priorities for all but one of the 21 co-schedules.* Initial results showed that two co-schedules (462.libquantum, 437.leslie3d) and (bt-mz.A, lu-mz.A) experience a throughput loss of 7.46% and 20.05%, respectively, at predicted priorities, as compared to that achieved at default priorities. Further investigation shows that for the co-schedule (bt-mz.A, lu-mz.A) mapping and executing the co-schedule with the predicted best priorities on hardware threads (5, 4), instead of (4, 5), results in a 3.56% higher throughput as compared to default priorities – this is in contrast to the 20.05% throughput loss experienced when executed on hardware threads (4, 5). Although we have not verified it, one possible reason for this is that the processor core favors one hardware thread over the other. Re-executing the co-schedule (462.libquantum, 437.leslie3d) on hardware threads (5,4), instead of (4, 5), results in predicted priorities yielding lower throughput than the default priorities. Thus, we claim that predicted best priorities yield equal or higher throughput than default priorities for 20 of the 21 co-schedules studied, and for the outlier the throughput loss is 7.46%.

- ii. *Using non-default priorities improves throughput.* The default priority pair yields best throughput for only six of the 21 co-schedules. For the remaining 15 the default priority pair yields throughput that is between 0.74% and 14.10% lower than that achieved with the best priority pair.
- iii. *Using the predicted best priority pair, rather than default priorities, improves throughput or at least provides throughput equal to that achieved with default priorities.* For 11 of the 21 co-schedules both the default and predicted priorities yield equal throughput. For nine of the 21 predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For two of these nine co-schedules the predicted priority pair yields a throughput improvement of less than 5%. Furthermore, for three the throughput improvement associated with executing with the predicted priority pair, rather than default priorities, is between 5% and 10% and for the other four the improvement is greater than 10%.
- iv. *Using predicted best priority pairs appears to be most applicable to floating-point “intensive” applications:* For eight co-schedules comprising applications for which the utilization of the floating-point unit exceeds that of the fixed-point unit by 10% or more, the predicted priority pairs, as compared to the default priorities, yield a throughput improvement between 3.56% and 16.42%. This result indicates that the methodology for predicting best priority pairs is most applicable to applications for which floating-point unit utilization dominates that of the fixed point unit by at least 10%.

7.2 FUTURE WORK

While this dissertation provides a methodology that can be used to set hardware thread priorities to improve the core throughput of SMT processors with software-controlled hardware thread priorities, the methodology can be improved and extended in several ways.

- Our current prediction methodology is limited to applications that have a time-ordered signature set comprising signatures in the set of signatures that characterize the signature-generating applications. In Section 7.2.1 we describe an extension of our methodology to predict best priority pairs for applications with signatures not included in the set.
- Our implementation for the IBM POWER5 was restricted to applications with a dominating signature. Using these applications, we could statically set priorities at the beginning of the execution of a co-schedule and keep them constant for the entire execution times of the two applications. However, as noted in Appendix G, in our implementation, 114 out of 149 application-input-data combinations did not have dominating signatures. For such applications we need to extend our methodology to dynamically adapt priorities when the co-scheduled signature phases change; this extension is discussed in Section 7.2.2.
- For applications with signature phases, a signature phase may correlate to a phase of execution. If this is true, then signatures can be used to detect phases of execution; this is discussed in Section 7.2.3.
- To validate resource utilization metrics and generate the prediction table, we create resource-stress and signature microbenchmarks by hand and empirically tune them to get the desired utilization levels. Thus, to improve productivity, in Section 7.2.4, we discuss an extension of this research to generate the microbenchmarks automatically.
- Signatures also may be used forming co-schedules of application threads. Mapping co-schedules of application threads to different cores based on their signatures can potentially improve throughput. This is described in Section 7.2.5.
- This dissertation seeks to improve $IPC_{\text{aggregate}}$. However, our methodology could be extended to study other metrics such as fairness and power consumption; this is discussed in Section 7.2.6.
- SpecCPU2006 benchmarks have been clustered according to their similarity by [64]. It would be interesting to see if the signatures for the SPEC CPU2006 benchmarks used in

this implementation are sufficient to characterize the clusters identified in [64]; this is discussed in Section 7.2.7.

These future research directions are discussed in this section.

7.2.1 Best Priority Pair Prediction for Co-Schedules Characterized by “Other” Signatures

The methodology we presented in this dissertation is limited to the prediction of the best priority pairs for co-schedules comprising applications that have a time-ordered signature set of signatures in the set of signatures that characterize the signature-generating applications. In order to predict for applications with signatures not in this set a prediction model is needed. There are numerous methods that we could explore to construct such a model. Below, using neural networks and regression analysis as examples, we describe how we might go about constructing such a model and how it could be used for prediction.

- A *neural network* is an artificial intelligence technique that uses a learning approach to model data and generate predictions. A set of independent variables are input to the network, which generates a prediction and computes the difference between the expected output and the value generated by the neural network. The accuracy of the prediction is then fed back to the neural network to adjust its prediction function and reduce prediction error. Once a neural network is trained properly with signature pairs in the initial set, then it could be used to predict for a signature pairs not in the set.
- *Regression analysis* is a technique that generates an analytical expression relating a dependent variable with one or more independent variables. The analytical expression generated is parameterized by the values of the independent variables to produce a value for the dependent variable. Regression analysis could be used to create an analytical expression parameterized by the resource utilization levels of the signature pairs in the initial set, i.e., those that characterize the signature-generating applications, that could predict the best priority pairs for co-schedules characterized by these signature pairs.

Once the expression has been generated, it can be used to predict for co-schedules characterized by signature pairs that were not used to develop the expression.

7.2.2 Dynamic Adaptation of Priorities for Applications with Signature Phases

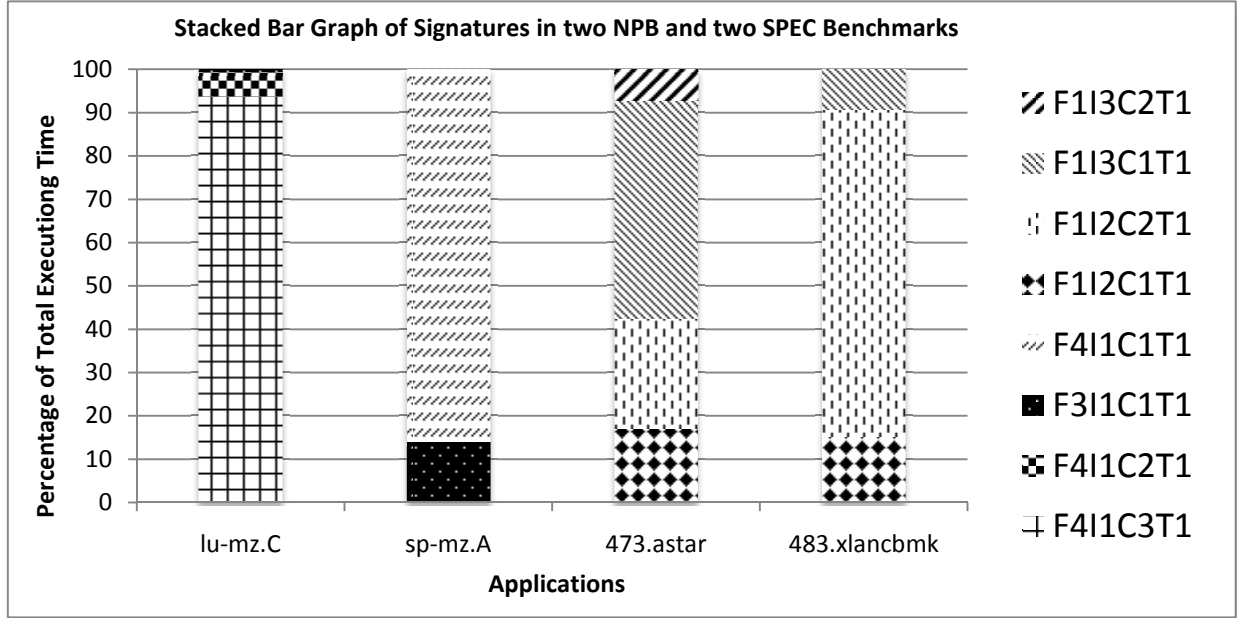


Figure 7.1: Stacked Bar Graph of Signatures found in two NAS and two SPEC Benchmarks

In this dissertation we had to restrict our validation study to those applications that have a dominating signature. As shown in Appendix G, 114 out of the 149 application-input-data combinations in our set of signature-generating applications did not have a dominating signature. To use our best priority pair prediction methodology for applications without a dominating signature, mechanisms must be developed to detect changes in signature phases and adapt the hardware thread priorities to the associated predicted best priority pair. For example, Figure 7.1 shows the stacked bar graph of the signature sets of two NAS and two SPEC benchmarks. In this figure, the X-axis represents the applications and the Y-axis represents the percentage of execution time a signature was observed for a given application. The legend shows the signature corresponding to the patterned bars in the graph. For

example, the sp-mz.A application has signatures F3I1C1T1 and F4I1C1T1 for 14% and 86% of its execution time, respectively.

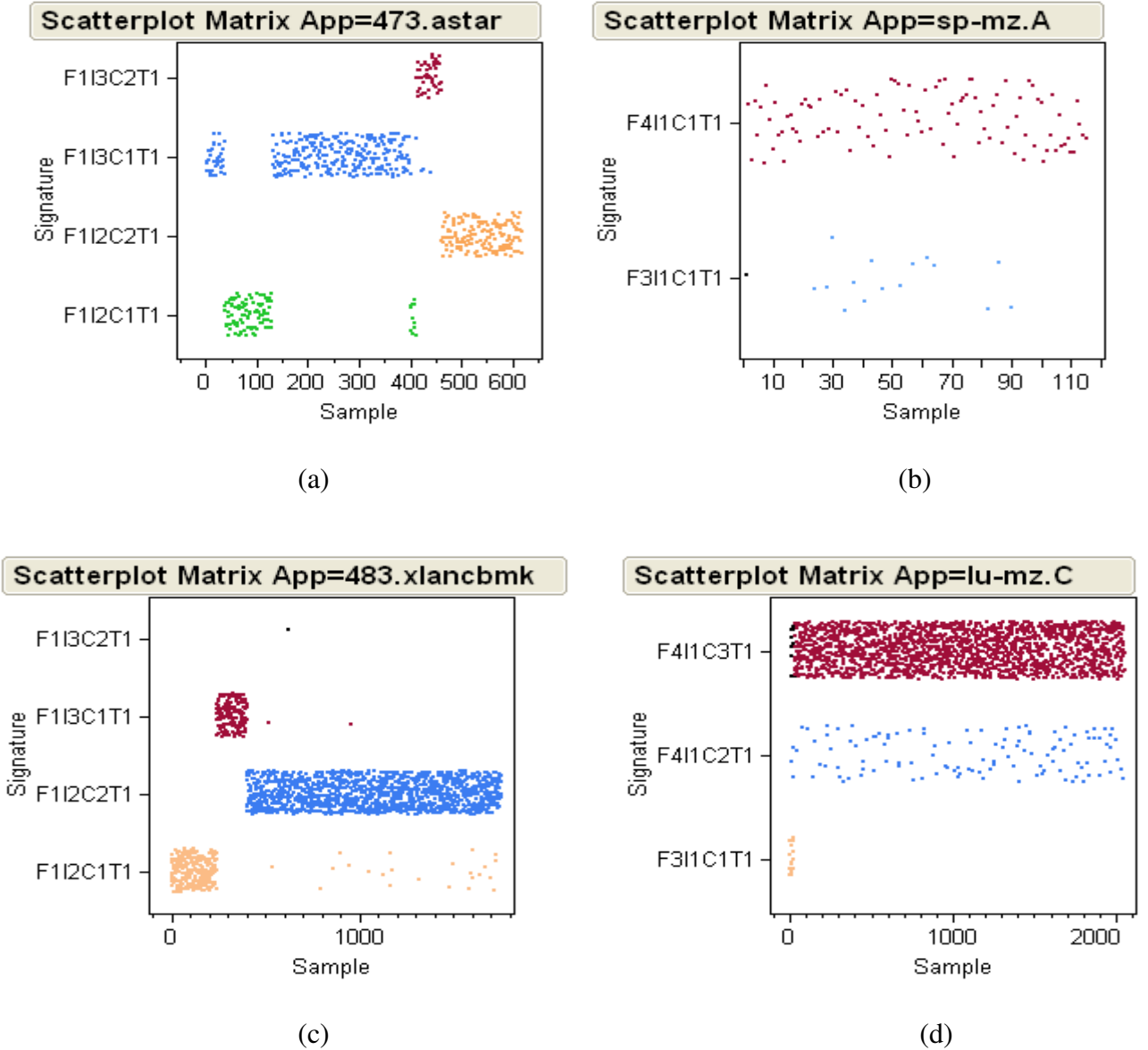


Figure 7.2: Scatter-plot of Signatures found in 2 SPEC CPU2006 and NAS NPB Benchmarks

The signatures can appear in distinct application execution phases or can be interleaved during an application's execution. For example, Figure 7.2 shows a scatter plot of the signatures of two SPEC CPU2006 in plots (a) and (c), and two NAS NPB benchmarks in plots (b) and (d) over time. In this figure, the X-axis represents execution time and the Y-axis represents signatures. As shown in Figure

7.2, the signatures of NAS NPB are interleaved and, thus, change frequently, whereas for the two SPEC CPU2006 benchmarks the signatures appear in phases.

Accordingly, for application pairs with signature phases that are shorter than the length of a dominating signature, the best priority pair may change when an application changes to a different signature phase. For such application pairs, we need to implement mechanisms for demarcating the signature phases of the applications and identifying the current signature pair. In this way, the appropriate predicted best priority pair can be used to set the priorities for each signature phase. Such an implementation must take into account the overhead associated with detecting phase changes as well as the cost associated with adapting the hardware thread priorities. On the IBM POWER5, it takes approximately one millisecond to set the hardware thread priority. Thus, the minimum phase length has to be significantly greater than one millisecond. One possible way to detect phase changes is to insert special instructions in the application code that announces signature phase changes and implement a run-time system that uses these special instructions to determine when and how to adapt hardware thread priorities. The implementation cost will determine the minimum phase length that can be used for dynamic priority adaptation; for shorter phases equal priorities (default) can be used.

7.2.3 Correlation of Application Phases and Signature Phases

Applications often go through different phases of execution. To detect application phases, phase detection techniques use metrics such as basic block vectors, data locality, working set size, and IPC [59-62]. Based on changes in the monitored metrics, phase detection techniques characterize an application's execution phases and carry out adaptations to optimize performance [54, 55].

In this dissertation we characterize applications using signature phases. Hence, a study that evaluates the correspondence of a signature phase to an execution phase would answer the following question: Can a shareable resource signature be used to detect an application's execution phases?

7.2.4 Automatic Benchmark Generation

To validate the methods used to realize the utilization metrics used in our POWER5 implementation, we created resource-stress microbenchmarks to test the accuracy of the associated metrics. These benchmarks are generated by hand and empirically fine-tuned. The signature microbenchmarks were created in a similar fashion, i.e., by combining the resource-stress benchmarks and empirically fine-tuning them until the desired signature was observed. To apply the methodology to a different architecture, the benchmarks for that architecture would have to be generated in the same way. Hence, automatic generation of these benchmarks would greatly facilitate the implementation of our best priority pair prediction methodology. To automatically generate these benchmarks, a template for each type of benchmark must be generated that can be fine-tuned to any resource utilization level without manual intervention. To create such a template, a model could be created that takes as input the desired resource utilization level and generates the appropriate code. For example, given the desired utilization level, such a model would generate the appropriate memory-access pattern for L2 cache and TLB utilization; for the functional units it would generate the distance between dependent instructions.

7.2.5 Co-schedule Formation

In this dissertation we are concerned with improving the throughput of a given co-schedule. However, given a set of applications, our methodology could be extended to map applications to cores. Related research [23, 25, 27-29, 32, 57] has studied the formation of co-schedules given a set of applications. Only [23, 32] use intervals of execution to dynamically adapt the co-schedules. It would be interesting to see if our signatures can identify co-schedules that yield more throughput improvements than related research.

7.2.6 Other Metrics

In this dissertation we studied improvements in throughput as measured by $IPC_{\text{aggregate}}$. However, there are other metrics that are of interest and our signature methodology could be used to optimize them. Some of these metrics are discussed below:

- **Fairness:** This metric ensures that each application thread in a co-schedule experiences the same percentage loss of throughput in SMT mode as compared to their single-threaded mode performance. This metric has been studied by [26] and [30]. In [26] researchers showed that malicious code can slow down the second application of a co-schedule by a factor of 10. In [30] researchers allocate time slices to an application based on its fair share of the L2 cache, e.g., in an n -threaded system, the performance obtained with a cache of $1/n^{th}$ the size. Using signatures of critical shareable core resources may yield higher accuracy in forming co-schedules from a job queue, as well as setting hardware thread priorities for a given co-schedule.
- **Power consumption:** Conflicts for resources can lead to higher latencies and as a result longer run times that increase consumption of power. Our signature methodology could be extended to include resources that are significant in terms of power usage and this information could be used to form co-schedules from a given job queue or set hardware thread priorities for a given co-schedule.

7.2.7 Representativity of SPEC CPU2006 Benchmarks

Researchers [64] studied and clustered SPEC CPU2006 suite according to similarity. They found that six of 12 integer-intensive and eight of 17 floating-point intensive applications were sufficient to represent the characteristics of the entire suite. In their study, they characterized applications using hardware performance counters and used clustering to form groups that have similar characteristics. Furthermore, they identified for each cluster a representative application. It would be interesting to see if the set of signatures found in this implementation can be used to characterize the representative of each cluster.

-threaded mode. This characterization generates a time-ordered signature set for each application. Assume a signature pair that characterizes either a co-schedule of signature phases of two applications or the majority of the execution times of two applications. In these cases, we propose a methodology that predicts the best priority pair. Our methodology can be used only for processors that

allow software to set hardware thread priorities. Hence, at present it can be used only on the IBM POWER5 and IBM POWER6 processors.

We presented an implementation of this methodology for the IBM POWER5 processor and demonstrated the efficacy of our methodology to improve throughput for applications with a dominating signature. We showed that, for the IBM POWER5, non-default hardware thread priority pairs should be used, in particular, to improve throughput of applications that have utilization of the floating-point unit 10% or higher than that of the fixed point unit. The proposed methodology makes significant contributions towards application characterization that prove useful in terms of determining hardware thread priorities that can enhance processor core throughput. The contributions made in this dissertation are described in Section 7.1, and Section 7.2 describes future work.

7.1 CONTRIBUTIONS

We presented a three-phase methodology to predict the best priority pair for a co-schedule executed on an SMT processor that supports software-controlled hardware thread priorities, and implemented and evaluated the methodology using an IBM POWER5 processor. The research questions that were raised and answered in this dissertation are as follows:

3. Can judicious setting of hardware thread priorities be used to improve the throughput of an SMT processor?
4. How can application characteristics be used to predict the best priority pair for a given co-schedule comprising two applications?

While answering these questions, this dissertation made the following major contributions:

6. Demonstrated that the judicious setting of hardware thread priorities can be used to improve SMT processor throughput.

- a. Using an IBM POWER5 simulator and 263 partial application trace-pairs, we showed that the default priority pair (equal priorities) does not provide the best throughput for 82% of the application trace-pairs.
 - b. For the application trace-pairs studied, the improvement in throughput, as measured by $IPC_{\text{aggregate}}$, achieved by the best priority pair, as compared to the default priority pair, is between 0.00% and 25.52%. The worst-case throughput differs from the best case by as little as 0.42% and as much as 51.09%. Thus, the choice of priority pair can have a significant impact on throughput and the wrong choice can lead to significant throughput loss.
- 7. Developed the concept of a Shareable Resource Signature. For an application executing in single threaded mode and for a specified execution time interval, the shareable resource signature characterizes the application's utilization of critical shareable core resources.
- 8. Developed a three-phase methodology that, given a signature pair that characterizes either a co-schedule of signature phases of two applications or the majority of the execution times of two applications, predicts the best priority pair for the co-schedule.
- 9. Implemented the methodology for the an IBM POWER5 processor, which shows the following:
 - a. 17 of 10,000 possible signatures are sufficient to characterize 95.6% of the execution times of a set of applications that consists of 20 SPEC CPU2006 benchmarks (1 data input), three NAS NPB benchmarks (3 data inputs), and 10 PETSc KSP solvers (12 data inputs). The cgs and lsqr PETSc KSP solvers have signatures that are independent of input data, while one of three NAS NPB benchmark (bt-mz) has a signature that is independent of the input data.
 - b. For 21 co-schedules of applications, each with a signature that characterizes 95% of its execution time, our validation study shows the following:
 - i. *Predicted best priorities yield higher throughput than default priorities for all but one of the 21 co-schedules.* Initial results showed that two co-schedules

(462.libquantum, 437.leslie3d) and (bt-mz.A, lu-mz.A) experience a throughput loss of 7.46% and 20.05%, respectively, at predicted priorities, as compared to that achieved at default priorities. Further investigation shows that for the co-schedule (bt-mz.A, lu-mz.A) mapping and executing the co-schedule with the predicted best priorities on hardware threads (5, 4), instead of (4, 5), results in a 3.56% higher throughput as compared to default priorities – this is in contrast to the 20.05% throughput loss experienced when executed on hardware threads (4, 5). Although we have not verified it, one possible reason for this is that the processor core favors one hardware thread over the other. Re-executing the co-schedule (462.libquantum, 437.leslie3d) on hardware threads (5,4), instead of (4, 5), results in predicted priorities yielding lower throughput than the default priorities. Thus, we claim that predicted best priorities yield equal or higher throughput than default priorities for 20 of the 21 co-schedules studied, and for the outlier the throughput loss is 7.46%.

- ii. *Using non-default priorities improves throughput.* The default priority pair yields best throughput for only six of the 21 co-schedules. For the remaining 15 the default priority pair yields throughput that is between 0.74% and 14.10% lower than that achieved with the best priority pair.
- iii. *Using the predicted best priority pair, rather than default priorities, improves throughput or at least provides throughput equal to that achieved with default priorities.* For 11 of the 21 co-schedules both the default and predicted priorities yield equal throughput. For nine of the 21 predicted priorities yield throughput that is between 0.59% and 16.42% higher than that achieved with default priorities. For two of these nine co-schedules the predicted priority pair yields a throughput improvement of less than 5%. Furthermore, for three the throughput improvement associated with executing with the predicted

priority pair, rather than default priorities, is between 5% and 10% and for the other four the improvement is greater than 10%.

- iv. *Using predicted best priority pairs appears to be most applicable to floating-point “intensive” applications:* For eight co-schedules comprising applications for which the utilization of the floating-point unit exceeds that of the fixed-point unit by 10% or more, the predicted priority pairs, as compared to the default priorities, yield a throughput improvement between 3.56% and 16.42%. This result indicates that the methodology for predicting best priority pairs is most applicable to applications for which floating-point unit utilization dominates that of the fixed point unit by at least 10%.

7.2 FUTURE WORK

While this dissertation provides a methodology that can be used to set hardware thread priorities to improve the core throughput of SMT processors with software-controlled hardware thread priorities, the methodology can be improved and extended in several ways.

- Our current prediction methodology is limited to applications that have a time-ordered signature set comprising signatures in the set of signatures that characterize the signature-generating applications. In Section 7.2.1 we describe an extension of our methodology to predict best priority pairs for applications with signatures not included in the set.
- Our implementation for the IBM POWER5 was restricted to applications with a dominating signature. Using these applications, we could statically set priorities at the beginning of the execution of a co-schedule and keep them constant for the entire execution times of the two applications. However, as noted in Appendix G, in our implementation, 114 out of 149 application-input-data combinations did not have dominating signatures. For such applications we need to extend our methodology to dynamically adapt priorities when the co-scheduled signature phases change; this extension is discussed in Section 7.2.2.

- For applications with signature phases, a signature phase may correlate to a phase of execution. If this is true, then signatures can be used to detect phases of execution; this is discussed in Section 7.2.3.
- To validate resource utilization metrics and generate the prediction table, we create resource-stress and signature microbenchmarks by hand and empirically tune them to get the desired utilization levels. Thus, to improve productivity, in Section 7.2.4, we discuss an extension of this research to generate the microbenchmarks automatically.
- Signatures also may be used forming co-schedules of application threads. Mapping co-schedules of application threads to different cores based on their signatures can potentially improve throughput. This is described in Section 7.2.5.
- This dissertation seeks to improve $IPC_{\text{aggregate}}$. However, our methodology could be extended to study other metrics such as fairness and power consumption; this is discussed in Section 7.2.6.
- SpecCPU2006 benchmarks have been clustered according to their similarity by [64]. It would be interesting to see if the signatures for the SPEC CPU2006 benchmarks used in this implementation are sufficient to characterize the clusters identified in [64]; this is discussed in Section 7.2.7.

These future research directions are discussed in this section.

7.2.1 Best Priority Pair Prediction for Co-Schedules Characterized by “Other” Signatures

The methodology we presented in this dissertation is limited to the prediction of the best priority pairs for co-schedules comprising applications that have a time-ordered signature set of signatures in the set of signatures that characterize the signature-generating applications. In order to predict for applications with signatures not in this set a prediction model is needed. There are numerous methods that we could explore to construct such a model. Below, using neural networks and regression analysis

as examples, we describe how we might go about constructing such a model and how it could be used for prediction.

- A *neural network* is an artificial intelligence technique that uses a learning approach to model data and generate predictions. A set of independent variables are input to the network, which generates a prediction and computes the difference between the expected output and the value generated by the neural network. The accuracy of the prediction is then fed back to the neural network to adjust its prediction function and reduce prediction error. Once a neural network is trained properly with signature pairs in the initial set, then it could be used to predict for a signature pairs not in the set.
- *Regression analysis* is a technique that generates an analytical expression relating a dependent variable with one or more independent variables. The analytical expression generated is parameterized by the values of the independent variables to produce a value for the dependent variable. Regression analysis could be used to create an analytical expression parameterized by the resource utilization levels of the signature pairs in the initial set, i.e., those that characterize the signature-generating applications, that could predict the best priority pairs for co-schedules characterized by these signature pairs. Once the expression has been generated, it can be used to predict for co-schedules characterized by signature pairs that were not used to develop the expression.

7.2.2 Dynamic Adaptation of Priorities for Applications with Signature Phases

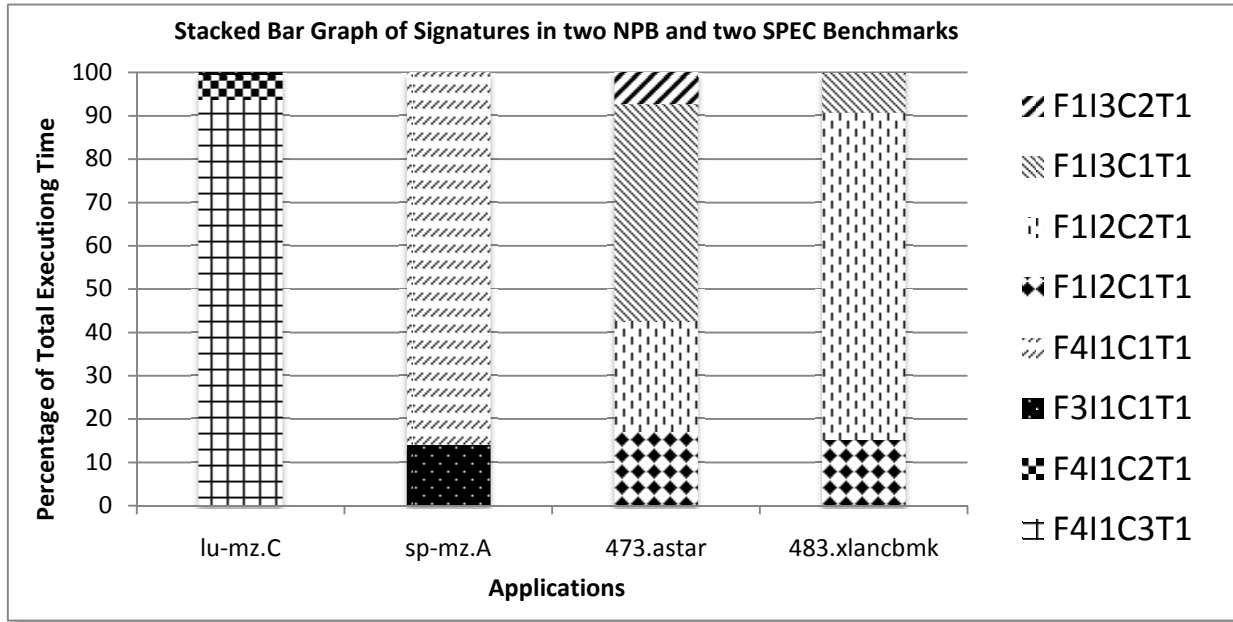


Figure 7.1: Stacked Bar Graph of Signatures found in two NAS and two SPEC Benchmarks

In this dissertation we had to restrict our validation study to those applications that have a dominating signature. As shown in Appendix G, 114 out of the 149 application-input-data combinations in our set of signature-generating applications did not have a dominating signature. To use our best priority pair prediction methodology for applications without a dominating signature, mechanisms must be developed to detect changes in signature phases and adapt the hardware thread priorities to the associated predicted best priority pair. For example, Figure 7.1 shows the stacked bar graph of the signature sets of two NAS and two SPEC benchmarks. In this figure, the X-axis represents the applications and the Y-axis represents the percentage of execution time a signature was observed for a given application. The legend shows the signature corresponding to the patterned bars in the graph. For example, the sp-mz.A application has signatures F3I1C1T1 and F4I1C1T1 for 14% and 86% of its execution time, respectively.

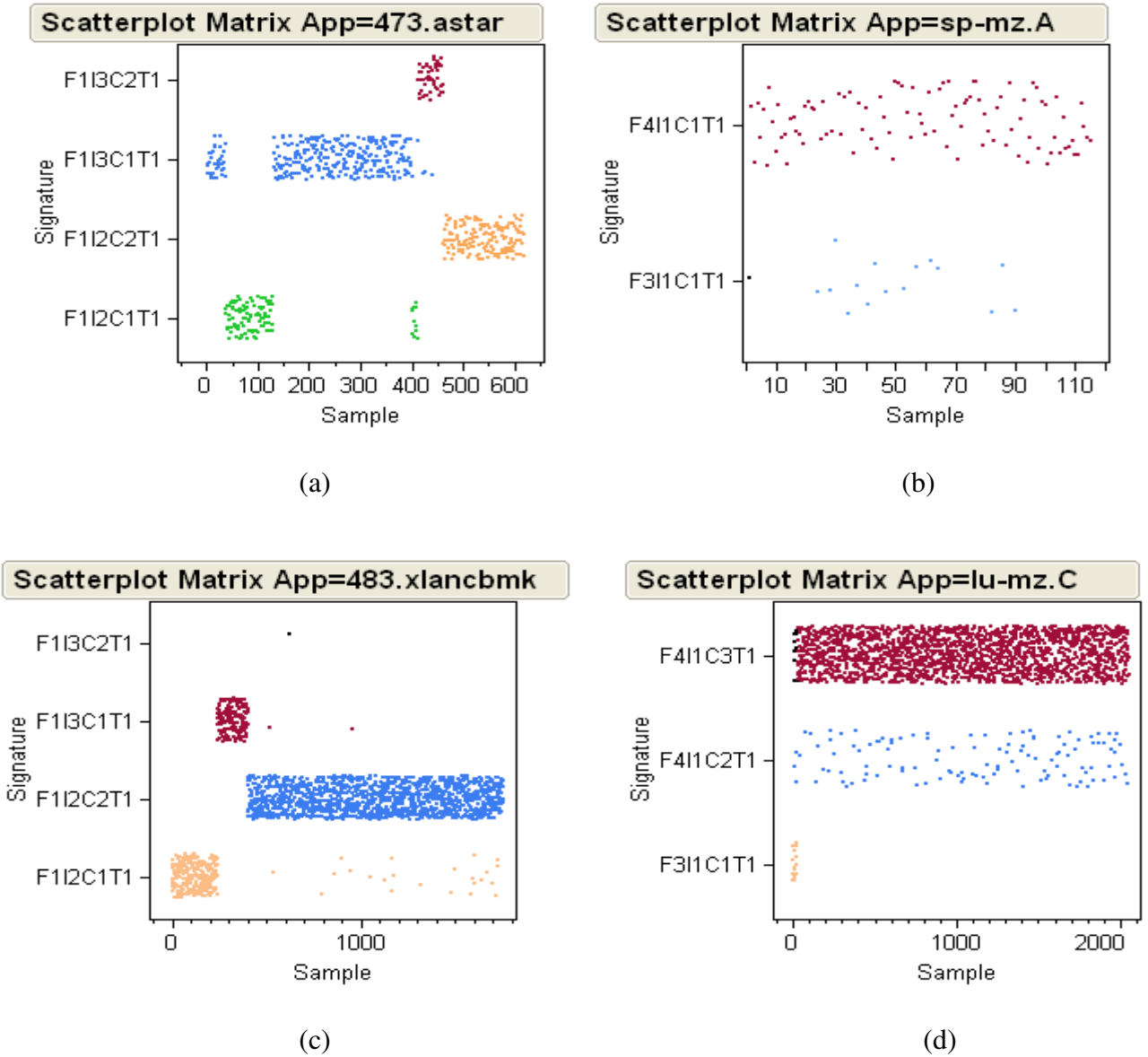


Figure 7.2: Scatter-plot of Signatures found in 2 SPEC CPU2006 and NAS NPB Benchmarks

The signatures can appear in distinct application execution phases or can be interleaved during an application's execution. For example, Figure 7.2 shows a scatter plot of the signatures of two SPEC CPU2006 in plots (a) and (c), and two NAS NPB benchmarks in plots (b) and (d) over time. In this figure, the X-axis represents execution time and the Y-axis represents signatures. As shown in Figure 7.2, the signatures of NAS NPB are interleaved and, thus, change frequently, whereas for the two SPEC CPU2006 benchmarks the signatures appear in phases.

Accordingly, for application pairs with signature phases that are shorter than the length of a dominating signature, the best priority pair may change when an application changes to a different signature phase. For such application pairs, we need to implement mechanisms for demarcating the signature phases of the applications and identifying the current signature pair. In this way, the appropriate predicted best priority pair can be used to set the priorities for each signature phase. Such an implementation must take into account the overhead associated with detecting phase changes as well as the cost associated with adapting the hardware thread priorities. On the IBM POWER5, it takes approximately one millisecond to set the hardware thread priority. Thus, the minimum phase length has to be significantly greater than one millisecond. One possible way to detect phase changes is to insert special instructions in the application code that announces signature phase changes and implement a run-time system that uses these special instructions to determine when and how to adapt hardware thread priorities. The implementation cost will determine the minimum phase length that can be used for dynamic priority adaptation; for shorter phases equal priorities (default) can be used.

7.2.3 Correlation of Application Phases and Signature Phases

Applications often go through different phases of execution. To detect application phases, phase detection techniques use metrics such as basic block vectors, data locality, working set size, and IPC [59-62]. Based on changes in the monitored metrics, phase detection techniques characterize an application's execution phases and carry out adaptations to optimize performance [54, 55].

In this dissertation we characterize applications using signature phases. Hence, a study that evaluates the correspondence of a signature phase to an execution phase would answer the following question: Can a shareable resource signature be used to detect an application's execution phases?

7.2.4 Automatic Benchmark Generation

To validate the methods used to realize the utilization metrics used in our POWER5 implementation, we created resource-stress microbenchmarks to test the accuracy of the associated metrics. These benchmarks are generated by hand and empirically fine-tuned. The signature

microbenchmarks were created in a similar fashion, i.e., by combining the resource-stress benchmarks and empirically fine-tuning them until the desired signature was observed. To apply the methodology to a different architecture, the benchmarks for that architecture would have to be generated in the same way. Hence, automatic generation of these benchmarks would greatly facilitate the implementation of our best priority pair prediction methodology. To automatically generate these benchmarks, a template for each type of benchmark must be generated that can be fine-tuned to any resource utilization level without manual intervention. To create such a template, a model could be created that takes as input the desired resource utilization level and generates the appropriate code. For example, given the desired utilization level, such a model would generate the appropriate memory-access pattern for L2 cache and TLB utilization; for the functional units it would generate the distance between dependent instructions.

7.2.5 Co-schedule Formation

In this dissertation we are concerned with improving the throughput of a given co-schedule. However, given a set of applications, our methodology could be extended to map applications to cores. Related research [23, 25, 27-29, 32, 57] has studied the formation of co-schedules given a set of applications. Only [23, 32] use intervals of execution to dynamically adapt the co-schedules. It would be interesting to see if our signatures can identify co-schedules that yield more throughput improvements than related research.

7.2.6 Other Metrics

In this dissertation we studied improvements in throughput as measured by $IPC_{\text{aggregate}}$. However, there are other metrics that are of interest and our signature methodology could be used to optimize them. Some of these metrics are discussed below:

- **Fairness:** This metric ensures that each application thread in a co-schedule experiences the same percentage loss of throughput in SMT mode as compared to their single-threaded mode performance. This metric has been studied by [26] and [30]. In [26] researchers showed that malicious code can slow down the second application of a co-

schedule by a factor of 10. In [30] researchers allocate time slices to an application based on its fair share of the L2 cache, e.g., in an n -threaded system, the performance obtained with a cache of $1/n^{th}$ the size. Using signatures of critical shareable core resources may yield higher accuracy in forming co-schedules from a job queue, as well as setting hardware thread priorities for a given co-schedule.

- Power consumption: Conflicts for resources can lead to higher latencies and as a result longer run times that increase consumption of power. Our signature methodology could be extended to include resources that are significant in terms of power usage and this information could be used to form co-schedules from a given job queue or set hardware thread priorities for a given co-schedule.

7.2.7 Representativity of SPEC CPU2006 Benchmarks

Researchers [64] studied and clustered SPEC CPU2006 suite according to similarity. They found that six of 12 integer-intensive and eight of 17 floating-point intensive applications were sufficient to represent the characteristics of the entire suite. In their study, they characterized applications using hardware performance counters and used clustering to form groups that have similar characteristics. Furthermore, they identified for each cluster a representative application. It would be interesting to see if the set of signatures found in this implementation can be used to characterize the representative of each cluster.

References

- [1].J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, CA, 2003.
- [2].T. Ungerer, B. Robic, and J. Silc, “A Survey of Processors with Explicit Multithreading,” *ACM Computing Surveys (CSUR)*, March 2003, 35(1):29-63.
- [3].D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing On-chip Parallelism,” in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA '95)*, IEEE Computer Society, June 1995, pp. 392-403.
- [4].P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-Way Multithreaded SPARC Processor,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '05)*, March 2005, 25(2):21-29.
- [5].D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, “Hyper-Threading Technology Architecture and Microarchitecture,” *Intel Technology Journal*, February 2002, 3(1):4-15.
- [6].D. Koufaty and D. T. Marr, “Hyper-Threading Technology in the Netburst Microarchitecture,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '03)*, March 2003, 23(2):56-65.
- [7].W. Margo, P. Peterson, and J. Tubella, “Hyper-Threading Technology: Impact on Compute-Intensive Workloads,” *Intel Technology Journal*, February 2002, 3(1):58-66.
- [8].Y. K. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belonov, and I. Santos, “Media Applications on Hyper-Threading Technology,” *Intel Technology Journal*, February 2002, 3(1):47-57.

- [9]. J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A Multithreaded PowerPC Processor for Commercial Servers," *IBM Journal Research and Development*, November 2000, 44(6):885-898.
- [10]. R. Kalla, B. Sinahroy, and J. M. Tendler, "IBM POWER5 Chip: a Dual-Core Multithreaded Processor," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '04)*, March 2004, 24(2):40-47.
- [11]. H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel, "Characterization of Simultaneous Multithreading (SMT) Efficiency in POWER5," *IBM Journal of Research and Development*, July 2005, 49(4/5):555-564.
- [12]. D. M. Tullsen, "Simulation and Modeling of a Simultaneous Multithreading Processor," in *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996.
- [13]. R. Espasa and M. Valero, "Exploiting Instruction and Data-Level Parallelism," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '97)*, September 1997, 17(5):20-27.
- [14]. U. Sigmund and T. Ungerer, "Evaluating a Multithreaded Superscalar Microprocessor versus a Multiprocessor Chip," in *Proceedings of the 4th PASA Workshop on Parallel Systems and Algorithms*, Germany, April 1996, pp. 147-159.
- [15]. J. P. Wittenburg, G. Meyer, and P. Pirsch, "Adapting and Extending Simultaneous Multithreading for High Performance Video Signal Processing Applications," in *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation*, January 1999, pp. 1-6.
- [16]. N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '06)*, July 2006, 26(4):52-60.

- [17]. alphaWorks: IBM Performance Simulator for Linux on POWER: Overview,” <http://www.alphaworks.ibm.com/tech/simppc>, accessed 02/20/2009.
- [18]. “IBM Power Systems Software – AIX,” <http://www.ibm.com/aix>, accessed 02/20/2009.
- [19]. “The Linux Kernel Archives,” <http://www.kernel.org>, accessed 02/20/2009.
- [20]. “Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations,” SG245768, <http://www.redbooks.ibm.com/abstracts/sg245768>, accessed 02/20/2009.
- [21]. H. Q. Le, et al., “IBM POWER6 Microarchitecture,” *IBM Journal of Research and Development*, November 2007, 51(6):639-662.
- [22]. “UltraSPARC-T2 Processor Overview,” <http://www.sun.com/processors/UltraSPARC-T2/index.xml>, accessed 02/20/2009.
- [23]. A. Snively and D. M. Tullsen, “Symbiotic Job Scheduling for a Simultaneous Multithreading Processor,” in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, ACM Press, November 2000, pp. 234-244.
- [24]. J. A. Redstone, S. J. Eggers, and H. M. Levy, “An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture,” in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, November 2000, pp. 245-256
- [25]. J. Nakajima and V. Pallipadi, “Enhancements for Hyper-Threading Technology in the Operating System Seeking the Optimal Scheduling,” in *Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software*, The USENIX Association, December 2002, pp. 3-3.
- [26]. D. Grunwald and S. Ghiasi, “Microarchitectural Denial of Service: Insuring Microarchitectural Fairness,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '02)*, IEEE Computer Society, November 2002, pp. 409-418.

- [27]. D. Doucette and A. Fedorova, "Base Vectors: A Potential Technique for Micro-architectural Classification of Applications," in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '07)*, in conjunction with ISCA-34, June 2007.
- [28]. T. Moseley, D. Grunwald, J. L. Kihm, and A. D. Connors, "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors," in *Proceedings of the 2005 International Conference on Computer Design (ICCD '05)*, IEEE Computer Society, October 2-5, 2005, pp. 373-380.
- [29]. R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005, pp.28-28.
- [30]. A. Fedorova, M. Seltzer, and M. D. Smith, "A Non-Work-Conserving Operating System Scheduler for SMT Processors," in *Proceedings of the Workshop on the Interaction between the Operating Systems and Computer Architecture (WIOSCA '06)*, in conjunction with ISCA-33, June 2006, pp. 1-6.
- [31]. F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, and A. Ramirez, "Predictable Performance in SMT Processors: Synergy between the OS and SMTs," *IEEE Transactions on Computers*, July 2006, 55(7): 785-799.
- [32]. J. Bulpin and I. Pratt, "Hyper-Threading Aware Process Scheduling Heuristics," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX '05)*, April 2005, pp. 399-402.
- [33]. C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and M. Valero, "Balancing HPC Applications Through Smart Allocation of Resources in MT Processors," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS '08)*, April 2008, pp. 1-12.
- [34]. C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C. Cher, and M. Valero, "Software-Controlled Priority Characterization of POWER5 Processor," in *Proceedings of the 35th International Symposium of Computer Architecture (ISCA '08)*, June 2008, pp. 415-426.

- [35]. C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero, “A Dynamic Scheduler for Balancing HPC Applications,” in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC08)*, November 2008, pp. 1-10.
- [36]. A. Snaveley, N. Wolter, and L. Carrington, “Modeling Application Performance by Convolving Machine Signatures with Application Profiles,” in *Proceedings of IEEE Workshop on Workload Characterization (WWC '01)*, December 2001, pp. 149-156.
- [37]. “SPEC CPU2000,” <http://www.spec.org/cpu2000/>, accessed 03/02/2009.
- [38]. “SPEC CPU2006,” <http://www.spec.org/cpu2006/>, accessed 03/02/2009.
- [39]. “STREAM2,” <http://www.cs.virginia.edu/stream/stream2/>, accessed 03/02/2009.
- [40]. “LMbench,” <http://sourceforge.net/projects/lmbench>, accessed 03/02/2009.
- [41]. “ITRACE – Instruction Tracing,” <http://perfinsp.sourceforge.net/itrace.html>, accessed 03/02/2009.
- [42]. “Alphaworks: Pmcount for Linux on Power Architecture,” <http://www.alphaworks.ibm.com/tech/pmcount>, accessed 03/02/2009.
- [43]. D. Villa, M. Meswani, P. Teller, and B. Olszewski, “Profiling Memory Subsystem Performance in an Advanced Power Virtualization Environment,” in *Proceedings of the First International Workshop on Operating System Interference in High Performance Applications (OSIHPA'05)*, in conjunction with the PACT05 Conference, September 2005, pp. 1-6.
- [44]. B. Maron, T. Chen, D. Vianney, B. Olszewski, S. Kunkel, and A. Mericas, “Workload Characterizations for the Design of Future Servers,” in *Proceedings of the 2005 Workload Characterization Symposium (IISWC'05)*, October 2005, pp. 129-136.
- [45]. “Statistical Toolbox – Documentation,” <http://www.mathworks.com/access/helpdesk/help/toolbox/stats/index.html?/access/helpdesk/help/toolbox/stats/classregtree.html>, accessed 03/29/2009.

- [46]. “MATLAB – The Language of Technical Computing,”
<http://www.mathworks.com/products/matlab/>, accessed 03/29/2009.
- [47]. A. Mericas, “POWER5 Performance Measurement and Characterization,” Tutorial at the IEEE IISWC Conference, October 2005.
- [48]. A. Mericas, “Performance Monitoring on the POWER5 Microprocessor,” *Performance Evaluation and Benchmarking*, L. K. John and L. Eckhout, Editors, CRC Press, 2006, pp. 247-266.
- [49]. “IBM p550,” http://www.nasi.com/IBM_p550.php, accessed January 12, 2009.
- [50]. “openSUSE,” <http://www.opensuse.org/en/>, accessed January 12, 2009.
- [51]. “PAPI,” <http://icl.cs.utk.edu/papi/>, accessed January 12, 2009.
- [52]. “NAS Parallel Benchmark Challenges,”
<http://www.nas.nasa.gov/Resources/Software/npb.html>NAS, accessed January 12, 2009.
- [53]. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries,” *Modern Software Tools in Scientific Computing*, Birkh Press, 1997, pp. 163-202.
- [54]. C. Krintz and R. Wolski, “Using Phase Behavior in Scientific Application to Guide Linux Operating System Customization,” in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005, pp.219-229.
- [55]. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, “Discovering and Exploiting Program Phases,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '03)*, December 2003, pp. 84-93.
- [56]. E. Perelman, M. Polito, J. Bouguet, J. Sampson, B. Calder, and C. Dulong, “Detecting Phases in Parallel Applications on Shared Memory Architectures,” in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, April 2006, pp. 187-192.

- [57]. D. Shepelov, J. C. Saez, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: A Scheduler for Heterogeneous Multicore Systems," *Operating Systems Review*, April 2009, 43(2): 66-75.
- [58]. G. Hinton, D. Sager, M. Upton, D. Boggs, "The Microarchitecture of the Pentium 4 Processor," in *Intel Technology Journal*, February 2001, 5(1):15-27.
- [59]. T. Sherwood, E. Perelman, B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001, pp. 3-14.
- [60]. A. Dhodapkar, J. E. Smith, "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis," *ACM SIGARCH Computer Architecture News*, May 2002, 30(2): 223-244.
- [61]. X. Shen, Y. Zhong, C. Ding, "Locality Phase Detection" in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASLPOS '04)*, October 2004, pp. 165-176.
- [62]. J. Lau, S. Schoenmackers, B. Calder, "Transition Phase Classification and Prediction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'05)*, February 2005, pp. 278-289.
- [63]. "IBM Power Systems performance benchmarks," <http://www-03.ibm.com/systems/power/hardware/benchmarks/index.html>, accessed 11/23/2009.
- [64]. A. Phansalkar, A. Joshi, and L. K. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," in *SIGARCH Computer Architecture News*, May 2007, 35(2): 412-423.

Appendix A: Utilization Validation Microbenchmarks

This appendix lists the source code of the stress benchmarks that were used to validate the utilization metrics described in Chapter 7.

Floating-point Unit (FPU) Utilization Microbenchmark:

```
# ***** PROGRAM COMPILATION *****  
# ***** This is directly written in assembler, there is no c language source *****  
# This program has to be compiled using the asm compiler and loader without using gcc as shown below:  
# as -a64 fpu_Stresser-stock.s -o fpu_Stresser-stock.o  
# ld -melf64ppc fpu_Stresser-stock.o -o fpu_Stresser-stock
```

```
        .file "fpu_Stresser.s"  
        .section      .rodata.str1.4,"aMS",@progbits,1  
        .align 2  
.LC20:  
        .string      "%t%f"  
        .section      .rodata.cst4,"aM",@progbits,4  
        .align 2  
.LC0:  
        .long        1042066440  
        .align 2  
.LC1:  
        .long        1047502258  
        .align 2  
.LC2:  
        .long        1052082438  
        .align 2  
.LC3:  
        .long        1054774846  
        .align 2  
.LC4:  
        .long        1058032142  
        .align 2  
.LC5:  
        .long        1059402673  
        .align 2  
.LC6:  
        .long        1064622233  
        .align 2  
.LC7:  
        .long        1060228615  
        .align 2  
.LC8:  
        .long        1054806051  
        .align 2  
.LC9:  
        .long        1059596450  
        .align 2  
.LC10:  
        .long        1051714863  
        .align 2  
.LC11:  
        .long        1052773659  
        .align 2  
.LC12:  
        .long        1049112871  
        .align 2  
.LC13:  
        .long        1059032165  
        .align 2  
.LC14:  
        .long        1059370796  
        .align 2  
.LC15:
```

```

        .long      1057719583
        .align 2
.LC16:  .long      1057727132
        .align 2
.LC17:  .long      1057722938
        .align 2
.LC18:  .long      1061391612
        .align 2
.LC19:  .long      1068289229
        .align 2
.LC22:  .long      1064589853
        .align 2
.LC24:  .long      1057679712
        .align 2
.LC26:  .long      1055365236
        .align 2
.LC28:  .long      1055773993
        .align 2
.LC30:  .long      1057187158
        .align 2
.LC32:  .long      1039954122
        .align 2
.LC34:  .long      1057354695
        .align 2
.LC36:  .long      1061082911
        .align 2
.LC38:  .long      1058028854
        .align 2
.LC40:  .long      1063131317
        .align 2
.LC42:  .long      1058532959
        .section   ".text"
        .align 2
        .p2align 4,,15
        .globl main
        .type      main, @function
main:
        mflr 0
        stwu 1,-176(1)
        lis 9,.LC7@ha
        lis 8,.LC0@ha
        lis 7,.LC1@ha
        lis 6,.LC2@ha
        lis 5,.LC3@ha
        lis 4,.LC4@ha
        lis 3,.LC5@ha
        lis 11,.LC6@ha
        lis 12,.LC9@ha
        lis 10,.LC10@ha
        stfd 18,64(1)
        nop
        nop
        lfs 18,.LC7@l(9)
        lis 9,.LC18@ha
        stfd 19,72(1)
        stfd 20,80(1)
        lfs 0,.LC18@l(9)

```



```

stfd 21,88(1)
lis 9,.LC19@ha
stfd 22,96(1)
stfd 23,104(1)
stfd 24,112(1)
stfd 25,120(1)
stw 29,20(1)
lfs 25,.LC0@l(8)
lis 29,.LC8@ha
lis 8,.LC11@ha
lfs 24,.LC1@l(7)
lfs 23,.LC2@l(6)
lis 7,.LC12@ha
lfs 22,.LC3@l(5)
lfs 21,.LC4@l(4)
lis 6,.LC13@ha
lis 5,.LC14@ha
lfs 20,.LC5@l(3)
lfs 19,.LC6@l(11)
lis 4,.LC15@ha
lis 11,.LC17@ha
lis 3,.LC16@ha
stfd 14,32(1)
stfd 15,40(1)
stfd 16,48(1)
stfd 17,56(1)
stfd 26,128(1)
stfd 27,136(1)
stfd 28,144(1)
stfd 29,152(1)
stfd 30,160(1)
stfd 31,168(1)
lfs 26,.LC17@l(11)
stw 0,180(1)
li 11,0
lfs 17,.LC8@l(29)
lfs 16,.LC9@l(12)
lfs 15,.LC10@l(10)
lfs 14,.LC11@l(8)
lfs 31,.LC12@l(7)
lfs 30,.LC13@l(6)
lfs 29,.LC14@l(5)
lfs 28,.LC15@l(4)
lfs 27,.LC16@l(3)
stfs 0,8(1)

```

```

.L2:
lis 12,0xee6
ori 10,12,45696
mtctr 10
.p2align 4,,15
lfs 0,.LC19@l(9)

```

```

.L3:
fadds 1,1,0
fadds 2,2,0
fadds 3,3,0
fadds 4,4,0
fadds 5,5,0
fadds 6,6,0
fadds 7,7,0
fadds 8,8,0
fadds 10,10,0
fadds 11,11,0
fadds 12,12,0
fadds 13,13,0
fadds 14,14,0
fadds 15,15,0
fadds 16,16,0
fadds 17,17,0
fadds 18,18,0
fadds 19,19,0
fadds 20,20,0
fadds 21,21,0

```

fadds 22,22,0
fadds 23,23,0
fadds 24,24,0
fadds 25,25,0
fadds 26,26,0
fadds 27,27,0
fadds 28,28,0
fadds 29,29,0
fadds 30,30,0
fadds 31,31,0
fadds 1,1,0
fadds 2,2,0
fadds 3,3,0
fadds 4,4,0
fadds 5,5,0
fadds 6,6,0
fadds 7,7,0
fadds 8,8,0
fadds 10,10,0
fadds 11,11,0
fadds 12,12,0
fadds 13,13,0
fadds 14,14,0
fadds 15,15,0
fadds 16,16,0
fadds 17,17,0
fadds 18,18,0
fadds 19,19,0
fadds 20,20,0
fadds 21,21,0
fadds 22,22,0
fadds 23,23,0
fadds 24,24,0
fadds 25,25,0
fadds 26,26,0
fadds 27,27,0
fadds 28,28,0
fadds 29,29,0
fadds 30,30,0
fadds 31,31,0
fadds 1,1,0
fadds 2,2,0
fadds 3,3,0
fadds 4,4,0
fadds 5,5,0
fadds 6,6,0
fadds 7,7,0
fadds 8,8,0
fadds 10,10,0
fadds 11,11,0
fadds 12,12,0
fadds 13,13,0
fadds 14,14,0
fadds 15,15,0
fadds 16,16,0
fadds 17,17,0
fadds 18,18,0
fadds 19,19,0
fadds 20,20,0
fadds 21,21,0
fadds 22,22,0
fadds 23,23,0
fadds 24,24,0
fadds 25,25,0
fadds 26,26,0
fadds 27,27,0
fadds 28,28,0
fadds 29,29,0
fadds 30,30,0
fadds 31,31,0
fadds 1,1,0
fadds 2,2,0

```

fadds 3,3,0
fadds 4,4,0
fadds 5,5,0
fadds 6,6,0
fadds 7,7,0
fadds 8,8,0
fadds 10,10,0
fadds 11,11,0
fadds 12,12,0
fadds 13,13,0
fadds 14,14,0
fadds 15,15,0
fadds 16,16,0
fadds 17,17,0
fadds 18,18,0
fadds 19,19,0
fadds 20,20,0
fadds 21,21,0
fadds 22,22,0
fadds 23,23,0
fadds 24,24,0
fadds 25,25,0
fadds 26,26,0
fadds 27,27,0
fadds 28,28,0
fadds 29,29,0
fadds 30,30,0
fadds 31,31,0
fadds 1,1,0
fadds 2,2,0
fadds 3,3,0
fadds 4,4,0
fadds 5,5,0
fadds 6,6,0
fadds 7,7,0
fadds 8,8,0
fadds 10,10,0
fadds 11,11,0
fadds 12,12,0
fadds 13,13,0
fadds 14,14,0
fadds 15,15,0
fadds 16,16,0
fadds 17,17,0
fadds 18,18,0
fadds 19,19,0
fadds 20,20,0
fadds 21,21,0
fadds 22,22,0
fadds 23,23,0
fadds 24,24,0
fadds 25,25,0
fadds 26,26,0
fadds 27,27,0
fadds 28,28,0
fadds 29,29,0
fadds 30,30,0
fadds 31,31,0
    bdnz .L3
    cmpwi 7,11,7
    addi 11,11,1
    bne 7,.L2
    lwz 29,180(1)
lfd 14,32(1)
lfd 15,40(1)
lfd 16,48(1)
lfd 17,56(1)
lfd 18,64(1)
lfd 19,72(1)
lfd 20,80(1)
mtr 29
lfd 21,88(1)

```

```

lwz 29,20(1)
lfd 22,96(1)
lfd 23,104(1)
lfd 24,112(1)
lfd 25,120(1)
lfd 26,128(1)
lfd 27,136(1)
lfd 28,144(1)
lfd 29,152(1)
lfd 30,160(1)
lfd 31,168(1)
addi 1,1,176
blr

.size      main,-main
.ident     "GCC: (GNU) 4.1.0 (SUSE Linux)"
.section   .note.GNU-stack,"",@progbits

```

Fixed-point Unit (FPU) Utilization Microbenchmark:

```

# ***** PROGRAM COMPILATION *****
# ***** This is directly written in assembler, there is no c language source *****
# This program has to be compiled using the asm compiler and loader without using gcc as shown below:
# as -a64 fxu_Stresser-stock.s -o fxu_Stresser-stock.o
# ld -melf64ppc fxu_Stresser-stock.o -o fxu_Stresser-stock
###PROGRAM DATA###
.data
.align 3
#value_list is the address of the beginning of the list
value_list:
    .quad 23, 50, 95, 96, 37, 85
#value_list_end is the address immediately after the list
value_list_end:

###STANDARD ENTRY POINT DECLARATION###
.section "opd", "aw"
.global _start
.align 3
_start:
    .quad _start, .TOC.@tocbase, 0

###ACTUAL CODE###
.text
_start:

    mflr 0
    stwu 1,-80(1)
    lis 15,0x5f5
    li 31,1
    li 30,2
    li 25,3
    ori 15,15,57600
    li 23,4
    li 24,5
    li 26,6
    li 18,7
    li 19,8
    li 20,9
    li 21,10
    li 22,11
    li 28,12
    li 27,13
    li 16,14
    li 17,0

.L2:
    divw 0,25,31
    addi 10,15,-1
    li 11,1
    rlwinm 9,10,0,30,31
    cmpwi 0,9,0
    add 12,0,17

```

```

add 0,31,12
add 7,26,12
add 30,30,12
add 25,25,12
add 31,23,12
add 8,24,12
add 6,18,12
add 5,19,12
add 4,20,12
add 3,21,12
add 29,22,12
add 28,28,12
add 27,27,12
add 26,16,12
beq 0, L15
cmpwi 1,9,1
beq 1, L24
cmpwi 6,9,2
beq 6, L25
add 0,0,12
add 30,30,12
add 25,25,12
add 31,31,12
add 8,8,12
add 7,7,12
add 6,6,12
add 5,5,12
add 4,4,12
add 3,3,12
add 29,29,12
add 28,28,12
add 27,27,12
add 26,26,12
li 11,2

```

.L25:

```

add 0,0,12
add 30,30,12
add 25,25,12
add 31,31,12
add 8,8,12
add 7,7,12
add 6,6,12
add 5,5,12
add 4,4,12
add 3,3,12
add 29,29,12
add 28,28,12
add 27,27,12
add 26,26,12
addi 11,11,1

```

.L24:

```

addi 16,11,1
add 0,0,12
add 30,30,12
add 25,25,12
cmpw 7,16,15
add 31,31,12
add 8,8,12
add 7,7,12
add 6,6,12
add 5,5,12
add 4,4,12
add 3,3,12
add 29,29,12
add 28,28,12
add 27,27,12
add 26,26,12
beq- 7, L23

```

.L15:

```

srwi 18,10,2
mtctr 18
.p2align 4,,15

```

.L3:

```
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
```

addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0

addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0
addi 30,30,0
addi 31,31,0
addi 0,0,0
addi 1,1,0
addi 2,2,0
addi 3,3,0
addi 4,4,0
addi 5,5,0
addi 6,6,0
addi 7,7,0
addi 8,8,0
addi 10,10,0
addi 11,11,0
addi 12,12,0
addi 13,13,0
addi 14,14,0
addi 15,15,0
addi 16,16,0
addi 17,17,0
addi 18,18,0
addi 19,19,0
addi 20,20,0
addi 21,21,0
addi 22,22,0
addi 23,23,0
addi 24,24,0
addi 25,25,0
addi 26,26,0
addi 27,27,0
addi 28,28,0
addi 29,29,0


```

        addi 30,30,0
        addi 31,31,0
        bdnz .L3
.L23:
        cmpwi 7,17,37
        addi 17,17,1
        bne+ 7,.,L2
        mr 3, 31
        li 0,1
        sc

```

L2 Cache Utilization Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*
    Code is based on the idea proposed in the following paper by D. Doucette and A. Fedorora
    "Base Vectors: A Potential Technique for Micro-architectural Classification of
    Applications",

*
    Basic algorithm is to allocate an array the size of the L2 cache or larger
*
    Think of this array as a multiple of cache lines
*
    Initialize the first element of every cache line to store the address of the first
    element of the next successive line, the last line of the array will store NULL.
*
    In a loop implement the concept of pointer chasing by first initializing the pointer
    to the first elem of the first line, the pointer then loads the address of the second
    element of the second line and so on, this will generate hits in the L2 cache missing L1.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main()
{

    int i,j,k, initindex, stride, elemXpage, padsz, pagesize, linesize, elemXline;
    int *arr, *elem=NULL;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    num_of_pages=480; //size of the L2 cache
    pagesize=4096; // 4k size
    padsz=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsz)/sizeof(int);

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(int); // num of elems in one page
    initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

    /***** first elem of each stores address of first element of third successive line **/
    stride=elemXline*3;
    for(i=0;i<(arrsize-stride);i+=stride)
    {
        arr[i]=(int) (arr+i+stride);
    }
    arr[arrsize-stride]=(int) NULL; //if last line, then there is no more pointer to store

```

```

        for(j=0;j<1825000;j++)
        {
            elem=(int *)arr[0]; //initialize to point to first elem of array
            while(elem!=NULL)// continue while not last line
            {
                elem=(int *)*elem; // load address of first elem of next line
            }
        }

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

free(arr);
free(elem);
}

```

TLB Utilization Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*   Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*   Think of this array as a multiple of pages and in our case we use 3000 page array
*   Initialize the first element of every cache line to store the address of the first
*   element of the next successive page, the last line of the array will store NULL.
*   In a loop implement the concept of pointer chasing by first initializing the pointer
*   to the first elem of the first line, the pointer then loads the address of the first
*   element of the second page and so on, this will generate hits misses in the TLB
*   as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main()
{

    int i,j,k, initindex, stride, elemXpage, padsz, pagesize, linesize, elemXline;
    int *arr, *elem=NULL;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsz=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsz)/sizeof(int);

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(int); // num of elems in one page
    initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

    /**** first elem of each page stores address of first element of first elem of a successive page **/

```

```

stride=elemXpage;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
}
arr[arrsize-stride]=(int) NULL; //if last page, then there is no more pointer to store

for(j=0;j<400000;j++)
{
    elem=(int *)arr[0]; //initialize to point to first elem of array
    while(elem!=NULL)// continue while not last line
    {
        elem=(int *)*elem; // load address of first elem of next line
    }
}

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

free(arr);
free(elem);
}

```

Appendix B: Signature Microbenchmarks

Microbenchmark design notes: For each of the 17 signature microbenchmarks, a main inner-loop accesses a 12MB array in the desired access pattern as well as computes floating point and integer instructions along with zero or more noop instructions to achieve the desired stress on the four resources. For seven of the 17 microbenchmarks corresponding to signatures F3I1C1T1, F3I1C2T1, F4I1C1T1, F4I1C2T1, F4I1C3T1, F4I2C1T1, and F5I1C1T1 we used floating-point array. While for the remaining 10 we used an integer array. The choice of array will determine the data-type and bytes accessed for every load and store. The seven microbenchmarks above correspond to floating-point intensive signatures and hence, we used floating-point arrays inside the main inner-loop for loads and stores. The F3I1C3T1 microbenchmark was originally designed with an integer array, however we leave its modification as future work as we do not use this microbenchmark for prediction. The main inner-loop is re-executed several times to achieve an execution time of approximately 100 seconds for each microbenchmark.

Using papi tool, signature was captured for one-second-time interval for all microbenchmarks. Based on experiments, we observed that predictions using signature microbenchmarks that did not have the desired signature for 95% or more of its execution gave inaccurate predictions. Hence, it was validated that signature microbenchmarks used for prediction, had the desired signature for 95% or more of its execution time. For the remaining microbenchmarks all but one had the desired signature for 95% or more if its execution time. The signature microbenchmark corresponding to signature F1I2C2T1 had the desired signature for 87% of its execution time and we leave the improvement of this microbenchmark as future work. Given below is the source code for the 17 signature microbenchmarks.

Source code of the microbenchmarks

F1I1C1T1 Microbenchmark:

```
/*** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
```

```

        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {

```

```

        printf("could not allocate memory\n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(int); // num of elems in one page
    initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS
    *****/

    PAPI_library_init(PAPI_VER_CURRENT);

    switch (papi_group)
    {

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 40:
        /***** pmcount group 40 *****/
        Events[0]=0x400000d1; //PM_TLB_MISS
        Events[1]=0x40000100; //PM_SLB_MISS
        Events[2]=0x40000104; //PM_BR_MPRED_CR
        Events[3]=0x40000105; //PM_BR_MPRED_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 42:
        /***** pmcount group 42 *****/
        Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
        Events[1]=0x4000003f; //PM_GRP_BR_REDIR
        Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;
    }

```

```

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC

```

```

break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/4;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i*stride);
}
arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

PAPI_start(EventSet);

for(j=0;j<645;j++)
{
    elem=(int *)arr[0]; //initialize to point to first elem of array
    while(elem!=NULL)// continue while not last line
    {
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
    }
}

```


[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

No modifications are required for this code

```
*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
```

```

/*****/

int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
int *arr,*elem=NULL;
int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1, elem10=1;
int elem11=1, elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset, lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsize=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n", papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 31:

```

```

/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline+2;

```


[illegible]


```

*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /****** PAPI STUFF *****/
    int  papi_group=5;
    int  samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /****** */

    int i,j,k, initindex,stride, elemXpage, padsize, pagesize, linesize, elemXline;

```

```

int *arr,*elem=NULL;
int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset,lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsz=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsz)/sizeof(int);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV

```

```

Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
    /***** pmcount group 41 *****/
    Events[0]=0x40000009; //PM_BR_UNCOND
    Events[1]=0x400000d4; //PM_BR_PRED_TA
    Events[2]=0x40000106; //PM_BR_PRED_CR
    Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
    Events[4]=0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 42:
            /***** pmcount group 42 *****/
            Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
            Events[1]=0x4000003f; //PM_GRP_BR_REDIR
            Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
    /***** pmcount group 43 *****/
    Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
    Events[1]=0x40000016; //PM_DTLB_MISS
    Events[2]=0x4000015f; //PM_LD_MISS_L1
    Events[3]=0x400001c0; // PM_LD_REF_L1
    Events[4]=0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
    /***** pmcount group 44 *****/
    Events[0]=0x4000000d; //PM_DATA_FROM_L2
    Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
    Events[2]=0x40000199; //PM_ST_REF_L1
    Events[3]=0x40000198; // PM_ST_MISS_L1
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
    /***** pmcount group 49 *****/
    Events[0]=0x40000010; //PM_DATA_FROM_L3
    Events[1]=0x400000db; //PM_DATA_FROM_LMEM
    Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
    Events[3]=0x40000013; //PM_DATA_FROM_RMEM
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
    /***** pmcount group 78 *****/
    Events[0]=0x40000037; //PM_FPU_FDIV

```

```

Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/****** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/****** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/****** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/****** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/****** END OF PAPI INIT *****/

/***** first elem of each page stores address of first element of first elem of a successive page ****/
stride=elemXline+3;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
}

```


[illegible]

```

asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");

asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");

//
elem2=*(elem+1);
elem=(int *)elem; // load address of first elem of next line

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18lld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

printf("\n\n*****Perf counter counts*****\n");

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F12C1T1 Microbenchmark:

/*** Compilation instructions

Compile the source code to object file with -O3 optimization
No modifications are required for this code

- * Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
- * Think of this array as a multiple of pages and in our case we use 3000 page array
- * Initialize the first element of every cache line to store the address of the first element of the next successive page, the last line of the array will store NULL.

```

*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /******

    int i,j,k, initindex,stride, elemXpage, padsize, pagesize,linesize, elemXline;
    int *arr,*elem=NULL;

```

```

int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset,lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsz=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsz)/sizeof(int);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU

```

```

Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA

```

```

Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/4;
for(i=0;i<(arrsize-stride);i+=stride)
{

```



```

        PAPI_stop(EventSet, values);

        for(i=0;i<NUM_EVENTS;i++)
        {
            printf("%-18ld",values[i]);
        }
        printf("\n");

        printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

        /* free the resources used by PAPI */
        PAPI_shutdown();
        free(arr);
        free(elem);
    }

```

F1I2C2T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

```



```

}

int main(int argc, char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval, number=NUM_EVENTS, Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    int *arr, *elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1, elem10=1;
    int elem11=1, elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n", papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(int); // num of elems in one page
    initindex=1+(padsize/sizeof(int)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS

```

```

*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

        case 42:
            /***** pmcount group 42 *****/
            Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
            Events[1]=0x4000003f; //PM_GRP_BR_REDIR
            Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

    case 43:
        /***** pmcount group 43 *****/
        Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
        Events[1]=0x40000016; //PM_DTLB_MISS
        Events[2]=0x4000015f; //PM_LD_MISS_L1
        Events[3]=0x400001c0; // PM_LD_REF_L1
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

```

```

case 44:
/***** pmcount group 44 *****/
Events[0]=0x400000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN

```

```

Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
    arr[i+1]=1;
}

arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

PAPI_start(EventSet);

for(j=0;j<45200;j++)
{
    elem=(int *)arr[0]; //initialize to point to first elem of array
    while(elem!=NULL &&(elem2!=0))// continue while not last line
    {

        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
    }
}

```

```

asm volatile("nop");

asm volatile("fadd 1,1,4");
asm volatile("fadd 2,2,1");
asm volatile("fadd 3,3,2");
asm volatile("fadd 4,4,3");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
/*
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
*/

elem2=*(elem+1);
elem=(int *)elem; // load address of first elem of next line

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F112C3T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*
*   Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*   Think of this array as a multiple of pages and in our case we use 3000 page array
*   Initialize the first element of every cache line to store the address of the first
*   element of the next successive page, the last line of the array will store NULL.
*   In a loop implement the concept of pointer chasing by first initializing the pointer
*   to the first elem of the first line, the pointer then loads the address of the first
*   element of the second page and so on, this will generate hits misses in the TLB
*   as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

```

```

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0
void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;
    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    int i,j,k, initindex,stride, elemXpage, padsizes, pagesize,linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsizes=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsizes)/sizeof(int);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

```

```

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(int); // num of elems in one page
    initindex=1+(pagesize/sizeof(int)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS
    *****/

    PAPI_library_init(PAPI_VER_CURRENT);

    switch (papi_group)
    {
        case 5:
            /***** pmcount group 5 *****/
            Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
            Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
            Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
            Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 31:
            /***** pmcount group 31 *****/
            Events[0]=0x40000039; //PM_FPU_FULL_CYC
            Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
            Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 41:
            /***** pmcount group 41 *****/
            Events[0]=0x40000009; //PM_BR_UNCOND
            Events[1]=0x400000d4; //PM_BR_PRED_TA
            Events[2]=0x40000106; //PM_BR_PRED_CR

```

```

Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 42:
            /***** pmcount group 42 *****/
            Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 43:
            /***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
        case 44:
            /***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 49:
            /***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 78:
            /***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 79:
            /***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 80:
            /***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM

```



```

Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline+1;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
    arr[i+1]=1;
}

arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

```

F1I3C1T1 Microbenchmark:

```

/**** Compilation instructions
        Compile the source code to object file with -O3 optimization
        No modifications are required for this code

*       Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*       Think of this array as a multiple of pages and in our case we use 3000 page array
*       Initialize the first element of every cache line to store the address of the first
*       element of the next successive page, the last line of the array will store NULL.
*       In a loop implement the concept of pointer chasing by first initializing the pointer
*       to the first elem of the first line, the pointer then loads the address of the first
*       element of the second page and so on, this will generate hits misses in the TLB
*       as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    /***** PAPI STUFF *****/
    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */

```

```

char errstring[PAPI_MAX_STR_LEN];
char name[PAPI_MAX_STR_LEN];
/*****

int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
int *arr,*elem=NULL;
int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset,lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsize=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEMP
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC

```

```

break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

```



```

asm volatile("addi 25,23,0");
asm volatile("addi 26,24,0");

asm volatile("addi 17,15,0");
asm volatile("addi 18,16,0");
asm volatile("addi 19,17,0");
asm volatile("addi 11,11,0");
asm volatile("addi 12,11,0");
asm volatile("addi 13,11,12");
asm volatile("addi 14,12,13");
asm volatile("addi 15,13,14");
asm volatile("addi 16,15,14");

elem=(int *)elem; // load address of first elem of next line
    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F2I1C1T1 Microbenchmark:

```

/**** Compilation instructions
Compile the source code to object file with -O3 optimization
No modifications are required for this code

* Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
* Think of this array as a multiple of pages and in our case we use 3000 page array
* Initialize the first element of every cache line to store the address of the first
* element of the next successive page, the last line of the array will store NULL.
* In a loop implement the concept of pointer chasing by first initializing the pointer
* to the first elem of the first line, the pointer then loads the address of the first
* element of the second page and so on, this will generate hits misses in the TLB
* as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
    }
}

```



```

        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****/

    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }
}

```

```

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(pagesize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 42:

```

```

        /***** pmcount group 42 *****/
        Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
        Events[1]=0x4000003f; //PM_GRP_BR_REDIR
        Events[2]=0x40000016; //PM_FLUSH_BR_MPRED
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 43:
        /***** pmcount group 43 *****/
        Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
        Events[1]=0x40000016; //PM_DTLB_MISS
        Events[2]=0x40000015f; //PM_LD_MISS_L1
        Events[3]=0x4000001c0; // PM_LD_REF_L1
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;
    case 44:
        /***** pmcount group 44 *****/
        Events[0]=0x4000000d; //PM_DATA_FROM_L2
        Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
        Events[2]=0x400000199; //PM_ST_REF_L1
        Events[3]=0x400000198; // PM_ST_MISS_L1
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 49:
        /***** pmcount group 49 *****/
        Events[0]=0x40000010; //PM_DATA_FROM_L3
        Events[1]=0x400000db; //PM_DATA_FROM_LMEM
        Events[2]=0x4000001ae; //PM_DATA_FROM_L2MISS
        Events[3]=0x40000013; //PM_DATA_FROM_RMEM
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 78:
        /***** pmcount group 78 *****/
        Events[0]=0x40000037; //PM_FPU_FDIV
        Events[1]=0x400000dd; //PM_FPU_FMA
        Events[2]=0x400000124; //PM_FPU_FMOV_FEST
        Events[3]=0x4000001b8; //PM_FPU_FEST
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 79:
        /***** pmcount group 79 *****/
        Events[0]=0x40000038; //PM_FPU_1FLOP
        Events[1]=0x400000dc; //PM_FPU_FSQRT
        Events[2]=0x400000125; //PM_FPU_FRSP_FCONV
        Events[3]=0x4000001b9; //PM_FPU_FIN
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 80:
        /***** pmcount group 80 *****/
        Events[0]=0x40000036; //PM_FPU_DENORM
        Events[1]=0x400000de; // PM_FPU_STALL3
        Events[2]=0x40000011c; //PM_FPU0_FIN
        Events[3]=0x400000121; //PM_FPU1_FIN
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

```

```

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page */
stride=elemXline/4;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
}
arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

PAPI_start(EventSet);

for(j=0;j<9000;j++)
{
    elem=(int *)arr[0]; //initialize to point to first elem of array
    while(elem!=NULL)// continue while not last line

```

```

        {
            asm volatile("fadd 1,1,7");
            asm volatile("fadd 2,2,1");
            asm volatile("fadd 3,3,2");
            asm volatile("fadd 4,4,3");
            asm volatile("fadd 5,5,4");
            asm volatile("fadd 6,6,5");
            asm volatile("fadd 7,7,6");
            asm volatile("fadd 8,8,7");
            asm volatile("fadd 9,9,8");
            asm volatile("fadd 10,10,9");
            asm volatile("fadd 11,11,10");
            asm volatile("fadd 12,12,11");
            asm volatile("addi 11,11,0");
            asm volatile("addi 12,12,0");
            asm volatile("addi 13,13,0");

            elem=(int *)elem; // load address of first elem of next line
        }
    }
    /* Stop counting and store the values into the array */
    PAPI_stop(EventSet, values);

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",values[i]);
    }
    printf("\n");

    printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

    /* free the resources used by PAPI */
    PAPI_shutdown();
    free(arr);
    free(elem);
}

```

F21C2T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

```

```

if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
{
    printf("PAPI_read failed and returned %d\n", retval);
    return;
}

if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
{
    printf("PAPI_reset failed and returned %d\n", retval);
    return;
}

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",handlervalues[i]);
}
printf("\n");

return;
}

```

```

int main(int argc,char *argv[])
{

```

```

    /***** PAPI STUFF *****/

```

```

    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

```

```

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

```

```

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

```

```

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

```

```

    int i,j,k, initindex,stride, elemXpage, padsize, pagesize,linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

```

```

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

```

```

    /***** Parse Command line arguments *****/

```

```

    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

```

```

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)

```

```

{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(pagesize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 40:
        /***** pmcount group 40 *****/
        Events[0]=0x400000d1; //PM_TLB_MISS
        Events[1]=0x40000100; //PM_SLB_MISS
        Events[2]=0x40000104; //PM_BR_MPRED_CR
        Events[3]=0x40000105; //PM_BR_MPRED_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL

```

```

Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 42:
        /***** pmcount group 42 *****/
        Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 43:
        /***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
        case 44:
        /***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 49:
        /***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 78:
        /***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 79:
        /***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 80:
        /***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN

```



```

Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline+2;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
    arr[i+1]=1;
}

arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

```

[illegible]


```

    }
    /* Stop counting and store the values into the array */
    PAPI_stop(EventSet, values);

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",values[i]);
    }
    printf("\n");

    printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

    /* free the resources used by PAPI */
    PAPI_shutdown();
    free(arr);
    free(elem);
}

```

F2I2C1T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6
static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
}

```

```

    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval, number=NUM_EVENTS, Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1, elem10=1;
    int elem11=1, elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

    /***** Parse Command line arguments *****/
    if(argc != 2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n", papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

```

```

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(pagesize/sizeof(int)); // Num of elem in pad segment +1

```

```

/*****
PAPI_INITIALIZATION AND EVENT SET ADDITIONS
*****/

```

```

PAPI_library_init(PAPI_VER_CURRENT);

```

```

switch (papi_group)
{

```

```

    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

    case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

        case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

    case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

        case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

    case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS

```

```

Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:

```

```

/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/4;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
}
arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

PAPI_start(EventSet);

for(j=0;j<26000;j++)
{
    elem=(int *)arr[0]; //initialize to point to first elem of array
    while(elem!=NULL)// continue while not last line
    {

        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
        asm volatile("nop");
    }
}

```



```

asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");

asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");

elem=(int *)elem; // load address of first elem of next line
    }
}
/* Stop counting and store the values into the array */
PAPL_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

/* free the resources used by PAPI */
PAPL_shutdown();
free(arr);
free(elem);
}

```

F212C2T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

```

```

if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
{
    printf("PAPI_read failed and returned %d\n", retval);
    return;
}

if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
{
    printf("PAPI_reset failed and returned %d\n", retval);
    return;
}

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18lld",handlervalues[i]);
}
printf("\n");

return;
}

```

```

int main(int argc,char *argv[])
{

```

```

    /***** PAPI STUFF *****/

```

```

    int papi_group=5;

```

```

    int samples_per_sec=1;

```

```

    long THRESHOLD=1;

```

```

    int EventSet=PAPI_NULL;

```

```

    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

```

```

    long_long values[NUM_EVENTS];

```

```

    /*This is where we store the values we read from the eventset */

```

```

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];

```

```

    /* We use number to keep track of the number of events in the EventSet */

```

```

    char errstring[PAPI_MAX_STR_LEN];

```

```

    char name[PAPI_MAX_STR_LEN];

```

```

    /*****

```

```

    int i,j,k, initindex,stride, elemXpage, padsize, pagesize,linesize, elemXline;

```

```

    int *arr,*elem=NULL;

```

```

    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;

```

```

    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;

```

```

    float ctr=0.0;

```

```

    int arrsize, num_of_pages;

```

```

    int offset,lineoffset=0;

```

```

    num_of_pages=3000; //size of the L2 cache

```

```

    pagesize=4096; // 4k size

```

```

    padsize=0; // experimental value

```

```

    linesize=128; //128 bytes line size

```

```

    elemXline=linesize/sizeof(int);

```

```

    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

```

```

    /***** Parse Command line arguments *****/

```

```

    if(argc !=2 )

```

```

    {

```

```

        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80,81, 92> \n");

```

```

        exit(1);

```

```

    }

```

```

    papi_group = atoi(argv[1]);

```

```

    printf("you enter group %d\n",papi_group);

```

```

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)

```

```

{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(pagesize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 40:
        /***** pmcount group 40 *****/
        Events[0]=0x400000d1; //PM_TLB_MISS
        Events[1]=0x40000100; //PM_SLB_MISS
        Events[2]=0x40000104; //PM_BR_MPRED_CR
        Events[3]=0x40000105; //PM_BR_MPRED_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC

```

```

break;

case 42:
    /***** pmcount group 42 *****/
    Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
    Events[1]=0x4000003f; //PM_GRP_BR_REDIR
    Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
    Events[3]=0x40000049; // PM_INST_CMPL
    Events[4]=0x400000bd; //PM_RUN_CYC
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
    /***** pmcount group 43 *****/
    Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
    Events[1]=0x40000016; //PM_DTLB_MISS
    Events[2]=0x4000015f; //PM_LD_MISS_L1
    Events[3]=0x400001c0; // PM_LD_REF_L1
    Events[4]=0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
    /***** pmcount group 44 *****/
    Events[0]=0x4000000d; //PM_DATA_FROM_L2
    Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
    Events[2]=0x40000199; //PM_ST_REF_L1
    Events[3]=0x40000198; // PM_ST_MISS_L1
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
    /***** pmcount group 49 *****/
    Events[0]=0x40000010; //PM_DATA_FROM_L3
    Events[1]=0x400000db; //PM_DATA_FROM_LMEM
    Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
    Events[3]=0x40000013; //PM_DATA_FROM_RMEM
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
    /***** pmcount group 78 *****/
    Events[0]=0x40000037; //PM_FPU_FDIV
    Events[1]=0x400000dd; //PM_FPU_FMA
    Events[2]=0x40000124; //PM_FPU_FMOV_FEST
    Events[3]=0x400001b8; //PM_FPU_FEST
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
    /***** pmcount group 79 *****/
    Events[0]=0x40000038; //PM_FPU_1FLOP
    Events[1]=0x400000dc; //PM_FPU_FSQRT
    Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
    Events[3]=0x400001b9; //PM_FPU_FIN
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
    /***** pmcount group 80 *****/
    Events[0]=0x40000036; //PM_FPU_DENORM
    Events[1]=0x400000de; // PM_FPU_STALL3
    Events[2]=0x4000011c; //PM_FPU0_FIN
    Events[3]=0x40000121; //PM_FPU1_FIN

```

```

Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline*3;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
    arr[i+1]=1;
    arr[i+2]=1;
    arr[i+3]=1;
    arr[i+4]=1;
    arr[i+5]=1;
    arr[i+6]=1;
    arr[i+7]=1;
    arr[i+8]=1;
    arr[i+9]=1;
    arr[i+10]=1;
    arr[i+11]=1;
    arr[i+12]=1;
    arr[i+13]=1;
}

arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1;
arr[arrsize-stride+2]=1;
arr[arrsize-stride+3]=1;
arr[arrsize-stride+4]=1;
arr[arrsize-stride+5]=1;
arr[arrsize-stride+6]=1;
arr[arrsize-stride+7]=1;
arr[arrsize-stride+8]=1;
arr[arrsize-stride+9]=1;

```

```

        arr[arrsize-stride+10]=1;
        arr[arrsize-stride+11]=1;
        arr[arrsize-stride+12]=1;
        arr[arrsize-stride+13]=1;

        /* ***** Start counters ***** */
        THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
        retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
        if(retval !=PAPI_OK)
        {
            printf("overflow call failed with return value %d\n",retval);
            exit(0);
        }

        printf("\n");    for(i=0;i<NUM_EVENTS;i++)
        {
            PAPI_event_code_to_name(Events[i],name);
            printf(" %s\t",name);
        }
        printf("\n");

        PAPI_start(EventSet);

        for(j=0;j<26200;j++)
        {
            elem=(int *)arr[0]; //initialize to point to first elem of array
            while(elem!=NULL &&(elem2!=0 || elem3!=0 || elem4!=0 || elem5!=0 || elem6!=0 || elem7!=0 || elem8!=0 || elem9!=0 ||
elem10!=0 || elem11!=0 || elem12!=0 || elem13!=0))// continue while not last line
            {
                asm volatile("fadd 1,1,1");
                asm volatile("fadd 2,2,1");
                asm volatile("fadd 3,3,2");
                asm volatile("fadd 4,4,3");
                asm volatile("fadd 5,5,4");
                asm volatile("fadd 6,6,5");
                asm volatile("fadd 7,7,7");
                asm volatile("fadd 8,8,8");
                asm volatile("fadd 9,9,9");
                asm volatile("fadd 10,10,10");
                asm volatile("fadd 11,11,11");
                asm volatile("fadd 12,12,12");
                asm volatile("fadd 13,13,13");
                asm volatile("fadd 14,14,14");
                asm volatile("fadd 15,15,15");
                asm volatile("fadd 16,16,16");
                asm volatile("fadd 17,17,17");
                asm volatile("fadd 18,18,18");
                asm volatile("fadd 19,19,19");
                asm volatile("fadd 20,20,20");
                asm volatile("fadd 21,21,21");
                asm volatile("fadd 22,22,22");
                asm volatile("fadd 23,23,23");
                asm volatile("fadd 24,24,24");
                asm volatile("fadd 25,25,25");
                asm volatile("fadd 26,26,26");
                asm volatile("fadd 27,27,27");
                asm volatile("fadd 28,28,28");
                asm volatile("fadd 29,29,29");
                asm volatile("fadd 30,30,30");
                asm volatile("fadd 1,1,1");
                asm volatile("fadd 2,2,1");
                asm volatile("fadd 3,3,2");
                asm volatile("fadd 4,4,3");
                asm volatile("fadd 5,5,4");
                asm volatile("fadd 6,6,5");
                asm volatile("fadd 7,7,7");
                asm volatile("fadd 8,8,8");
                asm volatile("fadd 9,9,9");
                asm volatile("fadd 10,10,10");
                asm volatile("fadd 11,11,11");
                asm volatile("fadd 12,12,12");
            }
        }

```

```

asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 21,21,21");
asm volatile("fadd 22,22,22");
asm volatile("fadd 23,23,23");
asm volatile("fadd 24,24,24");
asm volatile("fadd 25,25,25");
asm volatile("fadd 26,26,26");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");
asm volatile("addi 25,25,0");
asm volatile("addi 26,26,0");
asm volatile("addi 27,27,0");
asm volatile("addi 28,28,0");
asm volatile("addi 29,29,0");
asm volatile("addi 30,30,0");
asm volatile("addi 31,31,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);
elem=(int *)elem; // load address of first elem of next line

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F2I3C1T1 Microbenchmark:

```
/**** Compilation instructions
        Compile the source code to object file with -O3 optimization
        No modifications are required for this code

*
*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */
}
```



```

int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
/* We use number to keep track of the number of events in the EventSet */
char errstring[PAPI_MAX_STR_LEN];
char name[PAPI_MAX_STR_LEN];
/*****

int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
int *arr,*elem=NULL;
int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset,lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsize=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsize/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
Events[4]=0x40000049; // PM_INST_CMPL

```

```

Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM

```

```

Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

```



```

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n",(arrsize/stride)*j);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F31C1T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
}

```

```

    }
    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval, number=NUM_EVENTS, Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****
    int temp=0;
    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1, elem10=1.1;
    float elem11=1.1, elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(float);

    *****/ Parse Command line arguments *****/
    if(argc != 2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n", papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

```

```

elemXpage=pagesize/sizeof(float); // num of elems in one page
initindex=1+(pagesize/sizeof(float)); // Num of elem in pad segment +1

```

```

/*****
PAPI_INITIALIZATION AND EVENT SET ADDITIONS
*****/

```

```

PAPI_library_init(PAPI_VER_CURRENT);

```

```

switch (papi_group)
{

```

```

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

```

```

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

```

```

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

```

```

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

```

```

        case 42:
            /***** pmcount group 42 *****/
            Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
            Events[1]=0x4000003f; //PM_GRP_BR_REDIR
            Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

```

```

    case 43:
        /***** pmcount group 43 *****/
        Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
        Events[1]=0x40000016; //PM_DTLB_MISS

```

```

Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:

```



```

/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/2;

for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=1.1;
    arr[i+1]=1.1;
    arr[i+2]=1.1;
    arr[i+3]=1.1;
    arr[i+4]=1.1;
    arr[i+5]=1.1;
    arr[i+6]=1.1;
    arr[i+7]=1.1;
    arr[i+8]=1.1;
    arr[i+9]=1.1;
    arr[i+10]=1.1;
    arr[i+11]=1.1;
    arr[i+12]=1.1;
    arr[i+13]=1.1;
}
arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;
arr[arrsize-stride+2]=1.1;
arr[arrsize-stride+3]=1.1;
arr[arrsize-stride+4]=1.1;
arr[arrsize-stride+5]=1.1;
arr[arrsize-stride+6]=1.1;
arr[arrsize-stride+7]=1.1;
arr[arrsize-stride+8]=1.1;
arr[arrsize-stride+9]=1.1;
arr[arrsize-stride+10]=1.1;
arr[arrsize-stride+11]=1.1;
arr[arrsize-stride+12]=1.1;
arr[arrsize-stride+13]=1.1;

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

```

```

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

    PAPI_start(EventSet);

    for(j=0;j<14500;j++)
    {
        i=0;
        elem=(float *) (arr+0); //initialize to point to first elem of array
        for(i=0;i<(arrsize-2*stride);i+=stride)
        {
asm volatile("fadd 1,1,16");
asm volatile("fadd 2,2,1");
asm volatile("fadd 3,3,2");
asm volatile("fadd 4,4,3");
asm volatile("fadd 5,5,4");
asm volatile("fadd 6,6,5");
asm volatile("fadd 7,7,6");
asm volatile("fadd 8,8,7");
asm volatile("fadd 9,9,8");
asm volatile("fadd 10,10,9");
asm volatile("fadd 11,11,10");
asm volatile("fadd 12,12,11");
asm volatile("fadd 13,13,12");
asm volatile("fadd 14,14,13");
asm volatile("fadd 15,15,14");
asm volatile("fadd 16,16,15");
/* asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
*/

asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,1");
asm volatile("fadd 3,3,2");
asm volatile("fadd 4,4,3");
asm volatile("fadd 5,5,4");
asm volatile("fadd 6,6,5");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 21,21,21");
asm volatile("fadd 22,22,22");
asm volatile("fadd 23,23,23");
asm volatile("fadd 24,24,24");
asm volatile("fadd 25,25,25");
asm volatile("fadd 26,26,26");
asm volatile("fadd 27,27,27");
asm volatile("fadd 28,28,28");
asm volatile("fadd 29,29,29");
asm volatile("fadd 30,30,30");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");

```

```
elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);
```

```

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

```

```
printf("\n\n # of iters = %d\n", (arrsize/stride*));
    printf("%f %f %f %f %f %f %f %f %f %f %f %f %f %f\n", elem2, elem3, elem4, elem5, elem6, elem7, elem8, elem9, elem10, elem10, elem11,
```

```
/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
```

}

```

/**** Compilation instructions

```

```
* Basic algorithm is to allocate an array the size much larger than # of pages TLB and ERAT can store
* Think of this array as a multiple of pages and in our case we use 3000 page array
* Initialize the first element of every cache line to store the address of the first
* element of the next successive page, the last line of the array will store NULL.
* In a loop implement the concept of pointer chasing by first initializing the pointer
* to the first elem of the first line, the pointer then loads the address of the first
* element of the second page and so on, this will generate hits misses in the TLB
* as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
```

```
#define NUM_EVENTS 6
```

```
#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0
```

```

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****/
    int temp=0;
    int i,j,k, initindex,stride, elemXpage, padsizes, pagesize,linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1,elem10=1.1;
    float elem11=1.1,elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsizes=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsizes)/sizeof(float);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);
}

```

```

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(float); // num of elems in one page
    initindex=1+(pagesize/sizeof(float)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS
    *****/

    PAPI_library_init(PAPI_VER_CURRENT);

    switch (papi_group)
    {
        case 5:
            /***** pmcount group 5 *****/
            Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
            Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
            Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
            Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 31:
            /***** pmcount group 31 *****/
            Events[0]=0x40000039; //PM_FPU_FULL_CYC
            Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
            Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

        case 41:
            /***** pmcount group 41 *****/
            Events[0]=0x40000009; //PM_BR_UNCOND
            Events[1]=0x400000d4; //PM_BR_PRED_TA

```

```

Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/

```

```

Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page */
stride=elemXline/2;

for(i=0;i<(arrsize-stride);i+=stride)
{
arr[i]=1.1;
arr[i+1]=1.1;

arr[i+2]=1.1;
arr[i+3]=1.1;
arr[i+4]=1.1;
arr[i+5]=1.1;
arr[i+6]=1.1;
arr[i+7]=1.1;
arr[i+8]=1.1;
arr[i+9]=1.1;
arr[i+10]=1.1;
arr[i+11]=1.1;
arr[i+12]=1.1;
arr[i+13]=1.1;

}

arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;
arr[arrsize-stride+2]=1.1;
arr[arrsize-stride+3]=1.1;

```


[illegible]

```
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
```

[illegible]

```

asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);

elem=(float *) (arr+i+stride); // load address of first elem of next line
i+=stride;

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}

printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*i);

```



```
long THRESHOLD=1;
```

```
int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/
```

```
long_long values[NUM_EVENTS];
/*This is where we store the values we read from the eventset */
```

```
int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
/* We use number to keep track of the number of events in the EventSet */
char errstring[PAPI_MAX_STR_LEN];
char name[PAPI_MAX_STR_LEN];
/*****
```

```
int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
int *arr,*elem=NULL;
int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1, elem10=1;
int elem11=1, elem12=1, elem13=1, elem14=1, elem15=1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset, lineoffset=0;
num_of_pages=3000; //size of the L2 cache
pagesize=4096; // 4k size
padsize=0; // experimental value
```

```
linesize=128; //128 bytes line size
elemXline=linesize/sizeof(int);
arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);
```

```
/***** Parse Command line arguments *****/
```

```
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);
```

```
if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}
```

```
/***** Done parsing *****/
```

```
arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}
```

```
elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsize/sizeof(int)); // Num of elem in pad segment +1
```

```
/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/
```

```
PAPI_library_init(PAPI_VER_CURRENT);
```

```

switch (papi_group)
{

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 40:
        /***** pmcount group 40 *****/
        Events[0]=0x400000d1; //PM_TLB_MISS
        Events[1]=0x40000100; //PM_SLB_MISS
        Events[2]=0x40000104; //PM_BR_MPRED_CR
        Events[3]=0x40000105; //PM_BR_MPRED_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA
        Events[2]=0x40000106; //PM_BR_PRED_CR
        Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 42:
        /***** pmcount group 42 *****/
        Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
        Events[1]=0x4000003f; //PM_GRP_BR_REDIR
        Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 43:
        /***** pmcount group 43 *****/
        Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
        Events[1]=0x40000016; //PM_DTLB_MISS
        Events[2]=0x4000015f; //PM_LD_MISS_L1
        Events[3]=0x400001c0; // PM_LD_REF_L1
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 44:
        /***** pmcount group 44 *****/
        Events[0]=0x4000000d; //PM_DATA_FROM_L2
        Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
        Events[2]=0x40000199; //PM_ST_REF_L1

```

```

Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:

```


[illegible]

[illegible]


```
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
```

```

asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");

elem=(int *)elem; // load address of first elem of next line

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

```

F3I2C2T1 Microbenchmark:

```

/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)

```

```

    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /******

    int i,j,k, initindex,stride, elemXpage, padsize, pagesize,linesize, elemXline;
    int *arr,*elem=NULL;
    int elem2=1, elem3=1, elem4=1, elem5=1, elem6=1, elem7=1, elem8=1, elem9=1,elem10=1;
    int elem11=1,elem12=1, elem13=1, elem14=1, elem15=1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(int);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(int);

    /****** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80,81, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /****** Done parsing *****/

```

```

arr =(int *) malloc(arrsize*sizeof(int)); // create the array of integers
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(int); // num of elems in one page
initindex=1+(padsz/sizeof(int)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{

case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPREd_CR
Events[3]=0x40000105; //PM_BR_MPREd_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPREd
Events[3]=0x40000049; // PM_INST_CMPL

```



```

Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x40000015f; //PM_LD_MISS_L1
Events[3]=0x4000001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x400000199; //PM_ST_REF_L1
Events[3]=0x400000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x4000001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x400000124; //PM_FPU_FMOV_FEST
Events[3]=0x4000001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x400000125; //PM_FPU_FRSP_FCONV
Events[3]=0x4000001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x40000011c; //PM_FPU0_FIN
Events[3]=0x400000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF

```

```

Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page */
stride=elemXline*3;
for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=(int) (arr+i+stride);
    arr[i+1]=1;
    arr[i+2]=1;
    arr[i+3]=1;
    arr[i+4]=1;
    arr[i+5]=1;
    arr[i+6]=1;
    arr[i+7]=1;
    arr[i+8]=1;
    arr[i+9]=1;
    arr[i+10]=1;
    arr[i+11]=1;
    arr[i+12]=1;
    arr[i+13]=1;
}

arr[arrsize-stride]=(int) NULL; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1;
arr[arrsize-stride+2]=1;
arr[arrsize-stride+3]=1;
arr[arrsize-stride+4]=1;
arr[arrsize-stride+5]=1;
arr[arrsize-stride+6]=1;
arr[arrsize-stride+7]=1;
arr[arrsize-stride+8]=1;
arr[arrsize-stride+9]=1;
arr[arrsize-stride+10]=1;
arr[arrsize-stride+11]=1;
arr[arrsize-stride+12]=1;
arr[arrsize-stride+13]=1;

/***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)

```

```

{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

PAPI_start(EventSet);

//
    for(j=0;j<26200;j++)
    for(j=0;j<19000;j++)
    {
        elem=(int *)arr[0]; //initialize to point to first elem of array
        while(elem!=NULL &&(elem2!=0 || elem3!=0 || elem4!=0 || elem5!=0 || elem6!=0 || elem7!=0 || elem8!=0 || elem9!=0 ||
elem10!=0 || elem11!=0 || elem12!=0 || elem13!=0))// continue while not last line
        {
            asm volatile("fadd 1,1,1");
            asm volatile("fadd 2,2,1");
            asm volatile("fadd 3,3,2");
            asm volatile("fadd 4,4,3");
            asm volatile("fadd 5,5,4");
            asm volatile("fadd 6,6,5");
            asm volatile("fadd 7,7,7");
            asm volatile("fadd 8,8,8");
            asm volatile("fadd 9,9,9");
            asm volatile("fadd 10,10,10");
            asm volatile("fadd 11,11,11");
            asm volatile("fadd 12,12,12");
            asm volatile("fadd 13,13,13");
            asm volatile("fadd 14,14,14");
            asm volatile("fadd 15,15,15");
            asm volatile("fadd 16,16,16");
            asm volatile("fadd 17,17,17");
            asm volatile("fadd 18,18,18");
            asm volatile("fadd 19,19,19");
            asm volatile("fadd 20,20,20");
            asm volatile("fadd 21,21,21");
            asm volatile("fadd 22,22,22");
            asm volatile("fadd 23,23,23");
            asm volatile("fadd 24,24,24");
            asm volatile("fadd 25,25,25");
            asm volatile("fadd 26,26,26");
            asm volatile("fadd 27,27,27");
            asm volatile("fadd 28,28,28");
            asm volatile("fadd 29,29,29");
            asm volatile("fadd 30,30,30");
            asm volatile("fadd 1,1,1");
            asm volatile("fadd 2,2,1");
            asm volatile("fadd 3,3,2");
            asm volatile("fadd 4,4,3");
            asm volatile("fadd 5,5,4");
            asm volatile("fadd 6,6,5");
            asm volatile("fadd 7,7,7");
            asm volatile("fadd 8,8,8");
            asm volatile("fadd 9,9,9");
            asm volatile("fadd 10,10,10");
            asm volatile("fadd 11,11,11");
            asm volatile("fadd 12,12,12");
            asm volatile("fadd 13,13,13");
            asm volatile("fadd 14,14,14");
            asm volatile("fadd 15,15,15");
            asm volatile("fadd 16,16,16");
            asm volatile("fadd 17,17,17");
            asm volatile("fadd 18,18,18");
            asm volatile("fadd 19,19,19");
            asm volatile("fadd 20,20,20");
        }
    }
}

```

```

asm volatile("fadd 21,21,21");
asm volatile("fadd 22,22,22");
asm volatile("fadd 23,23,23");
asm volatile("fadd 24,24,24");
asm volatile("fadd 25,25,25");
asm volatile("fadd 26,26,26");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,1");
asm volatile("fadd 3,3,2");
asm volatile("fadd 4,4,3");
asm volatile("fadd 5,5,4");
asm volatile("fadd 6,6,5");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 21,21,21");
asm volatile("fadd 22,22,22");
asm volatile("fadd 23,23,23");
asm volatile("fadd 24,24,24");
asm volatile("fadd 25,25,25");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,1");
asm volatile("fadd 3,3,2");
asm volatile("fadd 4,4,3");
asm volatile("fadd 5,5,4");
asm volatile("fadd 6,6,5");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 21,21,21");
asm volatile("fadd 22,22,22");
asm volatile("fadd 23,23,23");
asm volatile("fadd 24,24,24");
asm volatile("fadd 25,25,25");
asm volatile("fadd 26,26,26");
asm volatile("fadd 27,27,27");
asm volatile("fadd 28,28,28");
asm volatile("fadd 29,29,29");
asm volatile("fadd 30,30,30");

```

```

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");
asm volatile("addi 25,25,0");

```

```

asm volatile("addi 26,26,0");
asm volatile("addi 27,27,0");
asm volatile("addi 28,28,0");
asm volatile("addi 29,29,0");
asm volatile("addi 30,30,0");
asm volatile("addi 31,31,0");
asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);
elem=(int *)elem; // load address of first elem of next line

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18lld",values[i]);
}
printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);
free(elem);
}

F41C1T1 Microbenchmark:
/**** Compilation instructions
    Compile the source code to object file with -O3 optimization
    No modifications are required for this code

*
*   Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*   Think of this array as a multiple of pages and in our case we use 3000 page array
*   Initialize the first element of every cache line to store the address of the first
*   element of the next successive page, the last line of the array will store NULL.
*   In a loop implement the concept of pointer chasing by first initializing the pointer
*   to the first elem of the first line, the pointer then loads the address of the first
*   element of the second page and so on, this will generate hits misses in the TLB
*   as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlers[values[0]];
static int EventSet=PAPI_NULL;

```

```

/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /******
    int temp=0;
    int i,j,k, initindex,stride, elemXpage, padsize, pagesize,linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1,elem10=1.1;
    float elem11=1.1,elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(float);

    /****** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
    }
}

```

```

        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(float); // num of elems in one page
    initindex=1+(padsz/sizeof(float)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS
    *****/

    PAPI_library_init(PAPI_VER_CURRENT);

    switch (papi_group)
    {

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;
    }

```

```

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```



```

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/4;

for(i=0;i<(arrsize-stride);i+=stride)
{
arr[i]=1.1;
arr[i+1]=1.1;
arr[i+2]=1.1;
arr[i+3]=1.1;
arr[i+4]=1.1;
arr[i+5]=1.1;
arr[i+6]=1.1;
arr[i+7]=1.1;
arr[i+8]=1.1;
arr[i+9]=1.1;
arr[i+10]=1.1;
arr[i+11]=1.1;
arr[i+12]=1.1;
arr[i+13]=1.1;
}

```

[illegible]

[illegible]

[illegible]

[illegible]

```

asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
/* elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);
*/

elem=(float *) (arr+i+stride); // load address of first elem of next line
i+=stride;

}

}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

```



```

}

int main(int argc, char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval, number=NUM_EVENTS, Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****
    int temp=0;
    int i,j,k, initindex, stride, elemXpage, padsizes, pagesize, linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1, elem10=1.1;
    float elem11=1.1, elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset, lineoffset=0;
    // num_of_pages=3000; //size of the L2 cache
    num_of_pages=200; //size of the L2 cache
    pagesize=4096; // 4k size
    padsizes=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsizes)/sizeof(float);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n", papi_group);

    if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
    && papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(float); // num of elems in one page
    initindex=1+(padsizes/sizeof(float)); // Num of elem in pad segment +1

```



```

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

    case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC

```

```

break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x400000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN

```

```

Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/* first elem of each page stores address of first element of first elem of a successive page */
// stride=elemXline/2;
stride=elemXline;

for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=1.1;
    arr[i+1]=1.1;
    arr[i+2]=1.1;
    arr[i+3]=1.1;
    arr[i+4]=1.1;
    arr[i+5]=1.1;
    arr[i+6]=1.1;
    arr[i+7]=1.1;
    arr[i+8]=1.1;
    arr[i+9]=1.1;
    arr[i+10]=1.1;
    arr[i+11]=1.1;
    arr[i+12]=1.1;
    arr[i+13]=1.1;
}

arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;
arr[arrsize-stride+2]=1.1;
arr[arrsize-stride+3]=1.1;
arr[arrsize-stride+4]=1.1;
arr[arrsize-stride+5]=1.1;
arr[arrsize-stride+6]=1.1;
arr[arrsize-stride+7]=1.1;
arr[arrsize-stride+8]=1.1;
arr[arrsize-stride+9]=1.1;
arr[arrsize-stride+10]=1.1;
arr[arrsize-stride+11]=1.1;
arr[arrsize-stride+12]=1.1;
arr[arrsize-stride+13]=1.1;

/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);

```

```
    printf(" %s\t",name);
}
printf("\n");
```

```
PAPI_start(EventSet);
```

```
//for(j=0;j<320000;j++)
for(j=0;j<280000;j++)
{
```

```
i=0;
elem=(float *) (arr+0); //initialize to point to first elem of array
for(i=0;i<(arrsize-2*stride);i+=stride)
{
```

[illegible]

[illegible]

[illegible]

```

asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
/*    asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");

*/

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");
asm volatile("addi 11,11,0");
asm volatile("addi 13,13,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);

elem=(float *) (arr+i+stride); // load address of first elem of next line
i+=stride;

    }
}
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}

printf("\n");

printf("\n\n # of iters = %d\n",(arrsize/stride)*j);
printf("%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f\n", elem2, elem3, elem4, elem5, elem6, elem7, elem8, elem9, elem10, elem11,
elem12, elem13);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);

}

F41C3T1 Microbenchmark:
/**** Compilation instructions

```


Compile the source code to object file with -O3 optimization
No modifications are required for this code

```
*      Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
*      Think of this array as a multiple of pages and in our case we use 3000 page array
*      Initialize the first element of every cache line to store the address of the first
*      element of the next successive page, the last line of the array will store NULL.
*      In a loop implement the concept of pointer chasing by first initializing the pointer
*      to the first elem of the first line, the pointer then loads the address of the first
*      element of the second page and so on, this will generate hits misses in the TLB
*      as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc, char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
```

```

char name[PAPI_MAX_STR_LEN];
/*****/
int temp=0;
int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
float *arr,*elem=NULL;
float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1, elem10=1.1;
float elem11=1.1, elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
float ctr=0.0;
int arrsize, num_of_pages;
int offset, lineoffset=0;
// num_of_pages=3000; //size of the L2 cache
num_of_pages=200; //size of the L2 cache
pagesize=4096; // 4k size
padsize=0; // experimental value

linesize=128; //128 bytes line size
elemXline=linesize/sizeof(float);
arrsize=((pagesize*num_of_pages) + padsize)/sizeof(float);

/***** Parse Command line arguments *****/
if(argc !=2 )
{
    printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
    exit(1);
}
papi_group = atoi(argv[1]);
printf("you enter group %d\n", papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44, 49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(float); // num of elems in one page
initindex=1+(padsize/sizeof(float)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
case 5:
/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

case 31:
/***** pmcount group 31 *****/
Events[0]=0x40000039; //PM_FPU_FULL_CYC
Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 40:
/***** pmcount group 40 *****/
Events[0]=0x400000d1; //PM_TLB_MISS
Events[1]=0x40000100; //PM_SLB_MISS
Events[2]=0x40000104; //PM_BR_MPRED_CR
Events[3]=0x40000105; //PM_BR_MPRED_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 41:
/***** pmcount group 41 *****/
Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

```

```

        /**** first elem of each page stores address of first element of first elem of a successive page **/
//    stride=elemXline/2;
stride=elemXline;

    for(i=0;i<(arrsize-stride);i+=stride)
    {
        arr[i]=1.1;
        arr[i+1]=1.1;
        arr[i+2]=1.1;
        arr[i+3]=1.1;
        arr[i+4]=1.1;
        arr[i+5]=1.1;
        arr[i+6]=1.1;
        arr[i+7]=1.1;
        arr[i+8]=1.1;
        arr[i+9]=1.1;
        arr[i+10]=1.1;
        arr[i+11]=1.1;
        arr[i+12]=1.1;
        arr[i+13]=1.1;
    }
    arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;
arr[arrsize-stride+2]=1.1;
arr[arrsize-stride+3]=1.1;
arr[arrsize-stride+4]=1.1;
arr[arrsize-stride+5]=1.1;
arr[arrsize-stride+6]=1.1;
arr[arrsize-stride+7]=1.1;
arr[arrsize-stride+8]=1.1;
arr[arrsize-stride+9]=1.1;
arr[arrsize-stride+10]=1.1;
arr[arrsize-stride+11]=1.1;
arr[arrsize-stride+12]=1.1;
arr[arrsize-stride+13]=1.1;

    /* ***** Start counters ***** */
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf(" %s\t",name);
}
printf("\n");

    PAPI_start(EventSet);

    for(j=0;j<280000;j++)
    {
        i=0;
        elem=(float *) (arr+0); //initialize to point to first elem of array
        for(i=0;i<(arrsize-2*stride);i+=stride)
        {
            asm volatile("nop");
            asm volatile("nop");
            asm volatile("nop");
            asm volatile("nop");
            asm volatile("nop");
            asm volatile("nop");
            asm volatile("nop");
        }
    }

```

[illegible]

[illegible]


```

asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
/*
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
*/

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");

```



```

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /***** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****/
    int temp=0;
    int i,j,k, initindex,stride, elemXpage, padsizes, pagesize,linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1,elem10=1.1;
    float elem11=1.1,elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsizes=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsizes)/sizeof(float);

    /***** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
        exit(1);
    }
    papi_group = atoi(argv[1]);
    printf("you enter group %d\n",papi_group);
}

```

```

    if(papi_group!=5 &&papi_group!=31 &&papi_group!=40 &&papi_group!=41 &&papi_group!=42 && papi_group!=43 && papi_group!=44 &&papi_group!=49
    &&papi_group!=78 &&papi_group!=79 && papi_group!=80 &&papi_group!=81 && papi_group!=92)
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
        exit(1);
    }

    /***** Done parsing *****/

    arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
    if(arr==NULL)
    {
        printf("could not allocate memory \n");
        exit(1);
    }

    elemXpage=pagesize/sizeof(float); // num of elems in one page
    initindex=1+(padsz/sizeof(float)); // Num of elem in pad segment +1

    /*****
    PAPI INITIALIZATION AND EVENT SET ADDITIONS
    *****/

    PAPI_library_init(PAPI_VER_CURRENT);

    switch (papi_group)
    {

    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

        case 40:
            /***** pmcount group 40 *****/
            Events[0]=0x400000d1; //PM_TLB_MISS
            Events[1]=0x40000100; //PM_SLB_MISS
            Events[2]=0x40000104; //PM_BR_MPRED_CR
            Events[3]=0x40000105; //PM_BR_MPRED_TA
            Events[4]=0x40000049; // PM_INST_CMPL
            Events[5]=0x400000bd; //PM_RUN_CYC
            break;

    case 41:
        /***** pmcount group 41 *****/
        Events[0]=0x40000009; //PM_BR_UNCOND
        Events[1]=0x400000d4; //PM_BR_PRED_TA

```

```

Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 42:
/***** pmcount group 42 *****/
Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
Events[1]=0x4000003f; //PM_GRP_BR_REDIR
Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
Events[3]=0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
/***** pmcount group 43 *****/
Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
Events[1]=0x40000016; //PM_DTLB_MISS
Events[2]=0x4000015f; //PM_LD_MISS_L1
Events[3]=0x400001c0; // PM_LD_REF_L1
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
/***** pmcount group 44 *****/
Events[0]=0x4000000d; //PM_DATA_FROM_L2
Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
Events[2]=0x40000199; //PM_ST_REF_L1
Events[3]=0x40000198; // PM_ST_MISS_L1
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
/***** pmcount group 49 *****/
Events[0]=0x40000010; //PM_DATA_FROM_L3
Events[1]=0x400000db; //PM_DATA_FROM_LMEM
Events[2]=0x400001ae; //PM_DATA_FROM_L2MISS
Events[3]=0x40000013; //PM_DATA_FROM_RMEM
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
/***** pmcount group 78 *****/
Events[0]=0x40000037; //PM_FPU_FDIV
Events[1]=0x400000dd; //PM_FPU_FMA
Events[2]=0x40000124; //PM_FPU_FMOV_FEST
Events[3]=0x400001b8; //PM_FPU_FEST
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
/***** pmcount group 79 *****/
Events[0]=0x40000038; //PM_FPU_1FLOP
Events[1]=0x400000dc; //PM_FPU_FSQRT
Events[2]=0x40000125; //PM_FPU_FRSP_FCONV
Events[3]=0x400001b9; //PM_FPU_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 80:
/***** pmcount group 80 *****/

```

```

Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page */
stride=elemXline/4;

for(i=0;i<(arrsize-stride);i+=stride)
{
arr[i]=1.1;
arr[i+1]=1.1;

arr[i+2]=1.1;
arr[i+3]=1.1;
arr[i+4]=1.1;
arr[i+5]=1.1;
arr[i+6]=1.1;
arr[i+7]=1.1;
arr[i+8]=1.1;
arr[i+9]=1.1;
arr[i+10]=1.1;
arr[i+11]=1.1;
arr[i+12]=1.1;
arr[i+13]=1.1;

}

arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;
arr[arrsize-stride+2]=1.1;
arr[arrsize-stride+3]=1.1;

```


[illegible]

[illegible]

[illegible]

```

asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");

```

```

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");
asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");
asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");

```

```

asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);
elem8=*(elem+7);
elem9=*(elem+8);
/* elem10=*(elem+9);
elem11=*(elem+10);
elem12=*(elem+11);
elem13=*(elem+12);
*/

elem=(float *) (arr+i+stride); // load address of first elem of next line
i+=stride;

}

/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

for(i=0;i<NUM_EVENTS;i++)
{
    printf("%-18ld",values[i]);
}

printf("\n");

printf("\n\n # of iters = %d\n", (arrsize/stride)*j);
printf("%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f\n", elem2, elem3, elem4, elem5, elem6, elem7, elem8, elem9, elem10, elem10, elem11,
elem12, elem13);

/* free the resources used by PAPI */
PAPI_shutdown();
free(arr);

}

```

F51C1T1 Microbenchmark:

```

/**** Compilation instructions
Compile the source code to object file with -O3 optimization
No modifications are required for this code

* Basic algorithm is to allocate an array the size much larger then # of pages TLB and ERAT can store
* Think of this array as a multiple of pages and in our case we use 3000 page array
* Initialize the first element of every cache line to store the address of the first
* element of the next successive page, the last line of the array will store NULL.
* In a loop implement the concept of pointer chasing by first initializing the pointer
* to the first elem of the first line, the pointer then loads the address of the first
* element of the second page and so on, this will generate hits misses in the TLB
* as a side effect it also misses L1 and L2 due to the large stride, all get hits in L3.
*/
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "papi.h" /* This needs to be included every time you use PAPI */

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

```

```

#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18ld",handlervalues[i]);
    }
    printf("\n");

    return;
}

int main(int argc,char *argv[])
{
    /****** PAPI STUFF *****/
    int papi_group=5;
    int samples_per_sec=1;
    long THRESHOLD=1;

    int EventSet=PAPI_NULL;
    /*must be initialized to PAPI_NULL before calling PAPI_create_event*/

    long_long values[NUM_EVENTS];
    /*This is where we store the values we read from the eventset */

    int retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /******
    int temp=0;
    int i,j,k, initindex, stride, elemXpage, padsize, pagesize, linesize, elemXline;
    float *arr,*elem=NULL;
    float elem2=1.1, elem3=1.1, elem4=1.1, elem5=1.1, elem6=1.1, elem7=1.1, elem8=1.1, elem9=1.1,elem10=1.1;
    float elem11=1.1,elem12=1.1, elem13=1.1, elem14=1.1, elem15=1.1;
    float ctr=0.0;
    int arrsize, num_of_pages;
    int offset,lineoffset=0;
    num_of_pages=3000; //size of the L2 cache
    pagesize=4096; // 4k size
    padsize=0; // experimental value

    linesize=128; //128 bytes line size
    elemXline=linesize/sizeof(float);
    arrsize=((pagesize*num_of_pages) + padsize)/sizeof(float);

    /****** Parse Command line arguments *****/
    if(argc !=2 )
    {
        printf("You must Enter Papi Group to monitor, choose from <5, 40, 42, 43, 44, 80, 81, 92> \n");
        exit(1);
    }
}

```

```

papi_group = atoi(argv[1]);
printf("you enter group %d\n",papi_group);

if(papi_group!=5 && papi_group!=31 && papi_group!=40 && papi_group!=41 && papi_group!=42 && papi_group!=43 && papi_group!=44 && papi_group!=49
&& papi_group!=78 && papi_group!=79 && papi_group!=80 && papi_group!=81 && papi_group!=92)
{
    printf("You must Enter Papi Group to monitor, choose from <5, 31, 40, 41, 42, 43, 44,49, 78, 79, 80, 81, 92> \n");
    exit(1);
}

/***** Done parsing *****/

arr =(float *) malloc(arrsize*sizeof(float)); // create the array of floats
if(arr==NULL)
{
    printf("could not allocate memory \n");
    exit(1);
}

elemXpage=pagesize/sizeof(float); // num of elems in one page
initindex=1+(pagesize/sizeof(float)); // Num of elem in pad segment +1

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPRED
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 31:
        /***** pmcount group 31 *****/
        Events[0]=0x40000039; //PM_FPU_FULL_CYC
        Events[1]=0x400000d6; //PM_CMPLU_STALL_FDIV
        Events[2]=0x400001b5; //PM_CMPLU_STALL_FPU
        Events[3]=0x40000049; // PM_INST_CMPL
        Events[4]=0x400000bd; //PM_RUN_CYC
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 40:
        /***** pmcount group 40 *****/
        Events[0]=0x400000d1; //PM_TLB_MISS
        Events[1]=0x40000100; //PM_SLB_MISS
        Events[2]=0x40000104; //PM_BR_MPRED_CR
        Events[3]=0x40000105; //PM_BR_MPRED_TA
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 41:
        /***** pmcount group 41 *****/

```

```

Events[0]=0x40000009; //PM_BR_UNCOND
Events[1]=0x400000d4; //PM_BR_PRED_TA
Events[2]=0x40000106; //PM_BR_PRED_CR
Events[3]=0x400001b2; //PM_BR_PRED_CR_TA
Events[4]=0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

        case 42:
            /***** pmcount group 42 *****/
            Events[0]=0x40000040; //PM_GRP_BR_REDIR_NONSPEC_
            Events[1]=0x4000003f; //PM_GRP_BR_REDIR
            Events[2]=0x40000116; //PM_FLUSH_BR_MPRED
            Events[3]=0x40000049; // PM_INST_CMPL
            Events[4]=0x400000bd; //PM_RUN_CYC
            Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 43:
    /***** pmcount group 43 *****/
    Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
    Events[1]=0x40000016; //PM_DTLB_MISS
    Events[2]=0x40000015f; //PM_LD_MISS_L1
    Events[3]=0x4000001c0; // PM_LD_REF_L1
    Events[4]=0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;
case 44:
    /***** pmcount group 44 *****/
    Events[0]=0x4000000d; //PM_DATA_FROM_L2
    Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
    Events[2]=0x400000199; //PM_ST_REF_L1
    Events[3]=0x400000198; // PM_ST_MISS_L1
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 49:
    /***** pmcount group 49 *****/
    Events[0]=0x40000010; //PM_DATA_FROM_L3
    Events[1]=0x400000db; //PM_DATA_FROM_LMEM
    Events[2]=0x4000001ae; //PM_DATA_FROM_L2MISS
    Events[3]=0x40000013; //PM_DATA_FROM_RMEM
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 78:
    /***** pmcount group 78 *****/
    Events[0]=0x40000037; //PM_FPU_FDIV
    Events[1]=0x400000dd; //PM_FPU_FMA
    Events[2]=0x400000124; //PM_FPU_FMOV_FEST
    Events[3]=0x4000001b8; //PM_FPU_FEST
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 79:
    /***** pmcount group 79 *****/
    Events[0]=0x40000038; //PM_FPU_1FLOP
    Events[1]=0x400000dc; //PM_FPU_FSQRT
    Events[2]=0x400000125; //PM_FPU_FRSP_FCONV
    Events[3]=0x4000001b9; //PM_FPU_FIN
    Events[4]= 0x40000049; // PM_INST_CMPL
    Events[5]=0x400000bd; //PM_RUN_CYC
break;

```

```

case 80:
/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 81:
/***** pmcount group 81 *****/
Events[0]=0x4000003a; //PM_FPU_SINGLE
Events[1]=0x400000df; //PM_FPU_STF
Events[2]=0x400001c1; //PM_LSU_LDF
Events[3]= 0x40000049; // PM_INST_CMPL
Events[4]=0x400000bd; //PM_RUN_CYC
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);

/***** END OF PAPI INIT *****/

/**** first elem of each page stores address of first element of first elem of a successive page **/
stride=elemXline/4;

for(i=0;i<(arrsize-stride);i+=stride)
{
    arr[i]=1.1;
    arr[i+1]=1.1;
    arr[i+2]=1.1;
    arr[i+3]=1.1;
    arr[i+4]=1.1;
    arr[i+5]=1.1;
    arr[i+6]=1.1;
    arr[i+7]=1.1;
    arr[i+8]=1.1;
    arr[i+9]=1.1;
    arr[i+10]=1.1;
    arr[i+11]=1.1;
    arr[i+12]=1.1;
    arr[i+13]=1.1;
}

arr[arrsize-stride]=1.1; //if last then there is no more pointer to store
arr[arrsize-stride+1]=1.1;

```


[illegible]

```
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
asm volatile("nop");
```

```
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
```

```

asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");
asm volatile("fadd 1,1,1");
asm volatile("fadd 2,2,2");
asm volatile("fadd 3,3,3");
asm volatile("fadd 4,4,4");
asm volatile("fadd 5,5,5");
asm volatile("fadd 6,6,6");
asm volatile("fadd 7,7,7");
asm volatile("fadd 8,8,8");
asm volatile("fadd 9,9,9");
asm volatile("fadd 10,10,10");
asm volatile("fadd 11,11,11");
asm volatile("fadd 12,12,12");
asm volatile("fadd 13,13,13");
asm volatile("fadd 14,14,14");
asm volatile("fadd 15,15,15");
asm volatile("fadd 16,16,16");
asm volatile("fadd 17,17,17");
asm volatile("fadd 18,18,18");
asm volatile("fadd 19,19,19");
asm volatile("fadd 20,20,20");

asm volatile("addi 11,11,0");
asm volatile("addi 12,12,0");
asm volatile("addi 13,13,0");
asm volatile("addi 14,14,0");
asm volatile("addi 15,15,0");
asm volatile("addi 16,16,0");
asm volatile("addi 17,17,0");
asm volatile("addi 18,18,0");
asm volatile("addi 19,19,0");
asm volatile("addi 20,20,0");
asm volatile("addi 21,21,0");
asm volatile("addi 22,22,0");
asm volatile("addi 23,23,0");
asm volatile("addi 24,24,0");

elem2=*(elem+1);
elem3=*(elem+2);
elem4=*(elem+3);
elem5=*(elem+4);
elem6=*(elem+5);
elem7=*(elem+6);

```


Appendix C: PETSc KSP Benchmark

The appendix lists the source code of the PETSc KSP benchmark below.

PETSc KSP Benchmark with periodic sampling to capture Signature:

```
/*
***** We have added PAPI code to ex10.c
***** the counters are started at at PreloadStage setupe and stopped right after
***** preload stage solve
***** The assumption is that option num_fac is equal to 1, if thats not the case
***** then stop the counters at the end of the while (num_fac--) loop befor
***** data structures are freed up
***** another assumption is that preload is set to false so use -f0 option, but
***** do not use -f1 or -f, instead warmup by running a code with papi group 5
***** and then rerun the codes with the different groups
*/

static char help[] = "Reads a PETSc matrix and vector from a file and solves a linear system.\n\
This version first preloads and solves a small system, then loads \n\
another (larger) system and solves it as well. This example illustrates\n\
preloading of instructions with the smaller system so that more accurate\n\
performance monitoring can be done with the larger one (that actually\n\
is the system of interest). See the 'Performance Hints' chapter of the\n\
users manual for a discussion of preloading. Input parameters include\n\
-f0 <input_file> : first file to load (small system)\n\
-f1 <input_file> : second file to load (larger system)\n\
-trans : solve transpose system instead\n\
\n";

/*
This code can be used to test PETSc interface to other packages.\n\
Examples of command line options: \n\
ex10 -f0 <datafile> -ksp_type preonly \n\
    -help -ksp_view \n\
    -num_numfac <num_numfac> -num_rhs <num_rhs> \n\
    -ksp_type preonly -pc_type lu -mat_type aijspooles/superlu/superlu_dist/aijmumps \n\
    -ksp_type preonly -pc_type cholesky -mat_type sbaijspooles/dscpack/sbaijmumps \n\
    -f0 <A> -fB <B> -mat_type sbaijmumps -ksp_type preonly -pc_type cholesky -test_inertia -mat_sigma <sigma> \n\
mpiexec -np <np> ex10 -f0 <datafile> -ksp_type cg -pc_type asm -pc_asm_type basic -sub_pc_type icc -mat_type sbaij
\n\n";
*/
/*T
  Concepts: KSP^solving a linear system
  Processors: n
T*/

/*
Include "petscksp.h" so that we can use KSP solvers. Note that this file
automatically includes:
  petsc.h      - base PETSc routines  petscvec.h - vectors
  petscsys.h   - system routines      petscmat.h - matrices
  petscis.h    - index sets           petscksp.h - Krylov subspace methods
  petscviewer.h - viewers              petscpc.h  - preconditioners
*/
#define _GNU_SOURCE

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <sched.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>

#include "papi.h" /* This needs to be included every time you use PAPI */
```

```

#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/
#define CYCLES_PER_SEC 1500000000
#define overflow_flag 0

void handler(int EventSet, void *address, long_long overflow_vector, void *context)
{
    int retval,i;

    if ((retval = PAPI_read(EventSet, handlervalues)) != PAPI_OK)
    {
        printf("PAPI_read failed and returned %d\n", retval);
        return;
    }

    if ((retval = PAPI_reset(EventSet)) != PAPI_OK)
    {
        printf("PAPI_reset failed and returned %d\n", retval);
        return;
    }

    for(i=0;i<NUM_EVENTS;i++)
    {
        printf("%-18lld",handlervalues[i]);
    }
    printf("\n");

    return;
}

static inline int val_to_char(int v)
{
    if (v >= 0 && v < 10)
        return '0' + v;
    else if (v >= 10 && v < 16)
        return ('a' - 10) + v;
    else
        return -1;
}

static char * cpuset_to_str(cpu_set_t *mask, char *str)
{
    int base;
    char *ptr = str;
    char *ret = 0;

    for (base = CPU_SETSIZE - 4; base >= 0; base -= 4) {
        char val = 0;
        if (CPU_ISSET(base, mask))
            val |= 1;
        if (CPU_ISSET(base + 1, mask))
            val |= 2;
        if (CPU_ISSET(base + 2, mask))
            val |= 4;
        if (CPU_ISSET(base + 3, mask))
            val |= 8;
        if (!ret && val)
            ret = ptr;
        *ptr++ = val_to_char(val);
    }
}

```

```

    }
    *ptr = 0;
    return ret ? ret : ptr - 1;
}

int main(int argc, char **args)
{
    KSP      ksp;      /* linear solver context */
    Mat      A,B;      /* matrix */
    Vec      x,b,u;     /* approx solution, RHS, exact solution */
    PetscViewer fd;     /* viewer */
    char      file[3][PETSC_MAX_PATH_LEN]; /* input file name */
    PetscTruth table,flag,flagB=PETSC_FALSE,trans=PETSC_FALSE,partition=PETSC_FALSE;
    PetscErrorCode ierr;
    PetscInt  its,num_numfac,m,n,M;
    PetscReal norm;
    PetscLogDouble tsetup,tsetup1,tsetup2,tsolve,tsolve1,tsolve2;
    PetscTruth preload=PETSC_TRUE,diagonalscale,isSymmetric,cknorm=PETSC_FALSE,Test_MatDuplicate=PETSC_FALSE;
    PetscMPIInt rank;
    PetscScalar sigma;
    int      papi_group=0;
    int      samples_per_sec=1;
    long     THRESHOLD=1;
    /***** PAPI STUFF *****/

    long_long values[NUM_EVENTS];
    /* This is where we store the values we read from the eventset */

    int i,retval,number=NUM_EVENTS,Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    /***** This is the part to bind a process to a cpu *****/
    /***** It prints out current cpu mask and the new mask *****/
    cpu_set_t cur_mask, new_mask;
    pid_t p;
    int cpu_to_bind;
    char mstr[1 + CPU_SETSIZE / 4];

    p=0; /* binds the current process */
    cpu_to_bind=4; /* binds it to processor 7 */

    if (sched_getaffinity(p, sizeof (cur_mask), &cur_mask) < 0) {
        perror("sched_getaffinity");
        return -1;
    }

    /* printf("pid %d's current affinity list: %s\n", p,
        cpuset_to_str(&cur_mask, mstr));
    */
    CPU_ZERO(&new_mask);

    CPU_SET(cpu_to_bind, &new_mask);

    if (sched_setaffinity(p, sizeof (new_mask), &new_mask)) {
        perror("sched_setaffinity");
        return -1;
    }

    if (sched_getaffinity(p, sizeof (cur_mask), &cur_mask) < 0) {
        perror("sched_getaffinity");
        return -1;
    }

    /* printf("pid %d's new affinity mask: %s\n", p,
        cpuset_to_str(&cur_mask, mstr));
    */

    /***** End of bindind task *****/

```



```

PetscInitialize(&argc,&args,(char *)0,help);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);CHKERRQ(ierr);
ierr = PetscOptionsHasName(PETSC_NULL,"-table",&table);CHKERRQ(ierr);
ierr = PetscOptionsHasName(PETSC_NULL,"-trans",&trans);CHKERRQ(ierr);
ierr = PetscOptionsHasName(PETSC_NULL,"-partition",&partition);CHKERRQ(ierr);

/*
   Determine files from which we read the two linear systems
   (matrix and right-hand-side vector).
*/
ierr = PetscOptionsGetString(PETSC_NULL,"-f",file[0],PETSC_MAX_PATH_LEN-1,&flag);CHKERRQ(ierr);
if (flag)
{
    ierr = PetscStrncpy(file[1],file[0]);CHKERRQ(ierr);
    preload = PETSC_FALSE;
}
else
{
    ierr = PetscOptionsGetString(PETSC_NULL,"-f0",file[0],PETSC_MAX_PATH_LEN-1,&flag);CHKERRQ(ierr);
    if (!flag) SETERRQ(1,"Must indicate binary file with the -f0 or -f option");
    ierr = PetscOptionsGetString(PETSC_NULL,"-f1",file[1],PETSC_MAX_PATH_LEN-1,&flag);CHKERRQ(ierr);
    if (!flag) {preload = PETSC_FALSE;} /* don't bother with second system */
}

ierr = PetscOptionsGetInt(PETSC_NULL,"-G",&papi_group,&flag);CHKERRQ(ierr);
if (!flag) SETERRQ(1,"Must indicate pmcount group from one of following 43,44,80,92 with the -G option.");
ierr = PetscOptionsGetInt(PETSC_NULL,"-S",&samples_per_sec,&flag);CHKERRQ(ierr);
if (!flag) SETERRQ(1,"Must indicate desired num of samples per sec with -S option.");

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

switch (papi_group)
{
    case 5:
        /***** pmcount group 5 *****/
        Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
        Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
        Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
        Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 43:
        /***** pmcount group 43 *****/
        Events[0]=0x40000014; //PM_DATA_TABLEWALK_CYC
        Events[1]=0x40000016; //PM_DTLB_MISS
        Events[2]=0x4000015f; //PM_LD_MISS_L1
        Events[3]=0x400001c0; // PM_LD_REF_L1
        Events[4]=0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;
    case 44:
        /***** pmcount group 44 *****/
        Events[0]=0x4000000d; //PM_DATA_FROM_L2
        Events[1]=0x400000ea; //PM_LSU_DERAT_MISS
        Events[2]=0x40000199; //PM_ST_REF_L1
        Events[3]=0x40000198; // PM_ST_MISS_L1
        Events[4]= 0x40000049; // PM_INST_CMPL
        Events[5]=0x400000bd; //PM_RUN_CYC
        break;

    case 80:

```

```

/***** pmcount group 80 *****/
Events[0]=0x40000036; //PM_FPU_DENORM
Events[1]=0x400000de; // PM_FPU_STALL3
Events[2]=0x4000011c; //PM_FPU0_FIN
Events[3]=0x40000121; //PM_FPU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;

case 92:
/***** pmcount group 92 *****/
Events[0]=0x40000004; //PM_3INST_CLB_CYC
Events[1]=0x40000005; //PM_4INST_CLB_CYC
Events[2]=0x40000129; //PM_FXU0_FIN
Events[3]=0x4000012a; //PM_FXU1_FIN
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC
break;
default:
printf("Group # %d is not valid\n",papi_group);
return 0;
}

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

PAPI_add_events(EventSet, Events, NUM_EVENTS);
/***** END OF PAPI INIT *****/

/* -----
Beginning of linear solver loop
----- */
/*
Loop through the linear solve 2 times.
- The intention here is to preload and solve a small system;
then load another (larger) system and solve it as well.
This process preloads the instructions with the smaller
system so that more accurate performance monitoring (via
-log_summary) can be done with the larger one (that actually
is the system of interest).
*/
PreLoadBegin(preload,"Load system");

/* ----- New Stage -----
Load system
----- */

/*
Open binary file. Note that we use FILE_MODE_READ to indicate
reading from this file.
*/
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[PreLoadIt],FILE_MODE_READ,&fd);CHKERRQ(ierr);

/*
Load the matrix and vector; then destroy the viewer.
*/
ierr = MatLoad(fd,MATAIJ,&A);CHKERRQ(ierr);

if (!preload)
{
flg = PETSC_FALSE;
ierr = PetscOptionsGetString(PETSC_NULL,"-rhs",file[2],PETSC_MAX_PATH_LEN-1,&flg);CHKERRQ(ierr);
if (flg)
{ /* rhs is stored in a separate file */
ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[2],FILE_MODE_READ,&fd);CHKERRQ(ierr);
}
}

```

```

}
if (rank)
{
ierr = PetscExceptionTry1(VecLoad(fd,PETSC_NULL,&b),PETSC_ERR_FILE_UNEXPECTED);
}
else
{
ierr = PetscExceptionTry1(VecLoad(fd,PETSC_NULL,&b),PETSC_ERR_FILE_READ);
}
if (PetscExceptionCaught(ierr,PETSC_ERR_FILE_UNEXPECTED) || PetscExceptionCaught(ierr,PETSC_ERR_FILE_READ))
{ /* if file contains no RHS, then use a vector of all ones */
PetscInt m;
PetscScalar one = 1.0;
ierr = PetscInfo(0,"Using vector of ones for RHS\n");CHKERRQ(ierr);
ierr = MatGetLocalSize(A,&m,PETSC_NULL);CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&b);CHKERRQ(ierr);
ierr = VecSetSizes(b,m,PETSC_DECIDE);CHKERRQ(ierr);
ierr = VecSetFromOptions(b);CHKERRQ(ierr);
ierr = VecSet(b,one);CHKERRQ(ierr);
}
else CHKERRQ(ierr);

ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);

/* Test MatDuplicate() */
if (Test_MatDuplicate)
{
ierr = MatDuplicate(A,MAT_COPY_VALUES,&B);CHKERRQ(ierr);
ierr = MatEqual(A,B,&flag);CHKERRQ(ierr);
if (!flag)
{
PetscPrintf(PETSC_COMM_WORLD, " A != B \n");CHKERRQ(ierr);
}
ierr = MatDestroy(B);CHKERRQ(ierr);
}

/* Add a shift to A */
ierr = PetscOptionsGetScalar(PETSC_NULL, "-mat_sigma", &sigma, &flag);CHKERRQ(ierr);
if (flag)
{
ierr = PetscOptionsGetString(PETSC_NULL, "-fB", file[2], PETSC_MAX_PATH_LEN-1, &flagB);CHKERRQ(ierr);
if (flagB)
{
/* load B to get A = A + sigma*B */
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, file[2], FILE_MODE_READ, &fd);CHKERRQ(ierr);
ierr = MatLoad(fd, MAT_AIJ, &B);CHKERRQ(ierr);
ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);
ierr = MatAXPY(A, sigma, B, DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr); /* A <- sigma*B + A */
}
else
{
ierr = MatShift(A, sigma);CHKERRQ(ierr);
}
}

/* Make A singular for testing zero-pivot of ilu factorization */
/* Example: ./ex10 -f0 <datafile> -test_zeropivot -set_row_zero -pc_factor_shift_nonzero */
ierr = PetscOptionsHasName(PETSC_NULL, "-test_zeropivot", &flag);CHKERRQ(ierr);
if (flag)
{
PetscInt row, ncols;
const PetscInt *cols;
const PetscScalar *vals;
PetscTruth flag1 = PETSC_FALSE;
PetscScalar *zeros;
row = 0;
ierr = MatGetRow(A, row, &ncols, &cols, &vals);CHKERRQ(ierr);
ierr = PetscMalloc(sizeof(PetscScalar)*(ncols+1), &zeros);
ierr = PetscMemzero(zeros, (ncols+1)*sizeof(PetscScalar));CHKERRQ(ierr);
ierr = PetscOptionsHasName(PETSC_NULL, "-set_row_zero", &flag1);CHKERRQ(ierr);
if (flag1)
{ /* set entire row as zero */

```

```

        ierr = MatSetValues(A,1,&row,ncols,cols,zeros,INSERT_VALUES);CHKERRQ(ierr);
    }
    else
    { /* only set (row,row) entry as zero */
        ierr = MatSetValues(A,1,&row,1,&row,zeros,INSERT_VALUES);CHKERRQ(ierr);
    }
    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
}

/* Check whether A is symmetric */
ierr = PetscOptionsHasName(PETSC_NULL, "-check_symmetry", &flag);CHKERRQ(ierr);
if (flag)
{
    Mat Atrans;
    ierr = MatTranspose(A, &Atrans);
    ierr = MatEqual(A, Atrans, &isSymmetric);
    if (isSymmetric)
    {
        PetscPrintf(PETSC_COMM_WORLD,"A is symmetric\n");CHKERRQ(ierr);
    }
    else
    {
        PetscPrintf(PETSC_COMM_WORLD,"A is non-symmetric\n");CHKERRQ(ierr);
    }
    ierr = MatDestroy(Atrans);CHKERRQ(ierr);
}

/*
    If the loaded matrix is larger than the vector (due to being padded
    to match the block size of the system), then create a new padded vector.
*/

ierr = MatGetLocalSize(A,&m,&n);CHKERRQ(ierr);
if (m != n)
{
    SETERRQ2(PETSC_ERR_ARG_SIZ, "This example is not intended for rectangular matrices (%d, %d)", m, n);
}
ierr = MatGetSize(A,&M,PETSC_NULL);CHKERRQ(ierr);
ierr = VecGetSize(b,&m);CHKERRQ(ierr);
if (M != m)
{ /* Create a new vector b by padding the old one */
    PetscInt j,mvec,start,end,indx;
    Vec tmp;
    PetscScalar *bold;

    ierr = VecCreate(PETSC_COMM_WORLD,&tmp);CHKERRQ(ierr);
    ierr = VecSetSizes(tmp,n,PETSC_DECIDE);CHKERRQ(ierr);
    ierr = VecSetFromOptions(tmp);CHKERRQ(ierr);
    ierr = VecGetOwnershipRange(b,&start,&end);CHKERRQ(ierr);
    ierr = VecGetLocalSize(b,&mvec);CHKERRQ(ierr);
    ierr = VecGetArray(b,&bold);CHKERRQ(ierr);
    for (j=0; j<mvec; j++)
    {
        indx = start+j;
        ierr = VecSetValues(tmp,1,&indx,bold+j,INSERT_VALUES);CHKERRQ(ierr);
    }
    ierr = VecRestoreArray(b,&bold);CHKERRQ(ierr);
    ierr = VecDestroy(b);CHKERRQ(ierr);
    ierr = VecAssemblyBegin(tmp);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(tmp);CHKERRQ(ierr);
    b = tmp;
}
ierr = VecDuplicate(b,&x);CHKERRQ(ierr);
ierr = VecDuplicate(b,&u);CHKERRQ(ierr);
ierr = VecSet(x,0.0);CHKERRQ(ierr);

/* ----- New Stage -----
    Setup solve for system
----- */

```

```

if (partition)
{
    MatPartitioning mpart;
    IS      mis,nis,isn,is;
    PetscInt  *count;
    PetscMPIInt  size;
    Mat      BB;
    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);CHKERRQ(ierr);
    ierr = PetscMalloc(size*sizeof(PetscInt),&count);CHKERRQ(ierr);
    ierr = MatPartitioningCreate(PETSC_COMM_WORLD, &mpart);CHKERRQ(ierr);
    ierr = MatPartitioningSetAdjacency(mpart, A);CHKERRQ(ierr);
    /* ierr = MatPartitioningSetVertexWeights(mpart, weight);CHKERRQ(ierr); */
    ierr = MatPartitioningSetFromOptions(mpart);CHKERRQ(ierr);
    ierr = MatPartitioningApply(mpart, &mis);CHKERRQ(ierr);
    ierr = MatPartitioningDestroy(mpart);CHKERRQ(ierr);
    ierr = ISPartitioningToNumbering(mis,&nis);CHKERRQ(ierr);
    ierr = ISPartitioningCount(mis,count);CHKERRQ(ierr);
    ierr = ISDestroy(mis);CHKERRQ(ierr);
    ierr = ISInvertPermutation(nis, count[rank], &is);CHKERRQ(ierr);
    ierr = PetscFree(count);CHKERRQ(ierr);
    ierr = ISDestroy(nis);CHKERRQ(ierr);
    ierr = ISort(is);CHKERRQ(ierr);
    ierr = ISAllGather(is,&isn);CHKERRQ(ierr);
    ierr = MatGetSubMatrix(A,is,isn,PETSC_DECIDE,MAT_INITIAL_MATRIX,&BB);CHKERRQ(ierr);

    /* need to move the vector also */
    ierr = ISDestroy(is);CHKERRQ(ierr);
    ierr = ISDestroy(isn);CHKERRQ(ierr);
    ierr = MatDestroy(A);CHKERRQ(ierr);
    A = BB;
}

/*
Conclude profiling last stage; begin profiling next stage.
*/
PreLoadStage("KSPSetUp");

/***** PAPI STUFF *****/
/* ***** Start counters *****/
THRESHOLD=CYCLES_PER_SEC/samples_per_sec;
retval = PAPI_overflow(EventSet,0x400000bd, THRESHOLD, overflow_flag, handler);
if(retval !=PAPI_OK)
{
    printf("overflow call failed with return value %d\n",retval);
    exit(0);
}

printf("\n");    for(i=0;i<NUM_EVENTS;i++)
{
    PAPI_event_code_to_name(Events[i],name);
    printf("%s\t",name);
}
printf("\n");

PAPI_start(EventSet);

/*
We also explicitly time this stage via PetscGetTime()
*/
ierr = PetscGetTime(&tsetup1);CHKERRQ(ierr);

/*
Create linear solver; set operators; set runtime options.
*/
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
num_numfac = 1;
ierr = PetscOptionsGetInt(PETSC_NULL,"-num_numfac",&num_numfac,PETSC_NULL);CHKERRQ(ierr);
while ( num_numfac-- )
{

```

```

ierr = KSPSetOperators(ksp,A,A,SAME_NONZERO_PATTERN);CHKERRQ(ierr);
ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

/*
Here we explicitly call KSPSetUp() and KSPSetUpOnBlocks() to
enable more precise profiling of setting up the preconditioner.
These calls are optional, since both will be called within
KSPSolve() if they haven't been called already.
*/
ierr = KSPSetUp(ksp);CHKERRQ(ierr);
ierr = KSPSetUpOnBlocks(ksp);CHKERRQ(ierr);
ierr = PetscGetTime(&tsetup2);CHKERRQ(ierr);
tsetup = tsetup2 - tsetup1;

/*
Test MatGetInertia()
Usage:
ex10 -f0 <mat_binaryfile> -ksp_type preonly -pc_type cholesky -mat_type seqsbaij -test_inertia -mat_sigma <sigma>
*/
ierr = PetscOptionsHasName(PETSC_NULL,"-test_inertia",&flag);CHKERRQ(ierr);
if (flag)
{
PC      pc;
PetscInt nneg, nzero, npos;
Mat      F;

ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = PCGetFactoredMatrix(pc,&F);CHKERRQ(ierr);
ierr = MatGetInertia(F,&nneg,&nzero,&npos);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_SELF," MatInertia: nneg: %D, nzero: %D, npos: %D\n",nneg,nzero,npos);
}

/*
Tests "diagonal-scaling of preconditioned residual norm" as used
by many ODE integrator codes including SUNDIALS. Note this is different
than diagonally scaling the matrix before computing the preconditioner
*/
ierr = PetscOptionsHasName(PETSC_NULL,"-diagonal_scale",&diagonalscale);CHKERRQ(ierr);
if (diagonalscale)
{
PC      pc;
PetscInt j,start,end,n;
Vec      scale;

ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = VecGetSize(x,&n);CHKERRQ(ierr);
ierr = VecDuplicate(x,&scale);CHKERRQ(ierr);
ierr = VecGetOwnershipRange(scale,&start,&end);CHKERRQ(ierr);
for (j=start; j<end; j++)
{
ierr = VecSetValue(scale,j,((PetscReal)(j+1))/((PetscReal)n),INSERT_VALUES);CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(scale);CHKERRQ(ierr);
ierr = VecAssemblyEnd(scale);CHKERRQ(ierr);
ierr = PCDiagonalScaleSet(pc,scale);CHKERRQ(ierr);
ierr = VecDestroy(scale);CHKERRQ(ierr);
}

/* ----- New Stage -----
Solve system
----- */

/*
Begin profiling next stage
*/
PreLoadStage("KSPSolve");

/*
Solve linear system; we also explicitly time this stage.
*/
ierr = PetscGetTime(&tsolve1);CHKERRQ(ierr);

```

```

if (trans)
{
ierr = KSPSolveTranspose(ksp,b,x);CHKERRQ(ierr);
ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
}
else
{
PetscInt num_rhs=1;
ierr = PetscOptionsGetInt(PETSC_NULL,"-num_rhs",&num_rhs,PETSC_NULL);CHKERRQ(ierr);
ierr = PetscOptionsHasName(PETSC_NULL,"-cknorm",&cknorm);CHKERRQ(ierr);
while ( num_rhs-- )
{
ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
}
ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
if (cknorm)
{ /* Check error for each rhs */
if (trans)
{
ierr = MatMultTranspose(A,x,u);CHKERRQ(ierr);
}
else
{
ierr = MatMult(A,x,u);CHKERRQ(ierr);
}
ierr = VecAXPY(u,-1.0,b);CHKERRQ(ierr);
ierr = VecNorm(u,NORM_2,&norm);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD," Number of iterations = %3D\n",its);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD," Residual norm %A\n",norm);CHKERRQ(ierr);
}
} /* while ( num_rhs-- ) */
ierr = PetscGetTime(&tsolve2);CHKERRQ(ierr);
tsolve = tsolve2 - tsolve1;

/***** PAPI STUFF *****/
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);
for(i=0;i<NUM_EVENTS;i++)
{
printf("%-18ld",values[i]);
}
printf("\n");
/* free the resources used by PAPI */
PAPI_shutdown();

/*
Conclude profiling this stage
*/
PreLoadStage("Cleanup");

/* ----- New Stage -----
Check error, print output, free data structures.
----- */

/*
Check error
*/
if (trans)
{
ierr = MatMultTranspose(A,x,u);CHKERRQ(ierr);
}
else
{
ierr = MatMult(A,x,u);CHKERRQ(ierr);
}
ierr = VecAXPY(u,-1.0,b);CHKERRQ(ierr);
ierr = VecNorm(u,NORM_2,&norm);CHKERRQ(ierr);

/*
Write output (optionally using table for solver details).
- PetscPrintf() handles output for multiprocessor jobs

```

by printing from only one processor in the communicator.

- KSPView() prints information about the linear solver.

```

*/
if (table)
{
    char    *matrixname,kspinfo[120];
    PetscViewer viewer;

/*
Open a string viewer; then write info to it.
*/
    ierr = PetscViewerStringOpen(PETSC_COMM_WORLD,kspinfo,120,&viewer);CHKERRQ(ierr);
    ierr = KSPView(ksp,viewer);CHKERRQ(ierr);
    ierr = PetscStrchr(file[PreLoadIt],',",&matrixname);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"%-8.8s %3D %2.0e %2.1e %2.1e %s\n",
        matrixname,its,norm,tsetup+tsolve,tsetup,tsolve,kspinfo);CHKERRQ(ierr);

/*
Destroy the viewer
*/
    ierr = PetscViewerDestroy(viewer);CHKERRQ(ierr);
}
else
{
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Number of iterations = %3D\n",its);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Residual norm %A\n",norm);CHKERRQ(ierr);
}

ierr = PetscOptionsHasName(PETSC_NULL, "-ksp_reason", &flag);CHKERRQ(ierr);
if (flag)
{
    KSPConvergedReason reason;
    ierr = KSPGetConvergedReason(ksp,&reason);CHKERRQ(ierr);
    PetscPrintf(PETSC_COMM_WORLD,"KSPConvergedReason: %D\n", reason);
}

} /* while ( num_numfac-- ) */

/*
Free work space. All PETSc objects should be destroyed when they
are no longer needed.
*/
ierr = MatDestroy(A);CHKERRQ(ierr); ierr = VecDestroy(b);CHKERRQ(ierr);
ierr = VecDestroy(u);CHKERRQ(ierr); ierr = VecDestroy(x);CHKERRQ(ierr);
ierr = KSPDestroy(ksp);CHKERRQ(ierr);
if (flagB) { ierr = MatDestroy(B);CHKERRQ(ierr); }
PreLoadEnd();
/* -----
End of linear solver loop
----- */

ierr = PetscFinalize();CHKERRQ(ierr);
return 0;
}

```

PETSc KSP Benchmark with cumulative performance counters used for SMT experiments:

```

/*
***** We have added PAPI code to ex10.c
***** the counters are started at at PreloadStage setupe and stopped right after
***** prelaod stage solve
***** The assumption is that option num_fac is equal to 1, if thats not the case
***** then stop the counters at the end of the while (num_fac--) loop befor
***** data structures are freed up
***** another assumption is that preload is set to false so use -f0 option, but
***** do not use -f1 or -f, instead warmup by running a code with papi group 5
***** and then rerun the codes with the different groups
*/

```

static char help[] = "Reads a PETSc matrix and vector from a file and solves a linear system.\n"


```

This version first preloads and solves a small system, then loads \n\
another (larger) system and solves it as well. This example illustrates\n\
preloading of instructions with the smaller system so that more accurate\n\
performance monitoring can be done with the larger one (that actually\n\
is the system of interest). See the 'Performance Hints' chapter of the\n\
users manual for a discussion of preloading. Input parameters include\n\
-f0 <input_file> : first file to load (small system)\n\
-f1 <input_file> : second file to load (larger system)\n\n\
-trans : solve transpose system instead\n\n";
/*
This code can be used to test PETSc interface to other packages.\n\
Examples of command line options: \n\
ex10 -f0 <datafile> -ksp_type preonly \n\
-help -ksp_view \n\
-num_numfac <num_numfac> -num_rhs <num_rhs> \n\
-ksp_type preonly -pc_type lu -mat_type aijspooles/superlu/superlu_dist/aijmumps \n\
-ksp_type preonly -pc_type cholesky -mat_type sbaijspooles/dscpack/sbaijmumps \n\
-f0 <A> -fB <B> -mat_type sbaijmumps -ksp_type preonly -pc_type cholesky -test_inertia -mat_sigma <sigma> \n\
mpiexec -np <np> ex10 -f0 <datafile> -ksp_type cg -pc_type asm -pc_asm_type basic -sub_pc_type icc -mat_type sbaij
\n\n";
*/
/*T
Concepts: KSP^solving a linear system
Processors: n
T*/

/*
Include "petscksp.h" so that we can use KSP solvers. Note that this file
automatically includes:
petsc.h - base PETSc routines petscvec.h - vectors
petscsys.h - system routines petscmat.h - matrices
petscis.h - index sets petscksp.h - Krylov subspace methods
petscviewer.h - viewers petscpc.h - preconditioners
*/
#define _GNU_SOURCE

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <sched.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>

#include "papi.h" /* This needs to be included every time you use PAPI */

#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"

#define NUM_EVENTS 6

static long long handlervalues[NUM_EVENTS];
static int EventSet=PAPI_NULL;
/*must be initialized to PAPI_NULL before calling PAPI_create_event*/

static inline int val_to_char(int v)
{
    if (v >= 0 && v < 10)
        return '0' + v;
    else if (v >= 10 && v < 16)
        return ('a' - 10) + v;
    else

```

```

        return -1;
    }

static char *cpuset_to_str(cpu_set_t *mask, char *str)
{
    int base;
    char *ptr = str;
    char *ret = 0;

    for (base = CPU_SETSIZE - 4; base >= 0; base -= 4) {
        char val = 0;
        if (CPU_ISSET(base, mask))
            val |= 1;
        if (CPU_ISSET(base + 1, mask))
            val |= 2;
        if (CPU_ISSET(base + 2, mask))
            val |= 4;
        if (CPU_ISSET(base + 3, mask))
            val |= 8;
        if (!ret && val)
            ret = ptr;
        *ptr++ = val_to_char(val);
    }
    *ptr = 0;
    return ret ? ret : ptr - 1;
}

int main(int argc, char **args)
{
    KSP      ksp;      /* linear solver context */
    Mat      A,B;      /* matrix */
    Vec      x,b,u;     /* approx solution, RHS, exact solution */
    PetscViewer fd;     /* viewer */
    char      file[3][PETSC_MAX_PATH_LEN]; /* input file name */
    PetscTruth table,flag,flagB=PETSC_FALSE,trans=PETSC_FALSE,partition=PETSC_FALSE;
    PetscErrorCode ierr;
    PetscInt  its,num_numfac,m,n,M;
    PetscReal norm;
    PetscLogDouble tsetup,tsetup1,tsetup2,tsolve,tsolve1,tsolve2;
    PetscTruth preload=PETSC_TRUE,diagonalscale,isSymmetric,cknorm=PETSC_FALSE,Test_MatDuplicate=PETSC_FALSE;
    PetscMPIInt rank;
    PetscScalar sigma;
    /***** PAPI STUFF *****/

    long_long values[NUM_EVENTS];
    /* This is where we store the values we read from the eventset */

    int i, retval, number=NUM_EVENTS, Events[NUM_EVENTS];
    /* We use number to keep track of the number of events in the EventSet */
    char errstring[PAPI_MAX_STR_LEN];
    char name[PAPI_MAX_STR_LEN];
    /*****

    /***** This are the variables to bind a process to a cpu *****/
    cpu_set_t cur_mask, new_mask;
    pid_t p;
    int cpu_to_bind;
    char mstr[1 + CPU_SETSIZE / 4];
    /*****

    PetscInitialize(&argc,&args,(char *)0,help);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-table",&table);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-trans",&trans);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-partition",&partition);CHKERRQ(ierr);

    /*
        Determine files from which we read the two linear systems
        (matrix and right-hand-side vector).

```

```

*/
ierr = PetscOptionsGetString(PETSC_NULL, "-f", file[0], PETSC_MAX_PATH_LEN-1, &flag); CHKERRQ(ierr);
if (flag)
{
    ierr = PetscStrncpy(file[1], file[0]); CHKERRQ(ierr);
    preload = PETSC_FALSE;
}
else
{
    ierr = PetscOptionsGetString(PETSC_NULL, "-f0", file[0], PETSC_MAX_PATH_LEN-1, &flag); CHKERRQ(ierr);
    if (!flag) SETERRQ(1, "Must indicate binary file with the -f0 or -f option");
    ierr = PetscOptionsGetString(PETSC_NULL, "-f1", file[1], PETSC_MAX_PATH_LEN-1, &flag); CHKERRQ(ierr);
    if (!flag) {preload = PETSC_FALSE;} /* don't bother with second system */
}

ierr = PetscOptionsGetInt(PETSC_NULL, "-C", &cpu_to_bind, &flag); CHKERRQ(ierr);
if (!flag) SETERRQ(1, "Must indicate desired cpu to bind with -C option.");

/***** This is the part to bind a process to a cpu *****/
/***** It prints out current cpu mask and the new mask *****/

p=0; /* binds the current process */

if (sched_getaffinity(p, sizeof (cur_mask), &cur_mask) < 0) {
    perror("sched_getaffinity");
    return -1;
}

/* printf("pid %d's current affinity list: %s\n", p,
    cpuset_to_str(&cur_mask, mstr));
*/
CPU_ZERO(&new_mask);

CPU_SET(cpu_to_bind, &new_mask);

if (sched_setaffinity(p, sizeof (new_mask), &new_mask)) {
    perror("sched_setaffinity");
    return -1;
}

if (sched_getaffinity(p, sizeof (cur_mask), &cur_mask) < 0) {
    perror("sched_getaffinity");
    return -1;
}

/* printf("pid %d's new affinity mask: %s\n", p,
    cpuset_to_str(&cur_mask, mstr));
*/

/***** End of bindind task *****/

/*****
PAPI INITIALIZATION AND EVENT SET ADDITIONS
*****/

PAPI_library_init(PAPI_VER_CURRENT);

/***** pmcount group 5 *****/
Events[0]=0x4000003c; //PM_GCT_NOSLOT_CYC
Events[1]=0x400000e2; //PM_GCT_NOSLOT_IC_MISS
Events[2]=0x4000012b; //PM_GCT_NOSLOT_SRQ_FULL
Events[3]=0x400001bc; // PM_GCT_NOSLOT_BR_MPREd
Events[4]= 0x40000049; // PM_INST_CMPL
Events[5]=0x400000bd; //PM_RUN_CYC

/* Creating the eventset */
PAPI_create_eventset(&EventSet);

```

```
PAPI_add_events(EventSet, Events, NUM_EVENTS);
/***** END OF PAPI INIT *****/
```

```
/* -----
   Beginning of linear solver loop
   ----- */
/*
   Loop through the linear solve 2 times.
   - The intention here is to preload and solve a small system;
     then load another (larger) system and solve it as well.
     This process preloads the instructions with the smaller
     system so that more accurate performance monitoring (via
     log_summary) can be done with the larger one (that actually
     is the system of interest).
   */
   PreLoadBegin(preload,"Load system");

/* ----- New Stage -----
   Load system
   ----- */

/*
   Open binary file. Note that we use FILE_MODE_READ to indicate
   reading from this file.
   */
   ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[PreLoadIt],FILE_MODE_READ,&fd);CHKERRQ(ierr);

/*
   Load the matrix and vector; then destroy the viewer.
   */
   ierr = MatLoad(fd,MATAIJ,&A);CHKERRQ(ierr);

   if (!preload)
   {
       flg = PETSC_FALSE;
       ierr = PetscOptionsGetString(PETSC_NULL,"-rhs",file[2],PETSC_MAX_PATH_LEN-1,&flg);CHKERRQ(ierr);
       if (flg)
       { /* rhs is stored in a separate file */
           ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);
           ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,file[2],FILE_MODE_READ,&fd);CHKERRQ(ierr);
       }
   }

   if (rank)
   {
       ierr = PetscExceptionTry1(VecLoad(fd,PETSC_NULL,&b),PETSC_ERR_FILE_UNEXPECTED);
   }
   else
   {
       ierr = PetscExceptionTry1(VecLoad(fd,PETSC_NULL,&b),PETSC_ERR_FILE_READ);
   }

   if (PetscExceptionCaught(ierr,PETSC_ERR_FILE_UNEXPECTED) || PetscExceptionCaught(ierr,PETSC_ERR_FILE_READ))
   { /* if file contains no RHS, then use a vector of all ones */
       PetscInt m;
       PetscScalar one = 1.0;
       ierr = PetscInfo(0,"Using vector of ones for RHS\n");CHKERRQ(ierr);
       ierr = MatGetLocalSize(A,&m,PETSC_NULL);CHKERRQ(ierr);
       ierr = VecCreate(PETSC_COMM_WORLD,&b);CHKERRQ(ierr);
       ierr = VecSetSizes(b,m,PETSC_DECIDE);CHKERRQ(ierr);
       ierr = VecSetFromOptions(b);CHKERRQ(ierr);
       ierr = VecSet(b,one);CHKERRQ(ierr);
   }
   else CHKERRQ(ierr);

   ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);

/* Test MatDuplicate() */
if (Test_MatDuplicate)
{
    ierr = MatDuplicate(A,MAT_COPY_VALUES,&B);CHKERRQ(ierr);
    ierr = MatEqual(A,B,&flg);CHKERRQ(ierr);
}
```

```

    if (!flg)
    {
        PetscPrintf(PETSC_COMM_WORLD, " A != B \n");CHKERRQ(ierr);
    }
    ierr = MatDestroy(B);CHKERRQ(ierr);
}

/* Add a shift to A */
ierr = PetscOptionsGetScalar(PETSC_NULL, "-mat_sigma", &sigma, &flg);CHKERRQ(ierr);
if (flg)
{
    ierr = PetscOptionsGetString(PETSC_NULL, "-fb", file[2], PETSC_MAX_PATH_LEN-1, &flgB);CHKERRQ(ierr);
    if (flgB)
    {
        /* load B to get A = A + sigma*B */
        ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, file[2], FILE_MODE_READ, &fd);CHKERRQ(ierr);
        ierr = MatLoad(fd, MAT_AIJ, &B);CHKERRQ(ierr);
        ierr = PetscViewerDestroy(fd);CHKERRQ(ierr);
        ierr = MatAXPY(A, sigma, B, DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr); /* A <- sigma*B + A */
    }
    else
    {
        ierr = MatShift(A, sigma);CHKERRQ(ierr);
    }
}

/* Make A singular for testing zero-pivot of ilu factorization */
/* Example: ./ex10 -f0 <datafile> -test_zeropivot -set_row_zero -pc_factor_shift_nonzero */
ierr = PetscOptionsHasName(PETSC_NULL, "-test_zeropivot", &flg);CHKERRQ(ierr);
if (flg)
{
    PetscInt    row, ncols;
    const PetscInt    *cols;
    const PetscScalar *vals;
    PetscTruth    flg1 = PETSC_FALSE;
    PetscScalar    *zeros;
    row = 0;
    ierr = MatGetRow(A, row, &ncols, &cols, &vals);CHKERRQ(ierr);
    ierr = PetscMalloc(sizeof(PetscScalar)*(ncols+1), &zeros);
    ierr = PetscMemzero(zeros, (ncols+1)*sizeof(PetscScalar));CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL, "-set_row_zero", &flg1);CHKERRQ(ierr);
    if (flg1)
    { /* set entire row as zero */
        ierr = MatSetValues(A, 1, &row, ncols, cols, zeros, INSERT_VALUES);CHKERRQ(ierr);
    }
    else
    { /* only set (row,row) entry as zero */
        ierr = MatSetValues(A, 1, &row, 1, &row, zeros, INSERT_VALUES);CHKERRQ(ierr);
    }
    ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
}

/* Check whether A is symmetric */
ierr = PetscOptionsHasName(PETSC_NULL, "-check_symmetry", &flg);CHKERRQ(ierr);
if (flg)
{
    Mat Atrans;
    ierr = MatTranspose(A, &Atrans);
    ierr = MatEqual(A, Atrans, &isSymmetric);
    if (isSymmetric)
    {
        PetscPrintf(PETSC_COMM_WORLD, "A is symmetric \n");CHKERRQ(ierr);
    }
    else
    {
        PetscPrintf(PETSC_COMM_WORLD, "A is non-symmetric \n");CHKERRQ(ierr);
    }
    ierr = MatDestroy(Atrans);CHKERRQ(ierr);
}

/*

```

```

        If the loaded matrix is larger than the vector (due to being padded
        to match the block size of the system), then create a new padded vector.
*/

ierr = MatGetLocalSize(A,&m,&n);CHKERRQ(ierr);
if (m != n)
{
    SETERRQ2(PETSC_ERR_ARG_SIZ, "This example is not intended for rectangular matrices (%d, %d)", m, n);
}
ierr = MatGetSize(A,&M,&N,PETSC_NULL);CHKERRQ(ierr);
ierr = VecGetSize(b,&m);CHKERRQ(ierr);
if (M != m)
{ /* Create a new vector b by padding the old one */
    PetscInt j,mvec,start,end,idx;
    Vec      tmp;
    PetscScalar *bold;

    ierr = VecCreate(PETSC_COMM_WORLD,&tmp);CHKERRQ(ierr);
    ierr = VecSetSizes(tmp,n,PETSC_DECIDE);CHKERRQ(ierr);
    ierr = VecSetFromOptions(tmp);CHKERRQ(ierr);
    ierr = VecGetOwnershipRange(b,&start,&end);CHKERRQ(ierr);
    ierr = VecGetLocalSize(b,&mvec);CHKERRQ(ierr);
    ierr = VecGetArray(b,&bold);CHKERRQ(ierr);
    for (j=0; j<mvec; j++)
    {
        idx = start+j;
        ierr = VecSetValues(tmp,1,&idx,bold+j,INSERT_VALUES);CHKERRQ(ierr);
    }
    ierr = VecRestoreArray(b,&bold);CHKERRQ(ierr);
    ierr = VecDestroy(b);CHKERRQ(ierr);
    ierr = VecAssemblyBegin(tmp);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(tmp);CHKERRQ(ierr);
    b = tmp;
}
ierr = VecDuplicate(b,&x);CHKERRQ(ierr);
ierr = VecDuplicate(b,&u);CHKERRQ(ierr);
ierr = VecSet(x,0.0);CHKERRQ(ierr);

/* ----- New Stage -----
    Setup solve for system
----- */

if (partition)
{
    MatPartitioning mpart;
    IS      mis,nis,isn,is;
    PetscInt *count;
    PetscMPIInt size;
    Mat      BB;
    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);CHKERRQ(ierr);
    ierr = PetscMalloc(size*sizeof(PetscInt),&count);CHKERRQ(ierr);
    ierr = MatPartitioningCreate(PETSC_COMM_WORLD, &mpart);CHKERRQ(ierr);
    ierr = MatPartitioningSetAdjacency(mpart, A);CHKERRQ(ierr);
    /* ierr = MatPartitioningSetVertexWeights(mpart, weight);CHKERRQ(ierr); */
    ierr = MatPartitioningSetFromOptions(mpart);CHKERRQ(ierr);
    ierr = MatPartitioningApply(mpart, &mis);CHKERRQ(ierr);
    ierr = MatPartitioningDestroy(mpart);CHKERRQ(ierr);
    ierr = ISPartitioningToNumbering(mis,&nis);CHKERRQ(ierr);
    ierr = ISPartitioningCount(mis,count);CHKERRQ(ierr);
    ierr = ISDestroy(mis);CHKERRQ(ierr);
    ierr = ISInvertPermutation(nis, count[rank], &is);CHKERRQ(ierr);
    ierr = PetscFree(count);CHKERRQ(ierr);
    ierr = ISDestroy(nis);CHKERRQ(ierr);
    ierr = ISSort(is);CHKERRQ(ierr);
    ierr = ISAllGather(is,&isn);CHKERRQ(ierr);
    ierr = MatGetSubMatrix(A,is,isn,PETSC_DECIDE,MAT_INITIAL_MATRIX,&BB);CHKERRQ(ierr);

    /* need to move the vector also */
    ierr = ISDestroy(is);CHKERRQ(ierr);
    ierr = ISDestroy(isn);CHKERRQ(ierr);
}

```

```

        ierr = MatDestroy(A);CHKERRQ(ierr);
        A = BB;
    }

    /*
    Conclude profiling last stage; begin profiling next stage.
    */
    PreLoadStage("KSPSetUp");

    /***** PAPI STUFF *****/
    /* ***** Start counters *****/

    PAPI_start(EventSet);

    /*
    We also explicitly time this stage via PetscGetTime()
    */
    ierr = PetscGetTime(&tsetup1);CHKERRQ(ierr);

    /*
    Create linear solver; set operators; set runtime options.
    */
    ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
    num_numfac = 1;
    ierr = PetscOptionsGetInt(PETSC_NULL,"-num_numfac",&num_numfac,PETSC_NULL);CHKERRQ(ierr);
    while ( num_numfac-- )
    {

        ierr = KSPSetOperators(ksp,A,A,SAME_NONZERO_PATTERN);CHKERRQ(ierr);
        ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);

        /*
        Here we explicitly call KSPSetUp() and KSPSetUpOnBlocks() to
        enable more precise profiling of setting up the preconditioner.
        These calls are optional, since both will be called within
        KSPSolve() if they haven't been called already.
        */
        ierr = KSPSetUp(ksp);CHKERRQ(ierr);
        ierr = KSPSetUpOnBlocks(ksp);CHKERRQ(ierr);
        ierr = PetscGetTime(&tsetup2);CHKERRQ(ierr);
        tsetup = tsetup2 - tsetup1;

        /*
        Test MatGetInertia()
        Usage:
        ex10 -f0 <mat_binaryfile> -ksp_type preonly -pc_type cholesky -mat_type seqsbaij -test_inertia -mat_sigma <sigma>
        */
        ierr = PetscOptionsHasName(PETSC_NULL,"-test_inertia",&flag);CHKERRQ(ierr);
        if (flag)
        {
            PC      pc;
            PetscInt nneg, nzero, npos;
            Mat      F;

            ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
            ierr = PCGetFactoredMatrix(pc,&F);CHKERRQ(ierr);
            ierr = MatGetInertia(F,&nneg,&nzero,&npos);CHKERRQ(ierr);
            ierr = PetscPrintf(PETSC_COMM_SELF," MatInertia: nneg: %D, nzero: %D, npos: %D\n",nneg,nzero,npos);
        }

        /*
        Tests "diagonal-scaling of preconditioned residual norm" as used
        by many ODE integrator codes including SUNDIALS. Note this is different
        than diagonally scaling the matrix before computing the preconditioner
        */
        ierr = PetscOptionsHasName(PETSC_NULL,"-diagonal_scale",&diagonalscale);CHKERRQ(ierr);
        if (diagonalscale)
        {
            PC      pc;
            PetscInt j,start,end,n;

```

```

Vec    scale;

ierr = KSPGetPC(ksp,&pc);CHKERRQ(ierr);
ierr = VecGetSize(x,&n);CHKERRQ(ierr);
ierr = VecDuplicate(x,&scale);CHKERRQ(ierr);
ierr = VecGetOwnershipRange(scale,&start,&end);CHKERRQ(ierr);
for (j=start; j<end; j++)
{
    ierr = VecSetValue(scale,j,((PetscReal)(j+1))/((PetscReal)n),INSERT_VALUES);CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(scale);CHKERRQ(ierr);
ierr = VecAssemblyEnd(scale);CHKERRQ(ierr);
ierr = PCDiagonalScaleSet(pc,scale);CHKERRQ(ierr);
ierr = VecDestroy(scale);CHKERRQ(ierr);
}

/* ----- New Stage -----
      Solve system
      ----- */

/*
Begin profiling next stage
*/
PreLoadStage("KSPSolve");

/*
Solve linear system; we also explicitly time this stage.
*/
ierr = PetscGetTime(&tsolve1);CHKERRQ(ierr);
if (trans)
{
    ierr = KSPSolveTranspose(ksp,b,x);CHKERRQ(ierr);
    ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
}
else
{
    PetscInt num_rhs=1;
    ierr = PetscOptionsGetInt(PETSC_NULL,"-num_rhs",&num_rhs,PETSC_NULL);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-cknorm",&cknorm);CHKERRQ(ierr);
    while ( num_rhs-- )
    {
        ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
    }
    ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
    if (cknorm)
    { /* Check error for each rhs */
        if (trans)
        {
            ierr = MatMultTranspose(A,x,u);CHKERRQ(ierr);
        }
        else
        {
            ierr = MatMult(A,x,u);CHKERRQ(ierr);
        }
        ierr = VecAXPY(u,-1.0,b);CHKERRQ(ierr);
        ierr = VecNorm(u,NORM_2,&norm);CHKERRQ(ierr);
        ierr = PetscPrintf(PETSC_COMM_WORLD," Number of iterations = %3D\n",its);CHKERRQ(ierr);
        ierr = PetscPrintf(PETSC_COMM_WORLD," Residual norm %A\n",norm);CHKERRQ(ierr);
    }
} /* while ( num_rhs-- ) */
ierr = PetscGetTime(&tsolve2);CHKERRQ(ierr);
tsolve = tsolve2 - tsolve1;

/***** PAPI STUFF *****/
/* Stop counting and store the values into the array */
PAPI_stop(EventSet, values);

printf("\n\n***** Perf counter counts *****\n");

for(i=0;i<NUM_EVENTS;i++)
{

```



```

        PAPI_event_code_to_name(Events[i],name);
        printf("%-20s = %20ld\n",name,values[i]);
    }
    printf("\n\n***** End of Perf counter counts *****\n");

/* free the resources used by PAPI */
PAPI_shutdown();

/*
Conclude profiling this stage
*/
PreLoadStage("Cleanup");

/* ----- New Stage -----
Check error, print output, free data structures.
----- */

/*
Check error
*/
if (trans)
{
    ierr = MatMultTranspose(A,x,u);CHKERRQ(ierr);
}
else
{
    ierr = MatMult(A,x,u);CHKERRQ(ierr);
}
ierr = VecAXPY(u,-1.0,b);CHKERRQ(ierr);
ierr = VecNorm(u,NORM_2,&norm);CHKERRQ(ierr);

/*
Write output (optionally using table for solver details).
- PetscPrintf() handles output for multiprocessor jobs
  by printing from only one processor in the communicator.
- KSPView() prints information about the linear solver.
*/
if (table)
{
    char    *matrixname,kspinfo[120];
    PetscViewer viewer;

/*
Open a string viewer; then write info to it.
*/
    ierr = PetscViewerStringOpen(PETSC_COMM_WORLD,kspinfo,120,&viewer);CHKERRQ(ierr);
    ierr = KSPView(ksp,viewer);CHKERRQ(ierr);
    ierr = PetscStrchr(file[PreLoadIt],',',&matrixname);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"%-8.8s %3D %2.0e %2.1e %2.1e %s\n",
        matrixname,its,norm,tsetup+tsolve,tsetup,tsolve,kspinfo);CHKERRQ(ierr);

/*
Destroy the viewer
*/
    ierr = PetscViewerDestroy(viewer);CHKERRQ(ierr);
}
else
{
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Number of iterations = %3D\n",its);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Residual norm %A\n",norm);CHKERRQ(ierr);
}

ierr = PetscOptionsHasName(PETSC_NULL, "-ksp_reason", &flag);CHKERRQ(ierr);
if (flag)
{
    KSPConvergedReason reason;
    ierr = KSPGetConvergedReason(ksp,&reason);CHKERRQ(ierr);
    PetscPrintf(PETSC_COMM_WORLD,"KSPConvergedReason: %D\n", reason);
}

```

```

} /* while ( num_numfac-- ) */

/*
Free work space. All PETSc objects should be destroyed when they
are no longer needed.
*/
ierr = MatDestroy(A);CHKERRQ(ierr); ierr = VecDestroy(b);CHKERRQ(ierr);
ierr = VecDestroy(u);CHKERRQ(ierr); ierr = VecDestroy(x);CHKERRQ(ierr);
ierr = KSPDestroy(ksp);CHKERRQ(ierr);
if (flgB) { ierr = MatDestroy(B);CHKERRQ(ierr); }
PreLoadEnd();
/* -----
      End of linear solver loop
----- */

ierr = PetscFinalize();CHKERRQ(ierr);
return 0;
}

```

Appendix D: Linux Kernel Modifications and Modules

This appendix describes the modifications to the Linux kernel to disable the use of hardware thread priorities. Additionally, the Linux kernel module used for setting hardware thread priorities is also listed.

Kernel Modifications:

File path: include/asm-powerpc/

File 1: ppc_asm.h lines 126 to 132

The original content was:

```
#define HMT_VERY_LOW or 31,31,31 # very low priority
#define HMT_LOW or 1,1,1
#define HMT_MEDIUM_LOW or 6,6,6 # medium low priority
#define HMT_MEDIUM or 2,2,2
#define HMT_MEDIUM_HIGH or 5,5,5 # medium high priority
#define HMT_HIGH or 3,3,3
```

The above was changed to:

```
#define HMT_VERY_LOW # very low priority
#define HMT_LOW
#define HMT_MEDIUM_LOW # medium low priority
#define HMT_MEDIUM
#define HMT_MEDIUM_HIGH # medium high priority
#define HMT_HIGH
```

File 2: processor.h lines 92 to 97

The original content was:

```
#define HMT_very_low() asm volatile("or 31,31,31 # very low priority")
#define HMT_low() asm volatile("or 1,1,1 # low priority")
#define HMT_medium_low() asm volatile("or 6,6,6 # medium low priority")
#define HMT_medium() asm volatile("or 2,2,2 # medium priority")
#define HMT_medium_high() asm volatile("or 5,5,5 # medium high priority")
#define HMT_high() asm volatile("or 3,3,3 # high priority")
```

The above was changed to:

```
#define HMT_very_low()
#define HMT_low()
#define HMT_medium_low()
#define HMT_medium()
#define HMT_medium_high()
#define HMT_high()
```

Linux Kernel Module to set hardware thread priorities at the operating system privilege level:

The module should be compiled using the *makefile* defined below. Insert the module using the insmod command with root privileges. To set hardware thread priorities on a given CPU, bind the shell to the given cpu using taskset command-line tool. Note that the module sets the priority of the hardware thread

that the calling shell is executing on. Hence, to set priority of a particular hardware thread, the user must first bind the shell to the desired hardware thread and then set the priority as explained below.

Makefile:

```
obj-m += SMTprio.o
```

```
all :
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
clean :
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Insert Module: insmod SMTprio.ko

Using the Module:

General Steps

- 1) Bind the shell to the hardware thread whose priority needs to be set
- 2) Set the hardware thread priority by writing the desired level (1-6) to the file /proc/SMTpriority
- 3) Repeat steps 1 and 2 for each hardware thread

Example: set priority of hardware thread 7 to 2

In the shell prompt we use the command ***taskset*** to first bind the shell to hardware thread 7. Note that taskset requires the PID of the shell.

```
>> /usr/bin/taskset -pc 7 <PID_SHELL>
```

Next from this shell we write the desired priority to the file /proc/SMTpriority

```
>> echo 2 > /proc/SMTpriority
```

Kernel Module Source Code:

Filename: SMTprio.c

```
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */
#include <linux/proc_fs.h> /* Necessary because we use proc fs */
#include <linux/string.h>
#include <linux/smp.h>
#include <asm/uaccess.h> /* for copy_to_user */
#include <asm/processor.h>
#define PROC_ENTRY_FILENAME "SMTpriority"
#define PROCFS_MAX_SIZE 2048

static char procfs_buffer[PROCFS_MAX_SIZE];
static unsigned long procfs_buffer_size = 0;
static struct proc_dir_entry *Our_Proc_File;
static ssize_t procfs_read(struct file *filp, /* see include/linux/fs.h */
char *buffer, /* buffer to fill with data */
size_t length, /* length of the buffer */
loff_t * offset)
```

```

{
    static int finished = 0;
    /*
     * We return 0 to indicate end of file, that we have
     * no more information. Otherwise, processes will
     * continue to read from us in an endless loop.
     */

    if ( finished ) {
        printk(KERN_INFO "procfs_read: END\n");
        finished = 0;
        return 0;
    }
    finished = 1;

    /*
     * We use put_to_user to copy the string from the kernel's
     * memory segment to the memory segment of the process
     * that called us. get_from_user, BTW, is
     * used for the reverse.
     */

    if ( copy_to_user(buffer, procfs_buffer, procfs_buffer_size) ) {
        return -EFAULT;
    }
    printk(KERN_INFO "procfs_read: read %lu bytes\n", procfs_buffer_size);
    return procfs_buffer_size; /* Return the number of bytes "read" */
}
/*
 * This function is called when /proc is written
 */
static ssize_t
procfs_write(struct file *file, const char *buffer, size_t len, loff_t * off)
{
    if ( len > PROCFS_MAX_SIZE ) {
        procfs_buffer_size = PROCFS_MAX_SIZE;
    }

    else {
        procfs_buffer_size = len;
    }
    if ( copy_from_user(procfs_buffer, buffer, procfs_buffer_size) ) {
        return -EFAULT;
    }
    printk(KERN_INFO "procfs_write: write %lu bytes\n", procfs_buffer_size);
    if(strncmp(procfs_buffer, "1", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0], smp_processor_id());
        asm volatile("or 31,31,31");
    }
    else if(strncmp(procfs_buffer, "2", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0], smp_processor_id());
        asm volatile("or 1,1,1");
    }
    else if(strncmp(procfs_buffer, "3", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0],smp_processor_id());
    }
}

```

```

        asm volatile("or 6,6");
    }
    else if(strncmp(procfs_buffer, "4", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0],smp_processor_id());
        asm volatile("or 2,2");
    }
    else if(strncmp(procfs_buffer, "5", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0], smp_processor_id());
        asm volatile("or 5,5");
    }
    else if(strncmp(procfs_buffer, "6", 1)==0)
    {
        printk(KERN_INFO "Changing to SMT priority to %c on cpu %u\n",procfs_buffer[0], smp_processor_id());
        asm volatile("or 3,3");
    }
    else
    {
        printk(KERN_INFO "Invalid SMT priority %c requested, please include 1,2,3,4,5, or 6 in the first character of
file\n",procfs_buffer[0]);
    }
    return procfs_buffer_size;
}

```

```

/*
 * This function decides whether to allow an operation
 * (return zero) or not allow it (return a non-zero
 * which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file
 * permissions. The permissions returned by ls -l are
 * for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op, struct nameidata *foo)
{
    /*
     * We allow everybody to read from our module, but
     * only root (uid 0) may write to it
     */

    if (op == 4 || (op == 2 && current->euid == 0))
        return 0;
    /*
     * If it's anything else, access is denied
     */
    return -EACCES;
}
/*
 * The file is opened - we don't really care about
 * that, but it does mean we need to increment the
 * module's reference count.
 */

```

```

int procfs_open(struct inode *inode, struct file *file)
{
    try_module_get(THIS_MODULE);
    return 0;
}
/*
 * The file is closed - again, interesting only because
 * of the reference count.
 */
int procfs_close(struct inode *inode, struct file *file)
{
    module_put(THIS_MODULE);
    return 0; /* success */
}

static struct file_operations File_Ops_4_Our_Proc_File = {
    .read = procfs_read,
    .write = procfs_write,
    .open = procfs_open,
    .release = procfs_close,
};
/*
 * Inode operations for our proc file. We need it so
 * we'll have some place to specify the file operations
 * structure we want to use, and the function we use for
 * permissions. It's also possible to specify functions
 * to be called for anything else which could be done to
 * an inode (although we don't bother, we just put
 * NULL).
 */
static struct inode_operations Inode_Ops_4_Our_Proc_File = {
    .permission = module_permission, /* check for permissions */
};
/** Module initialization and cleanup*/
int init_module(){
    /* create the /proc file */
    Our_Proc_File = create_proc_entry(PROC_ENTRY_FILENAME, 0644, NULL);
    /* check if the /proc file was created successfully */
    if (Our_Proc_File == NULL){
        printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
            PROC_ENTRY_FILENAME);
        return -ENOMEM;
    }
    Our_Proc_File->owner = THIS_MODULE;
    Our_Proc_File->proc_iops = &Inode_Ops_4_Our_Proc_File;
    Our_Proc_File->proc_fops = &File_Ops_4_Our_Proc_File;
    Our_Proc_File->mode = S_IFREG | S_IRUGO | S_IWUSR;
    Our_Proc_File->uid = 0;
    Our_Proc_File->gid = 0;
    Our_Proc_File->size = 80;
    printk(KERN_ALERT "/proc/%s created\n", PROC_ENTRY_FILENAME);
    return 0; /* success */
}
void cleanup_module(){
    remove_proc_entry(PROC_ENTRY_FILENAME, &proc_root);
    printk(KERN_INFO "/proc/%s removed\n", PROC_ENTRY_FILENAME);
}

```

Appendix E: Time Interval Data

This appendix has the data for the experiments done to determine the signature time interval.

The data for PETSc KSP is the file Time_interval_KSP.xlsx in the folder Appendix_files, or follow this link: [Appendix_files\Time_interval_KSP.xlsx](#)

The data for Spec CPU2006 is the file Time_interval_SpecCPU2006.xlsx in the folder Appendix_files, or follow this link: [Appendix_files\Time_interval_SpecCPU2006.xlsx](#)

The data for NPB is the file Time_interval_NPB.xlsx in the folder Appendix_files, or follow this link: [Appendix_files\Time_interval_NPB.xlsx](#)

The data for the selection of signature time interval is the file Overall Time intervals.xlsx in folder Appendix_files or follow this link: [Appendix_files\Overall Time intervals.xlsx](#)

Appendix F: Signature Data

This appendix has the data for the signatures found in the signature-generating applications.

First, we performed experiments to determine the number of levels we would use for each resource.

The first level we tried was two levels for the four resources, and the resulting signatures found using this definition is listed in file Partn=2f.xlsx in folder Appendix_files, or follow this link: [Appendix_files/Partn=2f.xlsx](#)

The next level we tried was 10 levels for the four resources, and the resulting signatures found using this definition is listed in file Partn=10f.xlsx in folder Appendix_files, or follow this link: [Appendix_files/Partn=10f.xlsx](#)

The utilization data that was shown in Table 6.3 was derived from the file called Util_data.xlsx in folder Appendix_files or follow this link: [Appendix_files\Util_data.xlsx](#)

The utilization data for the each of the benchmark suite can be accessed in Appendix_files\Utilization_by_benchmarks folder and the name of the files are shown below:

- Utilization data for NPB3.2 Serial benchmarks is in file NPB.xlsx or follow this link: [Appendix_files\Utilization by benchmarks\NPB.xlsx](#)
- Utilization data for KSP benchmarks is in file KSP.xlsx or follow this link: [Appendix_files\Utilization by benchmarks\KSP.xlsx](#)
- Utilization data for Spec CPU2006 integer benchmarks is in file Int2006.xlsx or follow this link: [Appendix_files\Utilization by benchmarks\Int2006.xlsx](#)
- Utilization data for Spec CPU2006 floating-point intensive benchmarks is in file FP2006.xlsx or follow this link: [Appendix_files\Utilization by benchmarks\FP2006.xlsx](#)

Data showing absence of instructions between dispatch and completion can be accessed in the Appendix_files folder in the file KSP_GCT_empty.xlsx at the link [Appendix_files\KSP_GCT_empty.xlsx](#)

Appendix G: Prediction and Validation Phase Results

This appendix has the data for the prediction table for the 17 signatures of the prediction signature set in the file Prediction_table.xlsx in folder Appendix_file or follows this link: [Appendix_files\Prediction_Table.xlsx](#).

The data is in the file called Dominating_Signature_thresholds.xlsx in folder Appendix_files or follow this link: [Appendix_files\Dominating_Signature_thresholds.xlsx](#)

Table G.1: Applications with Dominating Signatures

Data Set	Application	Signature
ref	445.gobmk	F1I3C1T1
ref	458.sjeng	F1I3C1T1
ref	453.povray	F1I2C1T1
ref	459.gemsFDTD	F1I1C1T1
ref	437.leslie3d	F3I2C2T1
ref	462.libquantum	F1I3C1T1
A	bt-mz.A	F5I1C1T1
A	lu-mz.A	F4I1C2T1
B	bt-mz.B	F5I1C1T1
B	lu-mz.B	F4I1C2T1
B	sp-mz.B	F4I1C1T1
C	bt-mz.C	F5I1C1T1
C	sp-mz.C	F3I1C1T1
arco3	bcds	F1I2C1T1
arco4	bicg	F1I2C1T1
arco6	bicg	F1I2C1T1
cfid.2.10	bicg	F1I2C1T1
poisson3	bicg	F1I2C1T1
arco3	chebychev	F1I2C1T1
cfid.1.10	chebychev	F1I1C1T1
arco3	cr	F1I2C1T1
arco4	cr	F1I2C1T1
arco3	gmres	F1I2C1T1
arco4	gmres	F1I2C1T1
poisson3	gmres	F1I2C1T1
arco4	lsqr	F1I3C1T1
arco6	lsqr	F1I3C1T1
poisson3	lsqr	F1I3C1T1
arco3	richardson	F1I2C1T1
cfid.2.10	richardson	F1I1C1T1
arco3	tcqmr	F1I2C1T1

arco3	tfqmr	F1I2C1T1
poisson3	tfqmr	F1I2C1T1
cfld.1.10	lsqr	F1I3C1T1

Appendix H: Simulation Data

This appendix has the data for simulation data that was used to identify the set of critical resources.

The summary of the data is presented below in Tables H.1 and H.2, whereas the entire data can be viewed in the file AppCharacts_Summary.xlsx in the folder Appendix_files, or follow this link: [Appendix_files\AppCharacts_Summary.xlsx](#)

Table H.1 Average CPU Cycles Servicing Misses in the TLB/Caches by a Benchmark Suite

	Average % iTLB Miss Cycles	Average % dTLB Miss Cycles	Average % iCache Miss Cycles	Average % dCache Miss Cycles
Int2000	0	0	0	10
FP2000	0	0	0	5
stream2	0	2	0	3
Int2006	0	5	3	28
FP2006	0	3	0	19
Lmbench	0	2	0	10

Table H.2 Average CPU cycles using FPU and FXU by a Benchmark Suite

	Average %FPU Utilization	Average %FXU Utilization
Int2000	0.01067763	36.0364539
FP2000	5.29665645	35.2276916
stream2	9.28158675	29.7958448
Int2006	0.00001175	29.0802671
FP2006	17.232863	11.0687445
Lmbench	0.000784	52.7178178

Curriculum Vita

Mitesh R. Meswani, the second son of Mr. Ramesh Meswani and Mrs. Kaumudini Meswani, was born on Dec 3rd, 1977. He earned his Bachelor's degree, from the University of Mumbai, India, in 1999 and his Master's Degree from the University of Texas at El Paso, Texas in 2000. While pursuing his Master's degree he worked as a Teaching Assistant at the Department of Computer Science between 1999 and 2000. After completing his Master's degree, he worked as a Software Engineer at Motorola Corporation in Fort Worth Texas during 2001. He joined the Computer Science Ph.D. program at the University of Texas at El Paso in Spring 2002. While pursuing his Ph.D. degree he worked as a Teaching Assistant and Research Assistant for the Department of Computer Science between 2002 and 2005, and as a Research Associate for the Department of Computer Science between 2006 and 2009. He did a summer internship at TNRCC Corporation in Austin, Texas, in 2002 and a Fall internship at Intel Corporation in Portland, Oregon in 2004. In Fall 2000, he was inducted in Upsilon Pi Epsilon (UPE), an International Honor Society for the Computing Sciences, and he is a member of ACM and IEEE since 2007.

During his graduate studies, he has published four technical papers in workshops. In addition, he presented a talk at the doctoral show case at SC08 Conference in Austin, Texas, in 2008 and presented a student poster at the LCI conference in Boulder, Colorado in 2009. Further information can be obtained by visiting his home page at <http://www.cs.utep.edu/students/mitesh>

Permanent Address:

61/62 J. K. Mehta Road,
Flat # 21, Premsagar Building
Santacruz West, Mumbai,
Maharashtra, India, 400054

This dissertation was typed by Mitesh R. Meswani