2018-01-01

A Framework To Audit Scheduling Events In The Linux Operating System

Edward G. Hudgins
University of Texas at El Paso, edwardghudgins@gmail.com

A FRAMEWORK TO AUDIT SCHEDULING EVENTS IN THE LINUX OPERATING SYSTEM

EDWARD GARWOOD HUDGINS

Master's Program in Computer Science

Upon the plains of hesitation

bleach the bones of countless millions

who on the threshold of victory

sat down to wait

and awaiting they died

A FRAMEWORK TO AUDIT SCHEDULING EVENTS IN THE LINUX OPERATING SYSTEM

by

EDWARD GARWORD HUDGINS, BSCS

THESIS

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

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2018

Acknowledgements

To my wife, Janette:

To the woman that is everything I need when I need it, even though I may not always agree of what form that may be. I know I have been impossible. I hope, that one day, I may be able repay the dedication and patience you have shown me. I love you so much, Mi Gatita, God's greatest gift. Always, forever, for eternity, for life.

To my family:

Your kindness and dedication made this all this possible. To the family that raised me and to the family that has taken me into their fold; words cannot express my gratitude. There is no sweeter feeling than knowing you have two families to which you can rely on.

Dr. Freudenthal, Dr. Salamah, Dr. Romero and the Computer Science faculty and staff

Thank you for the guidance you have given me. I am happy to not only to have called you my professors, but it is my honor to call you my friends.

To all my fellow Robustos, past and present,

You have made my time here at UTEP unforgettable. Thank you for making the robust lab a home.

To all the people that I have had the honor of calling my friends

Thank you for making this strenuous journey enjoyable.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Soft real-time systems have responsiveness requirements that are desirable but not critical for operational effectiveness. Due to human sensitivity to interface delays on gesture-driven devices, mobile devices are a common case of soft-real time systems. Mobile systems generally do not incorporate real-time schedulers, but instead utilize over-provisioning and a variety of scheduling heuristics to generally provide acceptable responsiveness. [11,12] These devices are highly multi-programmed energy limitations on mobile limit the extent of overprovisioning, thereby increasing the sensitivity of system behavior to the interaction of active applications and scheduling heuristics.

Operating systems generally only collect aggregate statistics on scheduler behavior. As described in Section 1.3, it is generally impractical to use aggregate metrics to associate an interface stutter with the program behaviors and scheduler heuristics that caused it.

This thesis describes a new scheduler logging framework named "Integrated Process Scheduler Archiver" (IPSA) intended to assist with this analysis. The remainder of this chapter expands on the motivations for this research. Chapter 2 describes the logging mechanism's architecture. Chapter 3 motivates and describes strategies employed to validate the accuracy of timing information recorded within log entries and characterize IPSA's perturbation of task scheduling. Finally, Chapter 4 summarizes results of this thesis research including its limitations. In addition, it includes a summary of potential future investigations. Appendix A contains a brief overview of the modifications to the operating system kernel. Appendix B contains a summary of the information extracted for the experiments described in Chapter 3.

1.1 Monitoring and analysis of process scheduling

This thesis describes the modification to the Linux scheduler used by Android. For notational convenience, we describe an operating system's schedulable entity as a task. The task data and state transitions monitored by IPSA are common to most operating systems and therefore we expect that IPSA's approach can be applied to other preemptive schedulers.

Scheduling is trivial when a system has less runnable tasks than available cores. When this is not the case, heuristics are used to determine execution schedule and therefore indirectly determine their order of completion. The assumption motivating the development of IPSA is that a logged sequence of scheduling decisions can provide insight into the interaction of scheduler policy and task behavior.

1.2 Relevance of operational delay

Each interface gesture (a stimulation) results execution bursts by multiple interdependent tasks, eventually resulting in an observable response. A sustained delay in the execution of the tasks on the critical path to the rendering of a response to a stimulation can create interface stutter. [10,11] Operational delay (also known as operational latency) is defined as the time interval between stimulation and response and is a performance metric for interactive systems. [10,11] Scheduling policy resulting in perceptible operational delay (observable as interface stutter) can degrade usability by because it impedes the user's ability to comprehend the result of a prior stimulation. [9]

1.3 Why are existing tools insufficient?

The purpose for this research is to collect timing information about each task's significant scheduling events. This data is anticipated to enable analysis of scheduling decisions. Existing data collection mechanisms provide some context to the state of the ready queue but are unsuitable for such analysis. For example, The Free Software Foundation's *ps* utility provides summary information about all tasks in the system. [18] Some artifacts of scheduling history could be exposed through analysis of frequent dumps of this information. However, this approach would be unlikely to yield a representative history of scheduling events because (1) the frequent execution of *ps* would likely interfere with the scheduling of the tasks being monitored and (2) despite the frequent execution of *ps*, relevant scheduling events might not be exposed.

Chapter 2: System Design

As described in Chapter 1, IPSA logs scheduling decisions. This chapter provides an overview of basic scheduling principles, the design of IPSA, and obstacles encountered during implementation.

2.0 Process Scheduling

This thesis defines a scheduling event as any function that results in one or multiple tasks transitioning task state. Figure 2.1 illustrates the Canonical Process State Diagram; the scheduling events defined are c*reate, scheduled, preempt, unblocked, block,* and *terminate* (when a task is completed/terminated).

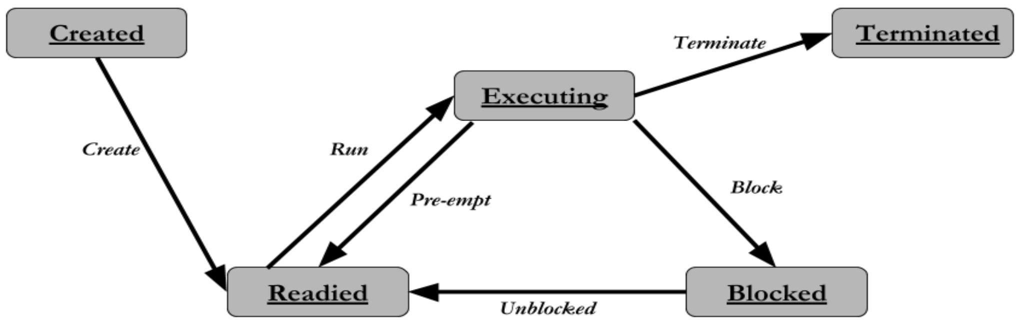

FIGURE 2.1: PROCESS STATE DIAGRAM

Operating systems may have one function handle multiple scheduling events. Figure 2.2 is an illustration of the Canonical Process State Diagram annotated with the names of functions within Linux that implement state transitions. In Linux, readied tasks are not removed from the ready queue when they are executing. As a result, the scheduler's function *dequeue* is called (only) when a task becoming *blocked* or *terminated,* and the purpose for this research is to collect timing information about each task's significant scheduling events. This data is anticipated to enable analysis of scheduling decisions. Existing data collection mechanisms provide some context to the state of the ready queue but are unsuitable for such analysis. For example, The Free Software Foundation's *ps* utility provides summary information about all tasks in the system. [18] Some artifacts of scheduling history could be exposed through analysis of frequent dumps of this information. However, this approach would be unlikely to yield a representative history of scheduling events because (1) the frequent execution of *ps* would likely interfere with the scheduling of the tasks being monitored and (2) despite the frequent execution of *ps*, relevant scheduling events might not be exposed. The function *enqueue* is called when a task becomes *readied*. For notational convenience, this thesis refers to the calling of these functions as "enqueue" and "dequeue" events.

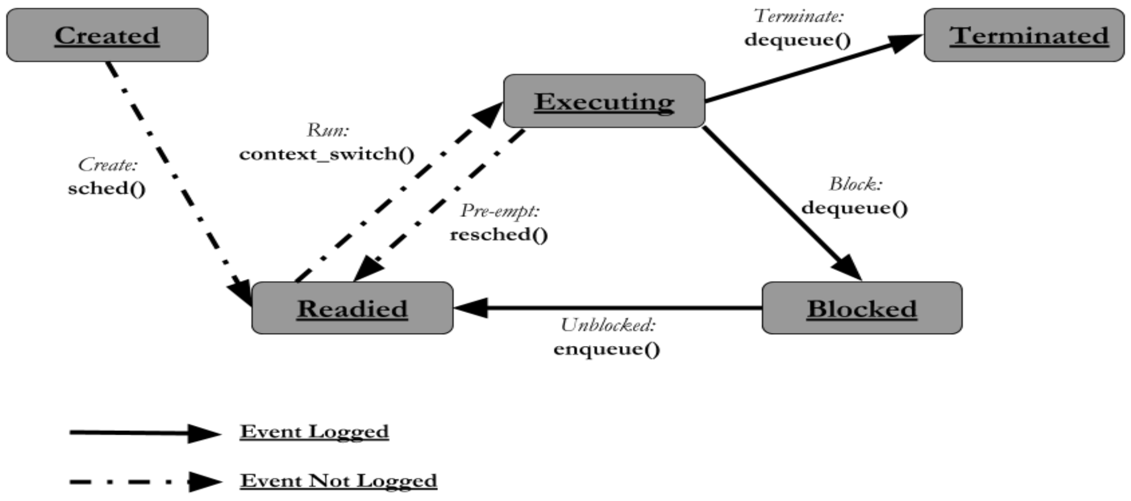

FIGURE 2.2: PROCESS STATE DIAGRAM – LINUX SCHEDULING EVENTS

2.1 Integrated Process Scheduler Archiver (IPSA)

IPSA logs timing-sensitive decisions in a manner that enables offline analysis. The design employed in IPSA uses multiple strategies intended to limit its impact on the system caused by the act of observation during that system's operation, also known as the observer effect. [17] For example, timing-sensitive recording of log entries into kernel memory is decoupled from their (less timing sensitive) extraction to user space.

2.1.1 Obtaining relevant information

As described in Chapter 1, IPSA is a lightweight framework that exposes significant aspects of scheduling history for offline analysis. Due to frequent preemption in Linux, tasks may be preempted multiple times in the interval between becoming readied and becoming blocked. Rather than incurring the overhead of recording and exporting all of these short execution bursts, IPSA instead only logs *unblocked* (enqueue), *block* and *terminate* (dequeue) events. To enable analysis of individual task execution history, each log entry includes the transitioning task's state, total CPU time, priority, and scheduling event count. (see Appendix B)

2.1.2 The IPSA subsystem design

IPSA's data is recorded by the kernel and transmitted to a proc file when read via a bounded buffer (labeled as RQE_Queue in Figures 2.3). The context diagram in Figure 2.3 displays how components of IPSA is coupled to the kernel and user-mode daemons.

Enqueue and dequeue events trigger the generation of log entries (see Figure 2.2). Each entry includes information about the affected task and contextual data about the ready queue at the time of the event (see Appendix B). Should queue capacity be exceeded when a user-mode daemon performs a read on the proc file (see Section 2.1.3), the oldest entries are overwritten.

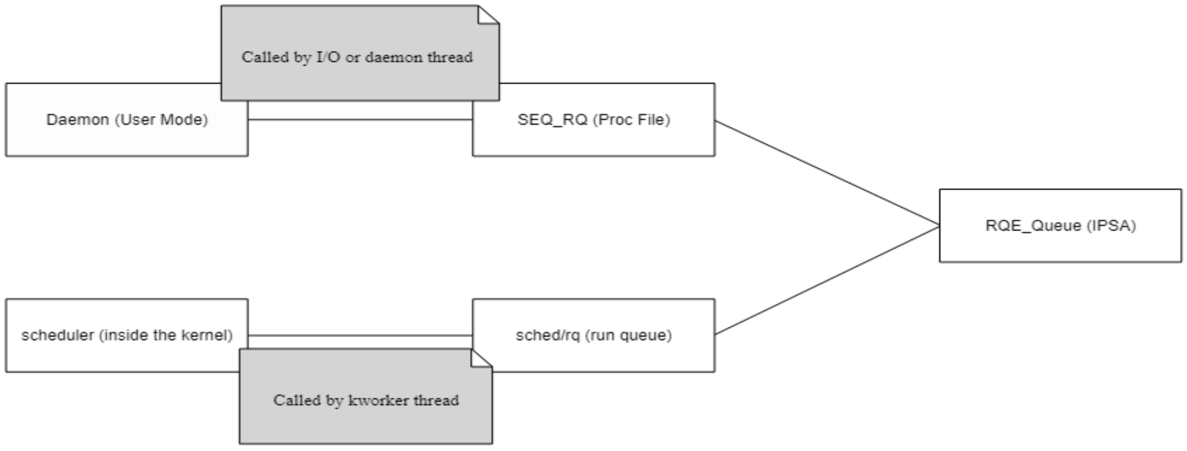

FIGURE 2.3: CONTEXT DIAGRAM – IPSA

2.1.3 Data Extraction

IPSA provides a proc file entry that exposes logged scheduler events to user programs. This allows user-mode programs the flexibility to perform either immediate or offline (post-mortem) analysis. Data can be collected continuously or in response to detection of an anomalous or otherwise interesting event.

2.1.4 Memory footprint

The total memory size of this implementation of IPSA is approximately 14.6 megabytes. This breaks down as follows: IPSA has a queue capacity of 35,000 Entries ($E = 35000$) per each Ready queue available to the device. The device used in this research had 2 ready queues available ($R = 2$). The Size of each entry is 208 bytes ($S = 208$). Finally, the overhead of the Control variables for this implementation is 80 Bytes ($C = 80$). Thus, memory footprint (M) of IPSA's components (described in Section 2.1.2) is:

$$M = \big((E * S) * R\big) + C$$

2.1.5 Indirectly logging kernel thread scheduling events

As described above, entries recorded by IPSA are exposed to user programs via a proc file. As illustrated in Figure 2.3, the Linux kernel's soft interrupts are executed in the context of its "kworker" thread. [19] Kworker is scheduled in the same manner as conventional threads. If the scheduling of kworker was logged by IPSA, each read from the proc filesystem would trigger the generation (and enqueueing) of additional entries. To prevent a circular dependency that would create an infinite cascade of entries, IPSA instead only records and exposes cumulative summaries of kworker execution events.

Chapter 3: Experimental Design

This chapter describes the methodology used to determine the accuracy of data collected by IPSA and measures perturbation of throughput caused by its collection. As described in Chapter 2, IPSA's data collection mechanism was integrated into Linux's scheduler. This section begins with a description of how the temporal data already exposed by the Linux operating system was used to validate IPSA. This is followed by an examination of throughput perturbation caused by IPSA.

3.1 Validation

As described in Chapter 2, IPSA log entries are derived from per-task data already maintained by the operating system. Experiments in this section are designed to confirm that CPU execution-time data within IPSA's scheduling event log entries are properly extracted. This is achieved by comparing user and elapsed time aggregated from the IPSA logs to the commonly used (and presumably accurate) aggregate timing data already exposed to utilities such as Free Software Foundation's Time utility. [18]

3.1.1 Experiment setup

Experiments were conducted using a single-core Galaxy Nexus (Android 4.4.4 augmented with IPSA) with CPU frequency locked to 700 Mhz. The experiments primarily measure and compare user-mode and elapsed time for multiple executions of a compute intensive matrix multiplication performed by a benchmark program created for this purpose named *matrixMultipy*.

Two groups of experiments are described. The first group are (essentially) run uni-programmed with little competition from other processes. The second explicitly examines the impact of significant multiprogramming upon timing measurements by simultaneously running two instances of matrixMultiply.

Results from these experiments indicate that IPSA's timing measurements are consistent with system timing measurements exposed by the Time utility. [18] Section 3.1.2 and 3.1.3 describe the expected and measured results of these experiments, which are highly consistent and summarized within Tables 3.1 and 3.2. Those tables also indicate (1) average and (2) maximal variation from the mean (as a percentage) and the number of times MatrixMultiply is preempted during its execution (the column is labeled "context switches").

3.1.2 Uni-programmed Experiments

The purpose of this experiment is to verify that aggregate execution time measurements from IPSA and system are consistent with aggregate process timing data already exposed by the Free Software Foundation's Time utility [14].

Methodology This experiment compares execution time of running matrixMultiply with little (if any) contention from other programs. Three measurements are collected compared: the aggregate (1) "user" and (2) elapsed (wall-clock) time reported by Time, and (3) the sum of user-mode CPU time recorded in IPSA's scheduling logs.

Expected results If there is negligible contention from other processes, user and elapsed times for matrixMultiply should be similar. More significantly, there should be negligible difference between user time (1) aggregated from IPSA's scheduling log and (2) exposed by Time.

3.1.3 Multi-programmed Experiments

The purpose of this experiment is to verify that aggregate execution time measurements from IPSA are consistent with system-collected timing measurements in the presence of significant multitasking. This section describes the methodology, expectations, and analysis of this experiment.

Methodology In this experiment, two instances of matrixMultiply are executed concurrently. This experiment uses the findings from the experiment described in Section 3.1.2 and compares them to the data collected from the execution of both concurrent instances of matrixMultiply. Like in the experiment described in Section 3.1.2, this experiment compares the total elapsed (wall-clock) time of program execution and the user-mode CPU time for both concurrent instances of matrixMultiply exposed by IPSA and reported by Linux.

Expected results. In this experiment, two instances of matrixMultiply are run concurrently with minimal contention from other tasks. Data from this experiment is tabulated in the second row of Table 3.1. Ideally, an independent CPU-bound task will require the same amount of CPU time independent of multitasking, and if two such tasks are multitasked, their total execution time should be double the execution time for one process. Differences from ideal timings are tabulated in the third row of Table 3.1.

Data Analysis The difference between mean user time aggregated from IPSA's scheduling log and Linux's reported user-time is ≈0.1% (see Table 3.1). This is less than the (small) measurement variations for both metrics among runs. Timings in IPSA's scheduling log are consistent with user-time measured by Linux. Total user time for both instances exposed by IPSA and elapsed time exposed by Linux differ by about ≈5.3%. This was similar to findings in Table 3.1, which had a difference of ≈5.5%. Total elapsed time for the competing tasks increased by an average of ≈2.01 times the elapsed time of the single process.

TABLE 3.1: UNI-PROGRAMMED AND MULTI-PROGRAMMED EXPERIMENTS

Experiment	**User (s) - IPSA**	**User (s) – Linux**	**Elapsed (s) - Linux**	**Context Switches**
Single CPU bound task	27.03 (±0.2%)	27.01 (±0.3%)	28.61 (±1.1%)	1662 (±13.4%)
Two CPU bound tasks	27.11 (±0.3%)	27.10 (±0.3%)	57.48 (±0.7%)	4075 (±9.1%)
Difference from ideal	0.003%	0.004%	0.009%	

3.2 Throughput Perturbation

The purpose of this experiment is to measure and characterize IPSA's impact upon the rate of progress of compute intensive tasks and identify extraction conditions where this progress is minimally perturbed.

As explained in Chapter 2, extraction of IPSA's logs are performed by user processes. This initial implementation of the IPSA subsystem does not provide any such tools beyond a Bash script that uses the FSF's Cat program to periodically copy logs.

3.2.1 Experimental Setup

Experiments were conducted using a single-core Galaxy Nexus (Android 4.4.4. augmented with IPSA). The two experiments described in this section primarily measure two benchmarks; a compute-intensive program that requires few system calls and an I/O intensive program.

Two groups of measurements are described. The first group are performed with the device locked at 700 Mhz while performing these benchmarks to characterize the observer effect on the scheduler while utilizing IPSA. The second examines the impact of varying speeds of the device's processor. The results provided in Table 3.3 are an average for each corresponding condition. The percentage provided is the margin of difference from this mean.

3.2.2 Throughput Perturbation at 700 Mhz

This experiment has two objectives; to measure task throughput and to establish a control group for the experiment described in Section 3.2.3. This experiment compares the results from three related experiments: a baseline group, IPSA's logging without extraction, and IPSA's logging with extraction. Benchmark completion times are summarized in Tables 3.2 which indicate (1) average and (2) maximal variation from the mean (as a percentage).

Baseline group (BASE) This experiment approximates the device without any modifications required to utilize logging or extraction. This is done as described in Section 3.2.1. Results from these experiments are labelled BASE (baseline) in Table 3.2. This is an approximation of an unmodified device since the IPSA remains incorporated within the OS kernel. In particular, kernel memory size is the increased by the memory footprint of IPSA (described in Chapter 2) and a flag comparison is performed when an enqueue or dequeue event occurs (described in Chapter 2).

IPSA's logging without extraction (NoEx) The purpose of this experiment is to measure throughput perturbation caused by IPSA's logging functionality independent of data migration to the user-space. Thus, logs in the buffer are simply disregarded. This thesis refers to this as NoEx (no extraction) in Table 3.2.

IPSA's logging with extraction (SEC1, CONT) This experiment measures the throughput perturbation caused by extraction at one second (SEC1) intervals, and also and with no delay between extraction (CONT). The one second interval was chosen as a potential reasonable interval for, while the continuous (no delay) was chosen to model the worst-case scenario described in Chapter 1.

As described in Chapter 2, only IPSA's logging infrastructure is integrated into the kernel and data extraction is delegated to a user program. This experiment measures the impact on throughput of extraction of IPSA's logs. Extraction was performed by a user-mode daemon at predetermined intervals; referred to as a delay. The daemon performs a copy on the proc file (described in Chapter 2) and delays for a period of time before performing the next copy. Note, the size of logs returned, nor the time required to read were analyze during this experiment.

Results Benchmarks strongly suggests that there are only small differences between the execution time when extraction is not performed, and logging disabled when executing a long running compute-bound job. In this context, the perturbation of the execution time is minimal when the interval between extractions is 1 second. In contrast, when there is no delay between extractions, execution time is almost doubled.

TABLE 3.2: THROUGHPUT PERTURBATION – 700 MHZ

Benchmark	**BASE (s)**	**NoEx (s)**	**SEC1 (s)**	**CONT (s)**
Computational avg (diff %)	26.28 (±0.1%)	26.43 (±0.2%)	26.63 (±0.1%)	47.34 (±0.2%)
I/O[1] avg (diff %)	124.83 (±0.03%)	125.45 (±0.2%)	126.18 (±0.04%)	210.64 (±0.2%)

3.2.3 Throughput Perturbation at varying speeds

The objective of this experiment is to characterize IPSA's impact on throughput at various CPU frequencies and extraction intervals. Results were graphed for empirical analysis. The experiments performed in Section 3.2.2 were performed on a single core device locked at 700 Mhz and this experiment uses those results as a baseline for comparison.

Both the compute-intensive and the I/O-intensive were performed at the device's 4 available clock frequencies: 350 Mhz, 700 Mhz, 920 Mhz, and 1.2 Ghz. As described in Section 3.2.2, throughput perturbation is measured at various extraction intervals. These extraction intervals provide an estimation of throughput perturbation as extraction approaches continuous.

IPSA disabled These experiments (with IPSA disabled) approximate execution conditions of an unmodified operating system. This is an approximation of an unmodified device since the IPSA remains incorporated within the OS kernel. In particular, kernel memory size is

[1] The I/O benchmark opens a file and then issues single-byte writes to the end of the file. These CPU-bound results are consistent with the filesystem's write-behind semantics which likely cause the the syscall interface and buffer cache (rather than the storage device) to be the likely throughput bottlenecks.

the increased by the memory footprint of IPSA (described in Chapter 2) and a flag comparison is performed when an enqueue or dequeue event occurs (described in Chapter 2).

IPSA's throughput perturbation while approaching continuous extraction As described in the previous section, Extraction was performed by a user-mode daemon at predetermined intervals, referred to as a delay. The daemon performs a copy on the proc file, which extracts all existing entries in the buffer (described in Chapter 2), and delays for a period of time before performing the next copy. This experiment analyzes delays of 1 second, 0.5 seconds, 0.25 seconds, and 0.1 seconds. Expectations are that as delays are shortened, perturbation increases.

Results The plots in figures 3.0 and 3.1 indicate benchmark execution time at various extraction intervals and CPU frequencies. The duration of *IPSA disabled* runs is inversely proportional to CPU clock frequency for both CPU and I/O intensive experiments. This indicates that throughput for both experiments is effectively limited by CPU clock frequency. Shorter intervals between extractions (labelled "delay" in figures 3.0 and 3.1) result in dilation of task execution time (labelled User-Time in figures 3.0 and 3.1). Execution time dilation for the lowest clock frequency (350 MHz) is particularly dramatic, especially when the interval between extractions is short.

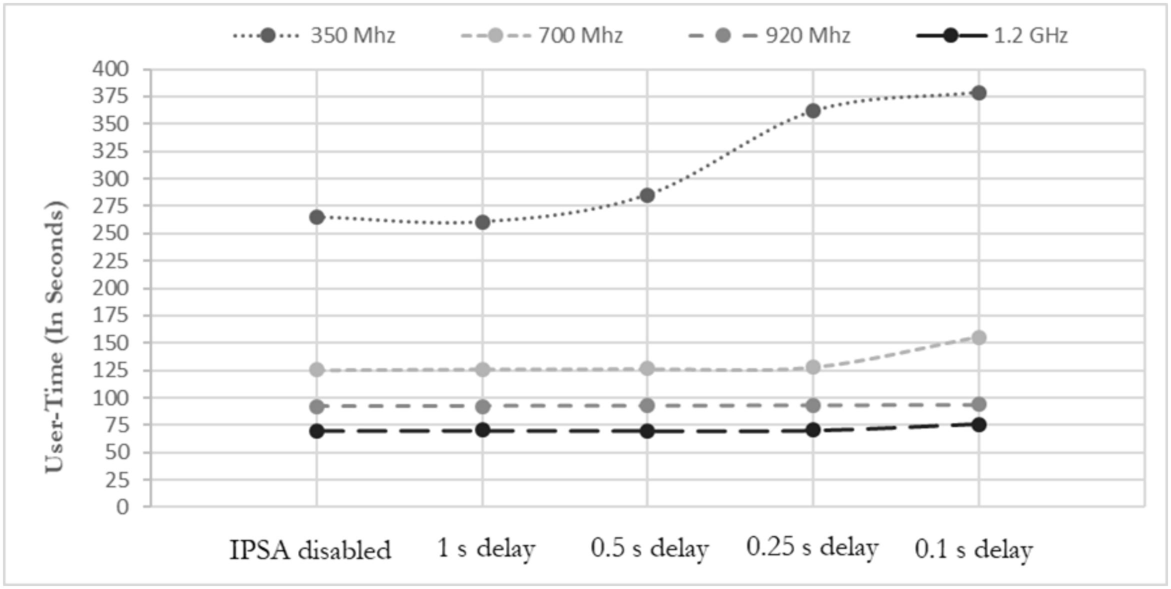

Figure 3.1: MEAN - I/O BENCHMARK AT VARYING SPEEDS

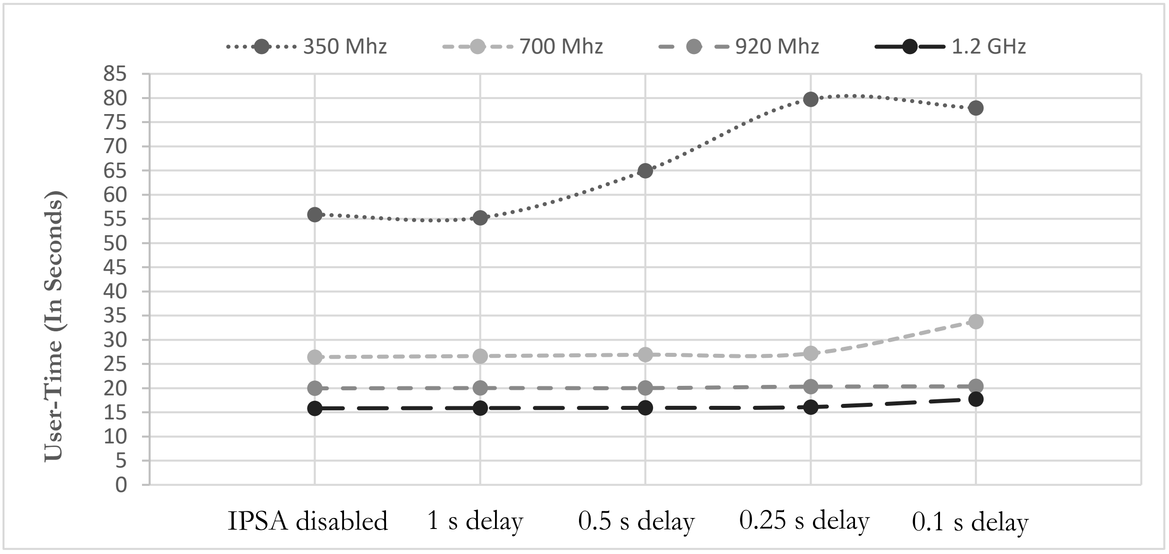

FIGURE 3.2: MEAN - COMPUTATIONAL BENCHMARK AT VARYING SPEEDS

Chapter 4: Discussion

4.1 Synopsis

Results from validation experiments described in Chapter 3 indicate that the timing data collected by IPSA is consistent with aggregate timing data already collected by Linux tools widely utilized in research. [14] Results collected during benchmarking suggest that logging to IPSA's in-kernel event queue (without extraction) has minimal impact on scheduling. As expected, when the processor is over-provisioned relative to offered load, (e.g. at higher clock frequencies) IPSA's throughput perturbation diminishes.

4.2 Limitations

This section describes the limitations of the research discussed in this thesis. IPSA does not record Linux's internal kworker thread's scheduling events. These scheduling events (as described in Chapter 2) caused an infinite cascade of logs. To prevent this, these events were counted.

The methodology in this research was limited in three ways (as described in Chapter 3). First, the speed of the processor was locked to a predetermined speed; this prevented a complete analysis of IPSA's interaction with dynamic voltage and frequency scaling. Second, the device was locked to a single-core, aggregations such as *weight* (a metric used for load-balancing between cores) were not logged or analyzed. Finally, extraction frequency was selected prior to analysis to avoid further perturbing the system. The duration of each extraction or the dynamic extraction intervals were not considered for this research.

4.3 Future work

Android is built upon the Linux scheduler. To minimize interface delays, Android tasks that implement UI components are assigned higher scheduling priority. Completely fair scheduling (Linux's scheduler for normal tasks) incorporates both priority and fairness mechanisms that prevent task starvation. The measurement and analysis of operational delay could be used to determine the feasibility of a CFS approach in power-limited, delay intolerant devices (such as smartphones and tablets) by utilizing IPSA to identify such phenomenon. When the detection of a perceptual interface delay occurs, IPSA could be modified to perform a retrospective extraction (extraction of logs during the time interval of the perceptual operational delay) to expose interactions task behavior and the scheduler that resulted in the delay.

References

[1] Tanenbaum, A & Bos, H. (2014). Modern operating systems Prentice Hall Press

[2] Li, T., Baumberger, D., & Hahn, S. (2009, February). Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In ACM Sigplan Notices (Vol. 44, No. 4, pp. 65–74). ACM.

[3] Aas, J. (2005). Understanding the Linux 2.6. 8.1 CPU scheduler. Retrieved Oct,16,(pp. 1–38).

[4] Wong, C. S., Tan, I., Kumari, R. D., & Wey, F. (2008). Towards achieving fairness in the Linux scheduler. ACM SIGOPS Operating Systems Review, 42(5), (pp. 34–43).

[5] Wong, C. S., Tan, I. K. T., Kumari, R. D., Lam, J. W., & Fun, W. (2008, August). Fairness and interactive performance of o (1) and cfs linux kernel schedulers. In 2008 International Symposium on Information Technology (Vol. 4, pp. 1–8). IEEE.

[6] Maia, C., Nogueira, L. M., & Pinho, L. M. (2010, July). Evaluating android os for embedded real-time systems. In 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (pp. 63–70).

[7] Maker, F., & Chan, Y. H. (2009). A survey on Android vs. Linux. University of California, (pp. 1–10).

[8] Jota, R., Ng, A., Dietz, P., & Wigdor, D. (2013, April). How fast is fast enough?: a study of the effects of latency in direct-touch pointing tasks. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2291–2300). ACM.

[9] Endo, Y., Wang, Z., Chen, J. B., & Seltzer, M. I. (1996). Using latency to evaluate interactive system performance. ACM SIGOPS Operating Systems Review, (pp. 185–199).

[10] Ng, A., Lepinski, J., Wigdor, D., Sanders, S., & Dietz, P. (2012, October). Designing for low-latency direct-touch input.In Proceedings of the 25th annual ACM symposium on User interface software and technology (pp. 453–464). ACM.

[11] Allison, R. S., Harris, L. R., Jenkin, M., Jasiobedzka, U., & Zacher, J. E. (2001, March). Tolerance of temporal delay in virtual environments. In Virtual Reality, 2001. Proceedings. IEEE (pp. 247–254). IEEE.

[12] Pavlovych, A., & Stuerzlinger, W. (2009, July). The tradeoff between spatial jitter and latency in pointing tasks. In Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems (pp. 187–196). ACM.

[13] Moon, S. B., Skelly, P., & Towsley, D. (1999, March). Estimation and removal of clock skew from network delay measurements. In INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Vol. 1, pp. 227-234). IEEE.

[14] Love, R. (2010). Linux kernel development. Pearson Education.

[15] Shneiderman, B. (2010). Designing the user interface: strategies for effective human computer interaction. Pearson Education India.

[16] Soares, L., & Stumm, M. (2010, October). FlexSC: Flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (pp. 33-46). USENIX Association.

[17] Mytkowicz, T., Sweeney, P. F Hauswirth, M., & Diwan, A. (2008). Observer effect and measurement bias in performance analysis. Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder.

[18] The Free Software Foundation (1998) https://www.gnu.org/

[19] Cheng, L., Rao, J., & Lau, F. (2016, April). vScale: automatic and efficient processor scaling for SMP virtual machines. *In Proceedings of the Eleventh European Conference on Computer Systems* (p. 2). ACM.

Appendix A: IPSA Implementation (Android)

A.0 Overview

This appendix provides an overview of the IPSA's key components. Sections A.1 and A.2 describe how the Android scheduler was modified. The repository listed below contains the source code.

Repo: https://github.com/ipsaAndroid/android_kernel_samsung_tuna/tree/cm-11.0

A.1 Modifications to kernel/sched.c

RQE_Q_CAP (Constant) This constant defines the size of the kernel buffer for IPSA

RQE_Entry (Struct) This struct is the record for scheduling events. Variables declared in this struct represent the state of the ready queue and tasks at the moment of the event. The variable that is crucial to this implementation is the sequence number, named seqNum.

RQE_Queue (Struct) This struct contains the buffer which stores the RQE_Entry records, control variables for logs (insert and delete index), and any other ready queue information that needs to be monitored.

rqe_discern_purpose_of_thread (function) This function determines if the task belongs to troubling applications (i.e kworker)

rqe_insert (function) This function builds the RQE_Entry and places it in the RQE_Queue. This function is called by enqueue and dequeue

A.2 Modifications to kernel/sched_debug.c

RQE_Q_CAP (Constant) This constant defines the size of the kernel buffer for IPSA.

SEQ_RQ (Seq proc file) This is the proc file IPSA writes to. The first read starts IPSA.

HARD_RESET (Proc file) This is the proc file where reading from invokes **rqRESET** function.

RQE_Get (Function) This function obtains the RQE_Entry from the RQE_Queue buffer.

rqRESET (function) This function resets all variables in the RQE_Queue and disables IPSA

Appendix B: Scheduling data exposed by IPSA

TABLE B.1: ENTRY LOG DECOMPOSITION

Data	Source	Notes
Sequence Number	Ready Queue	Unique Id assigned by IPSA
Application Name	Task	
Process Id	Task	
Timestamp of the Event	Ready Queue	
Execution Time	Task	
Total Context Switches	Task	
Type of Event	Task	
Priority of process	Task	
Total- Kworker events	Ready Queue	Thread counted rather than logged
Total - I/O events	Ready Queue	Thread counted rather than logged

Curriculum Vita

Edward G. Hudgins, raised in Alabama, served in the United States Army during Operation Enduring Freedom. After serving a four-year term, Edward was honorably discharged in 2010. Now calling Texas home, Edward enrolled at The University of Texas at El Paso. During his undergraduate studies, he joined the Robust Autonomic Systems Group research team; was awarded a REU stipend in 2012; and presented at the 2013 Constraints Programing Symposium. In 2014, Edward earned a Bachelor of Science in Computer Science from UTEP.
Shortly after, he began his Master of Science in Computer Science program at UTEP. Edward continued as a research assistant with Robust Autonomic Systems Group under the tutelage of Dr. Eric Freudenthal. Edward presented at the 15th Joint NMSU/UTEP Workshop on Mathematics, Computer Science and Computational Sciences in 2015. While pursuing his graduate degree, he also worked as the Information Technologies Student Work-study for the Computer Science department at UTEP. Edward was also a Public and Professional Programs Instructor for UTEP's P3 department. In 2016, Edward was hired with the United States Automobile Association (USAA) and is currently employed there as a Software Developer.

Contact Information: edwardghudgins@gmail.com

This thesis was typed by Edward Garwood Hudgins