© 2017 Ali Bohloolizamani

EFFICIENT MOBILE COMPUTING

BY

ALI BOHLOOLIZAMANI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Josep Torrellas

ABSTRACT

Smart handheld devices such as phones, tablets and watches are becoming more and more common rapidly. From a computer architect point of view, processor design for such computer systems is a complex problem. Since the Li-ion battery manufacturing technology is strictly limited by physical and technological limitations, the new generation of mobile processors should have a better energy efficiency to support an acceptable battery life. On the other hand, TLP of current mobile applications is measured to be mostly less than 2, which implies mobile processors should have high performance on user's demand to provide an acceptable QoE.

By shrinking the chip manufacturing technology size, SoC design has been the most preferred integration approach in such applications. For example, Apple's A10, iPhone7 SoC [1], has more than 3 billion transistors including 4 big.LITTLE cores, 6 GPU cores and caches. Due to power-density and heat-dissipation constraints of such integration level, providing high performance on demand in an efficient way is a complex control problem.

In the A10 architecture, assigning the right cores at the right time to running threads is a challenging complex control problem. The state of the art systems have control loops for controlling architectural parameters in different ways. In mobile devices controllers are heuristics-based mainly for simplicity. Considering the power-density and heat-dissipation issues in such systems, we propose an OS architecture and interface to provide an environment for improving the functionality of controllers in mobile computer systems.

To my family, for their love and support.

ACKNOWLEDGMENTS

Special thanks to Professor Josep Torrellas for such a professional relationship based on trust, who gave me the opportunity to make my dreams come true. Another special thanks to Raghavendra Pradyumna Pothukuchi for directing me in this project, his time and ideas.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

LIST OF ABBREVIATIONS ix

1 INTRODUCTION 1
  1.1 Mobile Applications Benchmarking 1
  1.2 Background on Controlled Systems 11

2 IMPLEMENTATION 15
  2.1 Problems with Linux Scheduler 15
  2.2 Implemented LKM 16
  2.3 Extensibility 19

3 EVALUATION 20
  3.1 Monitoring Baseline 20
  3.2 Baseline Improvements 21

4 CONCLUSION 27

REFERENCES 28

LIST OF TABLES

1.1 TLP values for different categories of commonly used mobile applications 6

LIST OF FIGURES

1.1 Three run-state migration methods 3
1.2 Example trace output demonstrating CPU context switches, processor 1 triggers a sched_switch event and the previous context was the swapper, processor 1 becomes active, there is 1 active CPU, processor 2 triggers the same kind of event and becomes active, now there are 2 active CPUs 6
1.3 Percentage of non-idle execution time different number of cores were running for the applications Google Chrome and Spotify 7
1.4 TLP over Time graph for the Spotify benchmark 8
1.5 TLP over time for aggressive scenario 9
1.6 TLP over time comparison - AnTuTu CPU test 10
1.7 Typical feedback control loop [2] 11
1.8 PID SISO controller for applications such as DVFS for delivering demanded QoS 12
1.9 An example of MIMO controller for architectural parameters in a computer system [2] 13

2.1 Implemented LKM for sensing scaled CPU times, and processes' state, and actuating on CPU masks for scheduling . . . 18

3.1 Monitoring baseline scheduler for 8 threads of Blackscholes from PARSEC 21
3.2 Monitoring baseline scheduler for 8 threads of Vips from PARSEC 22
3.3 Setting affinity of each thread to one core for 8 threads of Blackscholes from PARSEC 22
3.4 Setting affinity of each thread to one core for 8 threads of Vips from PARSEC 23
3.5 Setting affinity of each thread to two cores for 8 threads of Blackscholes from PARSEC 24
3.6 Setting affinity of each thread to two cores for 8 threads of Vips from PARSEC 24
3.7 Core-toggling every 0.25 second for 8 threads of Blackscholes from PARSEC 25

3.8 Core-toggling every 0.25 second for 8 threads of Vips from PARSEC 25
3.9 Speedup comparison among different implemented methods on Blackscholes and Vips from PARSEC, the best speeds are obtained by toggling each thread between big and LITTLE every 0.25s 26

LIST OF ABBREVIATIONS

CMP	Chip Multi-Processor
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
EAS	Energy Aware Scheduling
EE	Energy Efficient
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HP	High Performance
IO	Input Output
LKM	Linux Kernel Module
MIMO	Multiple Input Multiple Output
OS	Operating system
PID	Proportional Integral Derivative
QoE	Quality of Experience
QoS	Quality of Service
SISO	Single Input Single Output
SMT	Simultaneous Multithreading Technology
SoC	System on Chip
TLP	Thread-Level Parallelism
UI	User Interface

1 INTRODUCTION

In this chapter, an abstract work on mobile applications benchmarking is provided in detail, following by some background information on using formal control theory methods to control architectural parameters in a mobile computer system.

1.1 Mobile Applications Benchmarking

In this study, TLP over time is monitored for a big.LITTLE computer system running Android. Moreover, the quality of user experience on different core configurations is investigated for common Android applications.

1.1.1 Background

Traditional user-interactive desktop applications are mostly running on mobile devices nowadays. Tasks on mobiles are becoming increasingly more performance-intensive, and user's demand for multitasking is increasing. The mentioned trends drive the mobile industry towards more power-efficient processor designs. Due to the tight power budget on mobile devices, the benefit of using just DVFS is reaching a limit, and most vendors are turning to heterogeneous multi-core platforms. Most of the high-end smart phones in the market are dual-core, quad-core, or even octa-core devices.

Among the popular Android phones released in 2016, Samsung Galaxy S7, LG G5 and HTC 10 shipped with the quad-core Qualcomm Snapdragon 820 processor, Nexus 6P with an octa-core Snapdragon 810, and Samsung Galaxy Note 5 with an octa-core Exynos 7420 processor. However, some smartphones are equipped with only dual cores and still give satisfactory performance. Apple's A9 SoC [3] in iPhone 6s is dual-core, but is also one of the most powerful and energy-efficient mobile SoCs in the market today.

It seems there are two competing design decisions in the mobile processors manufacturing industry. Apple aims to improve per-thread performance, while Qualcomm favors introducing more cores. Qualcomm's deca-core Snapdragon 818 is shipping by the end of 2016.

Researchers have been studying core utilization of desktop and mobile devices for a while. Blake et al. [4] studied TLP on a suite of representative desktop applications in 2010. They showed that the number of cores that can be effectively used is less than 3 for the most commonly-used desktop applications. Later studies suggested that mobile device applications have a similar characteristic and cannot fully utilize a quad-core CPU.

Gao et al. [5] in 2015 analyzed mobile applications using TLP on an ARM big.LITTLE architecture and demonstrated that mobile applications utilize less than 2 cores on average, even with some background applications running concurrently. They observed a diminishing return on TLP with increasing number of cores and suggested that having many powerful cores is over-provisioning. They also claimed that current mobile workloads can benefit from an architecture that has the flexibility to accommodate both high performance and good energy-efficiency for different application execution phases.

Both the software and the architecture have changed since Gao's paper was written. Programmers are becoming more conscious of heterogeneous multicore [6] and try to write more efficient multi-threaded programs for such systems. The main reason is that most of the common mobile applications are web applications or cloud-based. They offload tasks on GPU, DSPs, and other ASICs in SoC. These units can already exploit much of the parallelism, leaving little for the CPU. On the device side, ARM big.LITTLE technology has evolved. The main idea is to have both energy efficient low performance cores(i.e., ARM A7) and high performance cores(i.e., ARM A15) in a same chip [7].

There are three scheduling methods for big.LITTLE designs. Global task scheduling is the most recent scheme among the three [8][9]. The other two are clustered switching and in-kernel switching. Figure 1.1 shows an example of two big cores and two little cores under the three models. In clustered switching, either two little cores or two big cores are running tasks at any given time. In in-kernel switching, one big core is paired with one little core and only one core from each pair is running a task at any given time. In

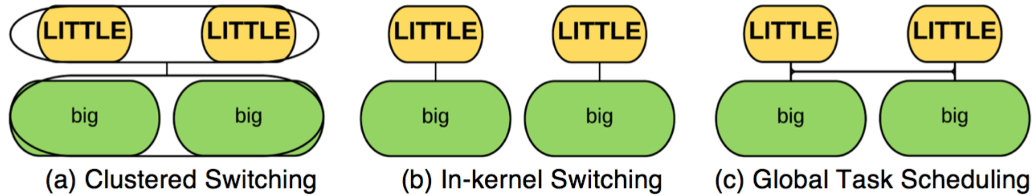

Figure 1.1: Three run-state migration methods

global task scheduling, any combination of cores can be running tasks at any given time. In Gao et al. [5], the authors conducted their experiments on an octal core Samsung Exynos 5410 SoC which uses clustered switching. Either four A15s or four A7s can be enabled at the same time. The system model is symmetric, since there is only one cluster running at any time. Although the paper was written in 2015, the device they used uses technology from 2011. Also, the architecture model from the OS point of view is symmetric, which reduces system model variation significantly.

Although previous studies of mobile device utilization suggest mobile applications utilize less than 2 cores most of the time, Android device vendors are heading towards more cores on chip hoping for a better QoE. With the global task scheduling schemes and new big.LITTLE devices, it will be interesting to revisit the question of the number of cores we actually need on the mobile processing unit. We study, the TLP of a suite of representative mobile applications on a recent octal core ARM big.LITTLE device that supports the latest version of the Android kernel's global task scheduler. The main motivation is to investigate whether the statements in [5] remain consistent with the most recent mobile technology.

1.1.2 Hardware Configuration

The Allwinner A80 Optimus development board is used for this study, it features an octal core Allwinner A80 SoC, including an ARM big.LITTLE octal core of four 1.6GHz A15s and four 1.2GHz A7s. Each core has 32KB/32KB L1 instruction and data caches. The A15s share a 2MB L2 cache and the A7s share a 512KB L2 cache. The board supports global task scheduling, with which any number of the eight cores can be running at the same time as long as power budget is satisfied. CPU0 is always running for OS tasks and

some systematic interrupts. The board is running Android 4.4.2-Kitkat interactive CPU governor mode, which is the most common choice for enabling DVFS for CPUs on mobile devices. Shell commands are used to force the kernel to adopt the specified core configurations as we see in the following, and the board is connected to a host machine, which is collecting dumped traces using the Android Studio platform [9].

1.1.3 Methodology

A wide range of popular Android applications are chosen as benchmark. This includes applications such as music and video players, social communication, games and web browsers. The focus is on three different core configurations. The default heterogeneous setup which has 4 big and 4 LITTLE cores, 2 big and 2 LITTLE cores, and 4 LITTLE cores. The thread level parallelism is used as metric to study Android applications. Each application is tested under standalone condition, being the only application running. There is also an aggressive use case which involves many background applications and a user who switches and interacts between different applications and Android GUI.

TLP

In the benchmarking experiment, I decided to use TLP as a metric for core utilization. TLP is a measure of core utilization, ignoring the fraction of the time spent with 0 cores running the application. TLP is used instead of a normal core utilization metric because many of the target applications are user driven, meaning they sit idle waiting for user interaction. Following equation is the formulation used for TLP in this study.

$$TLP = \frac{\sum_{i=1}^{n} C_i i}{1 - C_0}$$

C_i is the fraction of time spent where the system is using i cores and C_0 is the fraction of time spent idle for target application. The TLP metric is not a metric used to gauge performance, but is just a metric for system profiling. This metric provides a pretty accurate indicator of the number of

cores required to execute target workload with the target QoE. Basically it can be thought of as the average number of cores needed to run a workload over its non-idle portions of execution. The idea is to monitor different applications execution in terms of parallelism when enough computing power is provided.

Calculating TLP

The TLP for benchmarks is calculated from post processing stored dumped traces. In other words, a parser, parses and post processes the trace output file for each execution. The trace output, as seen in the example below in figure 1.2, traces events that occur within the system kernel. Each event has a PID for the process that triggered the event, a timestamp, the CPU that triggered the event, and the corresponding function being called. The main function that is considered is the sched_switch function.

By tracking who calls sched_switchs and when, the developed tool is able to track each CPU's context switches. If a core triggers a sched_switch and switches to the process swapper we know that core has become inactive. If a core moves from the swapper to execute some process, we know that the core has become active. By tracing the time cores become active, we can know how many cores are active at any given time. By knowing the number of cores active at any given time, calculating the TLP becomes trivial. We make sure to ignore any cores that are running the trace or our logging code to not over-estimate the TLP.

There are a few problems with the trace files when parsing. One of the main issues was dealing with dropped events. Occasionally the buffer that stores the trace output would overflow before flushing to the output file. So some events would be missed, including sched_switch events. When this happened we were be unable to know when a core becomes active or inactive. We accounted for this by tracking the last running process on a given CPU (by PID). If this process is seen executing instead on another processor and no more kernel events are being triggered for this process on the previous CPU, we know the process has migrated and the previous CPU has become inactive and the current one active.

```
# tracer: nop
#
# entries-in-buffer/entries-written: 164205/164205   #P:4
#
#                                  _-----=> irqs-off
#                                 / _----=> need-resched
#                                | / _---=> hardirq/softirq
#                                || / _--=> preempt-depth
#                                ||| /     delay
#           TASK-PID    TGID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |        |      |   ||||       |         |
          <idle>-0     (-----) [001] d..3  2630.829789: sched_switch: prev_comm=swapper/1
                                                                       prev_pid=0 ==>
                                                                       next_comm=m.spotify.music
                                                                       next_pid=29792
          <...>-29792 (-----)  [001] d..4  2630.830311: sched_wakeup: comm=Binder_3
                                                                       pid=6458
                                                                       success=1
          <idle>-0     (-----) [002] d..3  2630.869228: sched_switch: prev_comm=swapper/2
                                                                       prev_pid=0 ==>
                                                                       next_comm=hdmi proc
                                                                       next_pid=1502
```

Figure 1.2: Example trace output demonstrating CPU context switches, processor 1 triggers a sched_switch event and the previous context was the swapper, processor 1 becomes active, there is 1 active CPU, processor 2 triggers the same kind of event and becomes active, now there are 2 active CPUs

1.1.4 Result

TLP Measurements

As previously discussed, TLP measurements are conducted over a wide selection of common Android applications. Table 1.1 shows the results of this experiment when running the single application on the default big.LITTLE configuration with 4 big cores and 4 LITTLE cores.

Table 1.1: TLP values for different categories of commonly used mobile applications

Category	**Application**	**TLP**
System	None	1.02
Web Browser	Chrome	2.29
Video Player	Netflix	1.12
Music Player	Spotify	1.14
Communication	Telegram	1.12
Game	Angry Birds	1.15
Social Network	Facebook	1.42
Navigation	Google Maps	1.35
Email	Gmail	1.12
Daily Usage	**Average**	**1.26**

The first important thing to get from this TLP table is the system TLP.

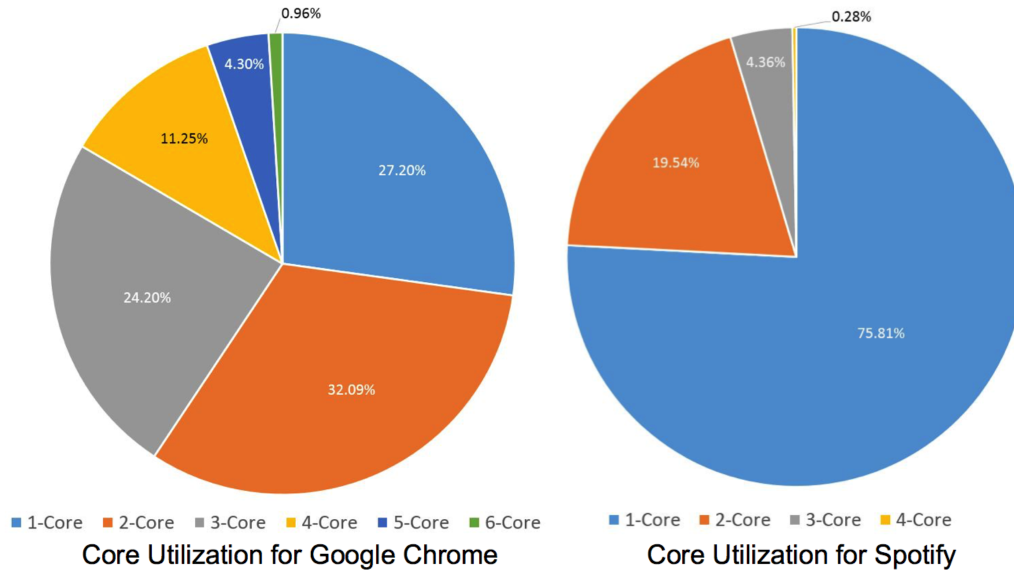

Figure 1.3: Percentage of non-idle execution time different number of cores were running for the applications Google Chrome and Spotify

This is the TLP of just running the Android system including UI deamons, frameworks and kernel itself (no applications). This TLP is very close to 1, which tells us that the Android operating system occupies 1 core. Moreover, the geometric mean of the application TLPs is 1.26. This tells us that popular Android applications don't fully utilize the cores simultaneously and possibly don't need to utilize a significant number of the cores. The only exception in our experiment is Google Chrome. In the Google Chrome experiments, the user opened several browser tabs, visited websites, and closed tabs in varying orders. Chrome is likely able to take advantage of the multicore system mainly because each tab could potentially run off in its own process and these processes can be scheduled independently on different cores depending on the workload required for the given tab. The following pie charts give a clearer description of what these TLP numbers mean.

There is shown that Google Chrome spent a significant portion of its execution time running on 2, 3, and 4 cores. This indicates that Google Chrome when used in the typical use case is able to take advantage of the multicore system and spread its workload across the multiple cores. However Spotify, spends most of its execution time running on a single core. This indicates that applications like Spotify only need 1 maybe 2 cores to run effectively (provide a good delay free user experience). Moreover, low TLP applications

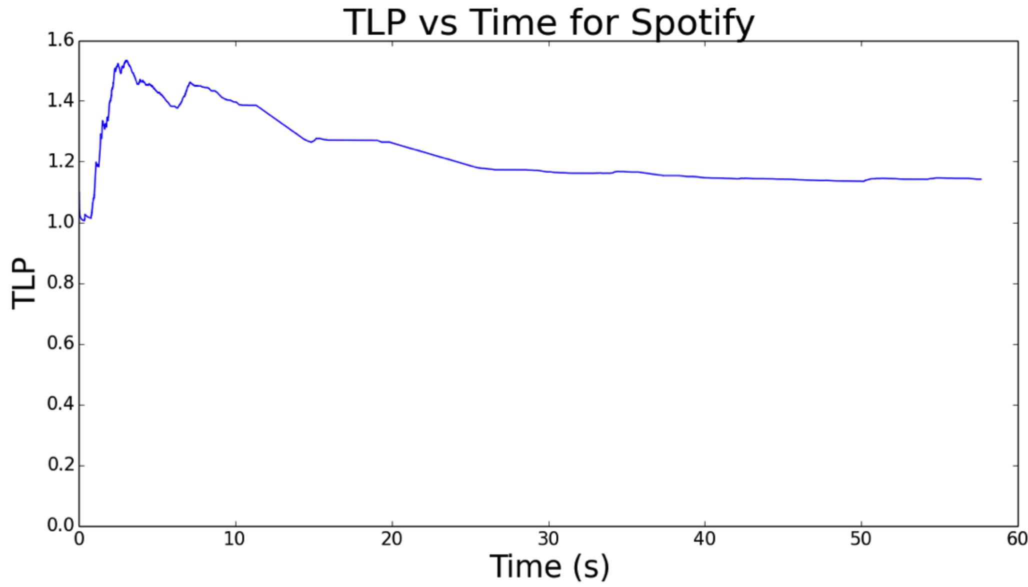

Figure 1.4: TLP over Time graph for the Spotify benchmark

could gain from interleaving and concurrency.

Android Apps TLP over time

A common application behavior detected is an immediate spike in TLP near the beginning of the application's lifetime. Figure 1.4 shows this general trend of TLP over time plot for application lifetime. The methodology for the test is to run the application by tapping the apps icon, later applying set of ordered UI commands to. Once the application was loaded we began taking the trace of the system followed by interacting with the application.

This trend shows that upon the first user interactions, the application's TLP significantly increases, indicating that the application must instantiate several threads or processes to handle the immediate burst of work related to launching, and the scheduler does not know how to effectively schedule this workload across all the cores. So it simply assigns each thread to a different core to maximize QoE. However, over time TLP begins to decrease and slowly move back towards a value of 1. We believe this happens because the Android global scheduler is able to identify the workload pattern and properly schedule the application's processes on proper core's.

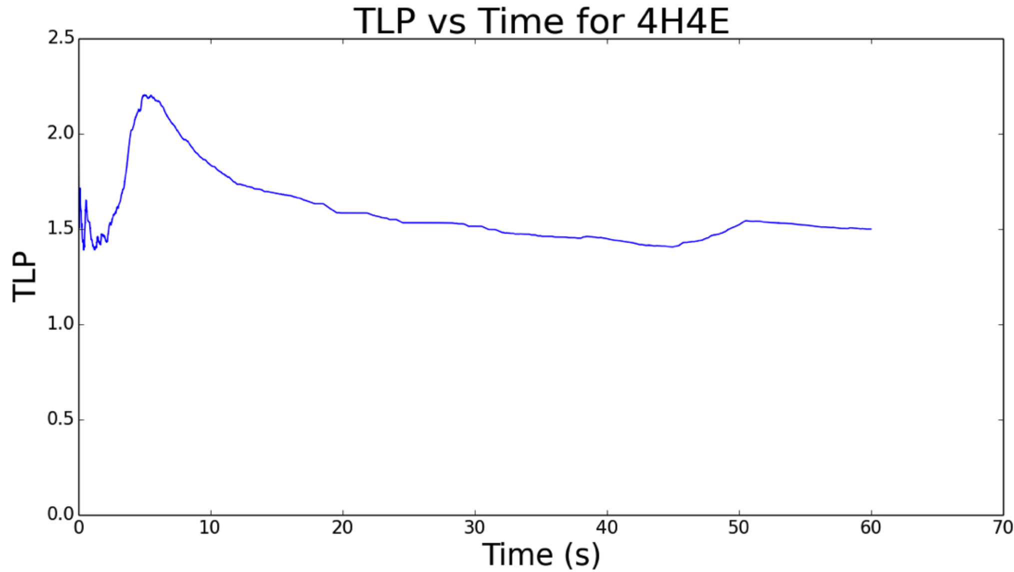

Figure 1.5: TLP over time for aggressive scenario

Aggressive scenario

In this test a more realistic general scenario, a user running multiple apps in the background (such as navigation, a music player, Facebook, etc...) while switching the top task and interacting with it is investigated. Figure 1.5 shows the TLP over time. The test has been done for 3 different architecture configurations (4HP4EE,2HP2EE,4EE) yet the result is about the same. The main idea behind having different sets of core configuration is to verify other system components i.e. core power gating is not interfering in our TLP observations. Note that the relatively low steady state TLP even for high use case scenarios running on fixed set of cores.

As mentioned there is no significant difference between TLP over time in the three configurations which indicates the low amount of parallelism exploited in these workloads. Considering the fact that most mobile devices are run in similar scenarios, it seems fair to argue that devices should be optimized a in way to fit for these workload scenarios.

Antutu Benchmark

On the other hand, I was curious to conduct the same test methodology for a potentially highly parallel application. AnTuTu [10], is developed mainly

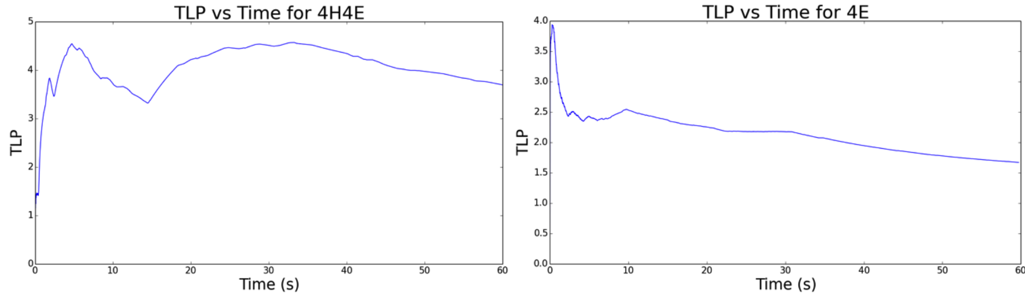

Figure 1.6: TLP over time comparison - AnTuTu CPU test

for performance measurements of ARM based mobile devices. It is a heavy non-realistic workload for mobile devices and it does not need any UI. This application is download-able from Google Play on Android devices, and there are several tests which monitor detailed information related to the CPU, GPU, network, and memory system usage for the application. Here are the results from running the CPU test on our different configurations.

As shown in figure 1.6, as we ran the CPU test on our cores' configurations there are noticeable differences in TLP over times. These differences show that providing less computing power (core counts) for the scheduler could degrade the performance of highly high parallel scenarios significantly, as we expected.

1.1.5 Illustration

In this short study, we showed that there is no need for having more cores when even the aggressive usage of up-to-date applications cannot effectively utilize 3 cores. Global Task Scheduling helps the software exploit more from big.LITTLE architectures. Moreover it is shown that reducing the core counts would degrade the performance of highly parallel program chunks.

To conclude, trends show that, if the behavior of the mobile applications is going to be the same in the future, an acceptable single thread performance would satisfy the user in terms of QoE. Later, having a few cores plus accelerators and coprocessors would be a better design decision for handheld mobile devices.

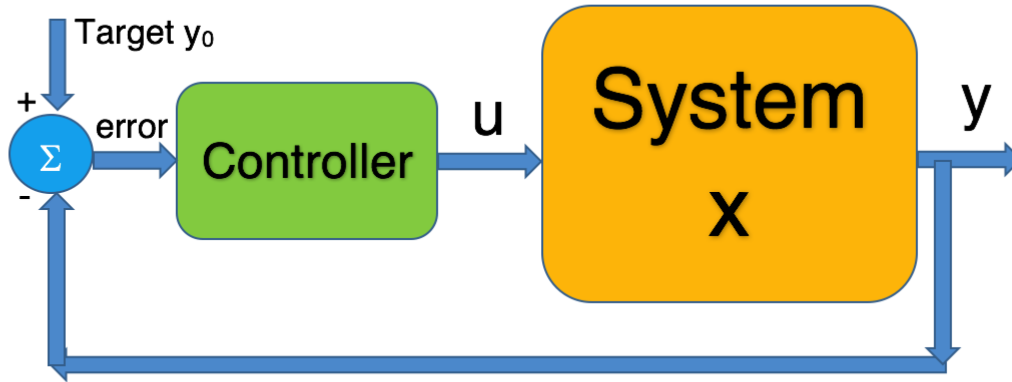

Figure 1.7: Typical feedback control loop [2]

1.2 Background on Controlled Systems

A typical closed-loop controller, also known as a feedback controller, has a high level schematic shown in the sub-figure 1.7. Assume the system block includes all the parts of the computer system and there are sets of input and output values for the system. For system x in the figure representing the computer system, input set is u, and y is the output. The ultimate goal of the controller is to actuate on the input set to make the system follow the given reference value(s) for the output. In other words, the current output is fed-back and the error from the reference is the controller input and the controller determines the u values based on the current error and history of the controller.

There are different aspects and trade-offs in controller design. What type of formal design would work best for the target system? How fast does a controller reach the reference value? How much over/under shoots the system could tolerate? How resilience is the design to different artifacts such as sensor errors in sensing feedback signals or variation in the system model? Mentioned considerations are all questions which controller designer has to answer in a smart way to efficiently operate for target application. Obviously answers to, the mentioned design decisions should correlate, otherwise systems efficiency degrades significantly. For example, the faster a controller reaches the target values, the more over/under shoots it may apply to the system inputs, and the less system model variation, Lets the controller designer

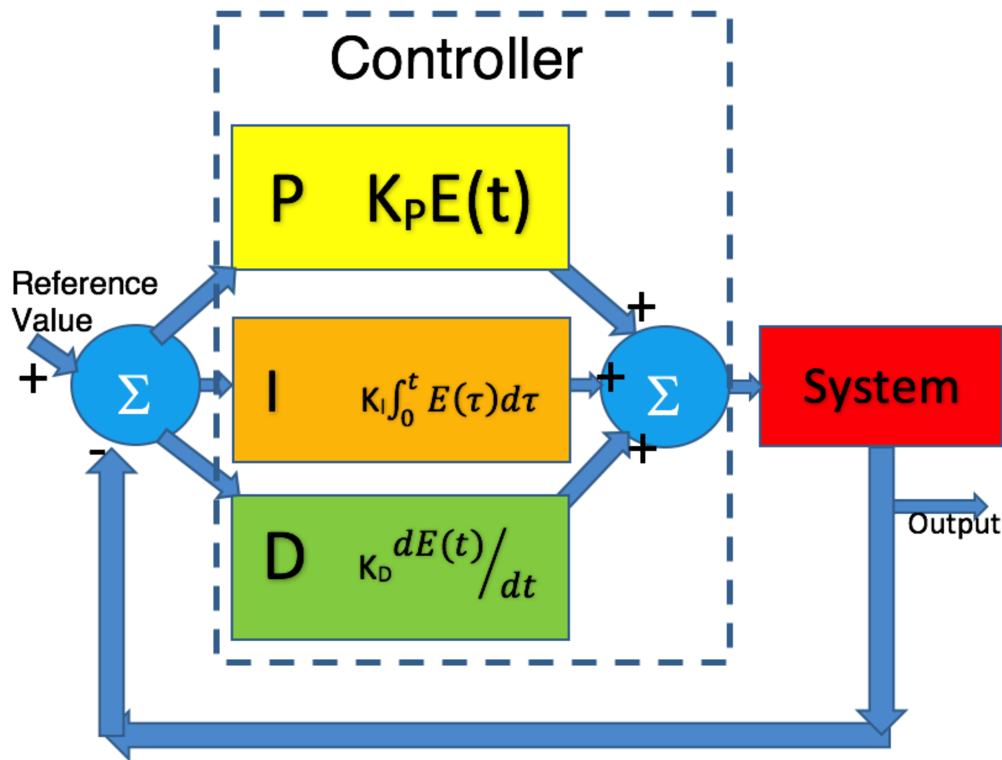

Figure 1.8: PID SISO controller for applications such as DVFS for delivering demanded QoS

to come with a better controller.

For example, one of the common controllers, which has been widely used in computer architectures, is PID controller. As shown in figure 1.8, it simply combines three different informations on the error signal to actuate on the input of the system. The output could be one of the performance counters inside the processor(lets say the instruction count). The reference value is passed by application to the kernel for QoS. Finally, the input could be different voltage steps for DVFS.

There are different architectural parameters tightly collaborating with each other. If the system has a different decoupled SISO controller for each different parameter, it is possible that the controllers work against each other and the whole system functionality degrades as a result. However, the biggest advantage of SISO controller is design simplicity along with great effectiveness. The controller designer only needs to find a proper coefficient for each

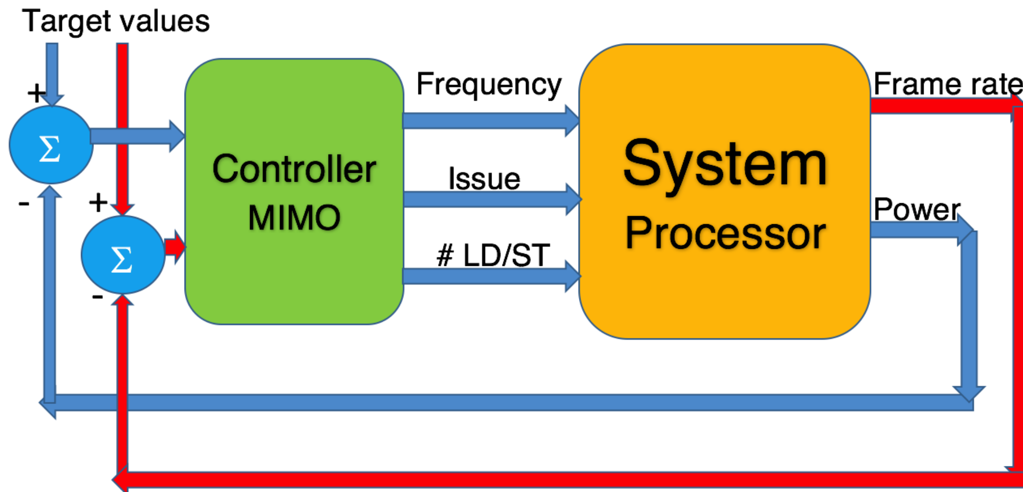

Figure 1.9: An example of MIMO controller for architectural parameters in a computer system [2]

term in the PID controller.

An alternative approach would be designing a MIMO controller as shown in figure 1.9. The controller tries to keep the screen frame rate at the satisfactory level, lets say 60fps, and also meet the power envelop. In other words, the system tries to provide target QoE for user. The MIMO controller actuates on frequency, issue width and load/store buffer length.

The effectiveness of control theory over other methodologies for computer architecture tuning is explained in detail in the work of Pothukuchi et al. [2]. In table 1, they summarized the comparison of different techniques for architectural tuning. Since in this work, I mainly tackled the implementation issues of using controllers in practice on a big.LITTLE Odroid developement board, it's recommended to refer to their work for more details on architectural controllers. There is also a detailed tutorial on architectural controller formal design flow published by the same authors [11].

Considering the study at the beginning of this report, and also recent big.LITTLE ARM architectures for mobile processors, the problem of designing an efficient controller for mobile processors is getting even more challenging. Every new version of mobile processors should speed up the previous version, mainly because mobile applications are getting more and more complex and users are demanding more concurrency and better QoE.

Moreover, such processor should have less overall power consumption for a specific code chunk, in other words, it needs to process more energy-efficiently to provide a longer battery life. Consequently, based on the fact that average TLP is less than 2, there are both a major need and potentially huge benefit on efficiently controlling of mobile processors.

2 IMPLEMENTATION

Researchers have worked on custom ad-hock designs of the kernel scheduler to get a better energy efficiency. In [12] they are addressing the same performance and power efficiency problem but for a different SMT CMP platform. There are performance issues with the Linux scheduler even for a general purpose processor [13]. After implementing an architectural controller on a real big.LITTLE system, those problems with the baseline system manifest as performance degradation. As a result, a smarter scheduler is needed. Therefore, I implemented an interface using LKM, which lets the programmer write any scheduling policy in user space without changing any kernel code. This saves huge time in development.

2.1 Problems with Linux Scheduler

Exploiting the big.LITTLE architecture flexibility for power efficient computing has been a challenging problem. Mainly, because there are several architectural parameters controlled by different units. Those units usually work independently from each other. Assume an SoC for a mobile device with a big.LITTLE octal core CPU architecture. Those cores should overall consume less than the maximum power budget, which often implies that some of them need to be in a lower power level or power-gated, to meet the power envelop constraint. Temperature is also a critical factor for safety. There is a thermal management unit implemented in firmware, to kick in when the temperature of a core hits a certain point. The thermal management unit then takes care of putting the heated core to a lower power state. There is no customization for applications. In other words, we do not have any information about the workload in run-time on our asymmetric machine.

Implementing an effective scheduler to work in such complicated environment is an open challenging problem in OS scheduling. Mainly, because the above mentioned artifacts related to power and temperature manage-

ment units, could easily act against the scheduler decisions. As a result, performance and also energy efficiency both are sub-optimal. There is an interesting work in this context. However, the authors bounded the application to web browsing [14]. They addressed the same problem only for web browsing by benchmarking and tuning the scheduler for that pattern. Moreover, there is another work on a software approach for distributing tasks on a big.LIITLE architecture for multi-thread applications [15].

There is no work on trying to find a smart controller to tackle this problem for general purpose applications. To be more specific, there is no work on architectural controllers that perform very well for all mobile applications. There are some proposals on EAS(Energy Aware Scheduling) to converge to a better resource management leading to a better energy efficiency [16]. Unfortunately the current-updated implementation of the Linux scheduler works poorly on recent big.LITTLE development boards e.g. Odroid-XU4. I ran multi-programmed SPEC2000 and multi-threaded PARSEC[17] applications on my Odroid-XU4 board while logging temperature and performance values. The overall system performance degrades significantly when the thermal management unit kicks in for high temperatures. I decided to tackle the problems of the scheduler with a hierarchical controller consisting of different blocks implemented with formal control-theory methods, user and kernel space tasks and many interfaces between different software layers.

2.2 Implemented LKM

My formal controller is implemented in user space. On the other hand, the scheduler runs in the kernel space. There are a set of OS metrics fed into the controller's block: some information about processes, cores, temperature and power. To provide controllers running in user space with such information, there are some challenges. Some information such as processes' states and CPU time change rapidly. The main challenge is to have impact on scheduler decisions by monitoring critical values in the kernel from the user space. In our design, there are values such as application processes' state and scaled CPU-time, or the scaled time each process spent running on a core. Acquiring such values frequently from user space has a large overhead for system performance. Since we needed to implement it in an efficient way to

be able to do better than the baseline performance. I implemented an interface based on a Linux Kernel Module that interfaces with both the sensors and the actuators of our controller.

Linux Kernel Modules are section of C code that can be installed and removed into the kernel at run time. They extend the functionality of the kernel without the need to reboot the system or changing the kernel code. A type of modules which is really common is the device driver module. It allows the kernel to access hardware connected to the IO, which is installable on demand. Without LKM, we would have to rebuild the kernel and add new functionality directly into the kernel source code. Besides having larger kernels, it would be such a waste of time to rebuild and restore the kernel after a single change in implementation. Also, kernel program debugging would be hard since any error would crash the system and any change would need a rebuild. On the other hand, LKM has the advantage of fast installment and low cost of recovery. For example, in our system a single restart would recover the system and remove the faulty module.

After installing the LKM into the kernel, a virtual file named"status" is generated in the Proc file system. Every access to the status file from the controller invokes the corresponding call back function. The controller, at the beginning, registers application processes to the module by writing their process-ID into the status file. After registration, updated OS metrics of registered IDs will be ready upon the controller's request by reading the status file. The implemented LKM sets up a timer interrupt and a handler to hook up monitoring and actuating tasks to the work queue, which is served by on kernel threads.

As shown in figure 2.1, there is a local linked list for storing updated OS metrics inside the module implementation. The implemented linked list makes it possible to respond to the controller, updated version of values with out locking the main data structure in the kernel. The assigned work queue task grabs the lock of task_struct, one of Linux data structures holding interesting metrics which is needed in controller, traverses the list and finally updates OS metrics in internal data structures, and also applies the output values of the controller to task_struct. The current implementation monitors the scaled CPU time, processes state, and actuates on CPU masks for scheduling of each task. In other words, the controller can decide where to run each thread next time.

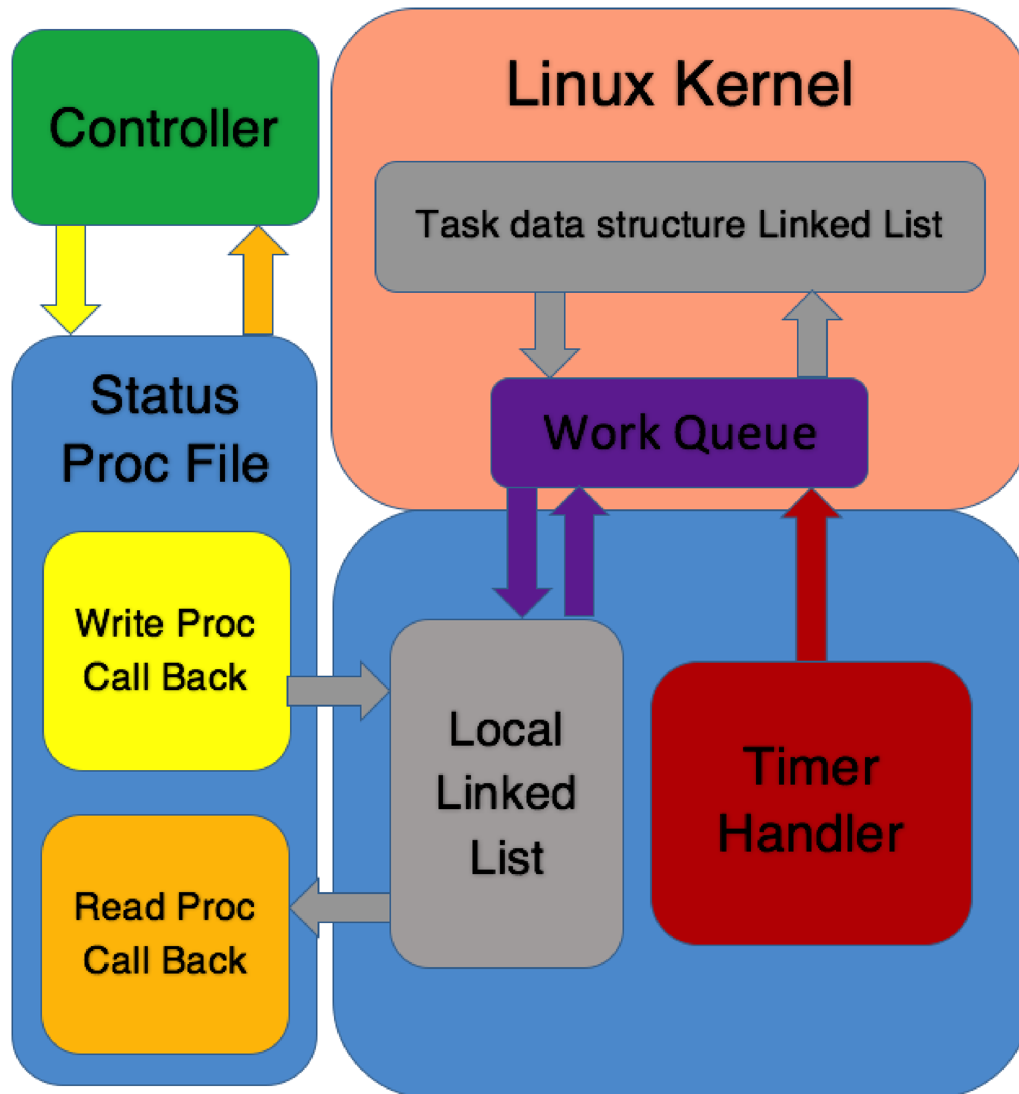

Figure 2.1: Implemented LKM for sensing scaled CPU times, and processes' state, and actuating on CPU masks for scheduling

2.3 Extensibility

In the current implementation, processes' information, core utilization and cache locality metrics are obtained from a centralized LKM and Perf tool using performance counters. In future studies, we might need to measure run time power consumption, yet in most of the development boards there is no such sensors or the latency to access them is pretty high(i.e., 500ms for Odroid-XU3). I designed and implemented a power view tool, a smart power sampler, which can log power(maximum 50w) every 10ms. It can also fed it back to the controller itself, either through fast serial communication or a typical network. Technically, the power view box is programmed on a Raspberry Pie B+ development board, a current sensor, and a GrovePi development extension board adding analog-to-digital converters to the system.

3 EVALUATION

To evaluate the effectiveness of our platform, we implemented different versions of controlling methods, all in kernel space using the kernel module, to improve multi-threaded application performance. The main goal of the controller is to help the Linux scheduler schedule threads on the big.LITTLE architecture fairly.

3.1 Monitoring Baseline

We used our platform for monitoring the baseline scheduler, while running 8 threads of Blackscholes, one of the applications of the PARSEC benchmark suit. Blackscholes is a CPU intensive workload without synchronization. We are not interfering in scheduler decisions. The monitoring result is shown in the figure 3.1. The Y axis is the core number: 0 to 3 are LITTLE and 4 to 7 are big cores. The X axis is simply time, each solid colored line in the graph corresponds to a running thread. The vertical dashed lines represent the end of a thread. Note that the monitoring is conducted only for the parallel section, which is the main focus of this chapter.

Frequency-scaled CPU times are also provided in the table in figure 3.1. The ideal scenario for a multi-threaded program would be one in which all threads are terminating at about the same time. In other words, they may have similar CPU times at any given time. In figure 3.1, PID 0 was always running on a big core. Consequently it finished first among all other threads. On the other hand PID 3 was mostly running on a LITTLE core. As a result it finished as the last thread. Since Blackscholes lacks synchronization, it can reveal the issues with the Linux scheduler on a heterogeneous architecture clearly.

In contrast of Blackscholes, the Vips application from PARSEC suite has fine-grained synchronization. Figure 3.2 shows the same monitoring experiment on the Vips application. Note that in figure 3.2 all threads terminated

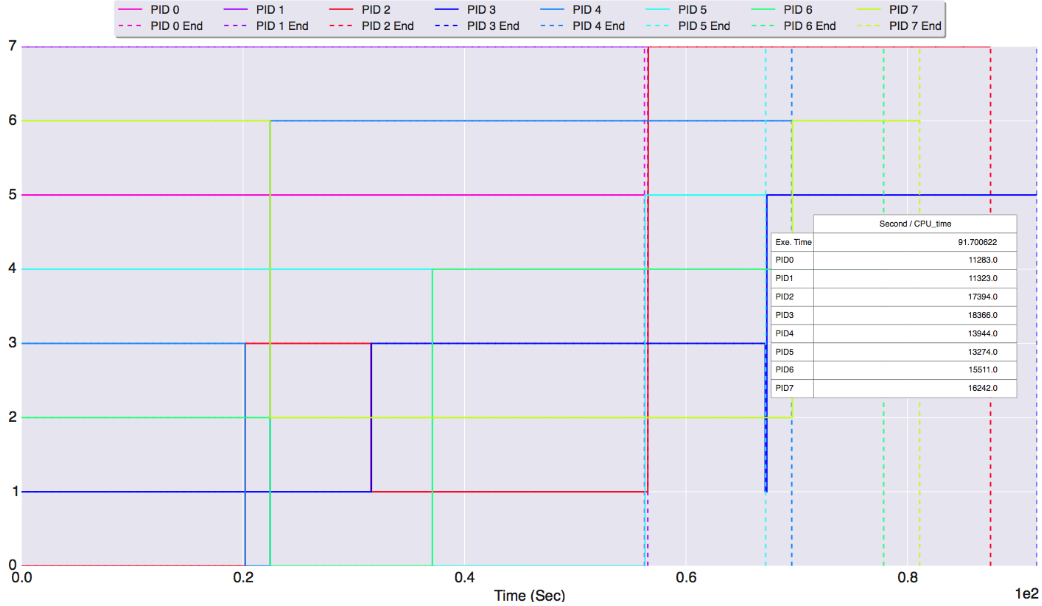

Figure 3.1: Monitoring baseline scheduler for 8 threads of Blackscholes from PARSEC

at about the same time, mainly because of the fine-grained synchronization, which causes threads to change cores all the time. After this monitoring observation, our main focus was on designing a universal controller for the scheduler to achieve a better performance, using our platform. Note that, the measured performance overhead of our tool on the system for monitoring is about 3%.

3.2 Baseline Improvements

The next set of experiments consisted of forcing the Linux scheduler to use affinity scheduling for threads, using our platform. Figures 3.3 and 3.4 show the timing result when there is only one core assigned to each thread. As we expected, again, the Blackscholes suffers from non uniform thread termination due to lack of synchronization. On the other hand, for Vips the affinity scheme does not hurt the simultaneous termination. It improves the performance. I believe the performance gain here comes from eliminating unnecessary migrations.

The next set of experiments were on setting affinity of each thread to two cores, one big and one little, using my platform. Figures 3.5 and 3.6

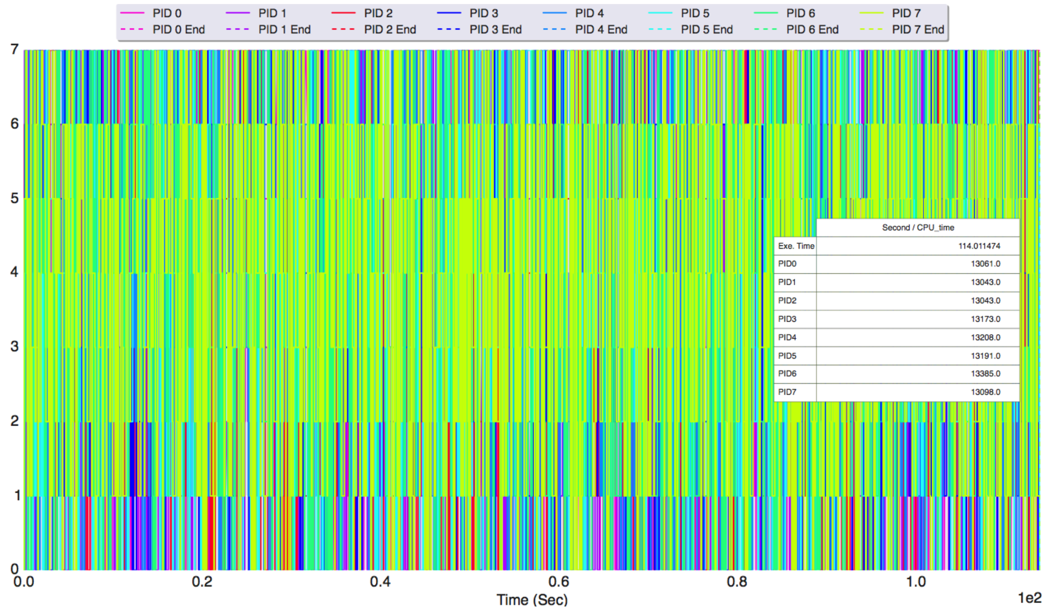

Figure 3.2: Monitoring baseline scheduler for 8 threads of Vips from PARSEC

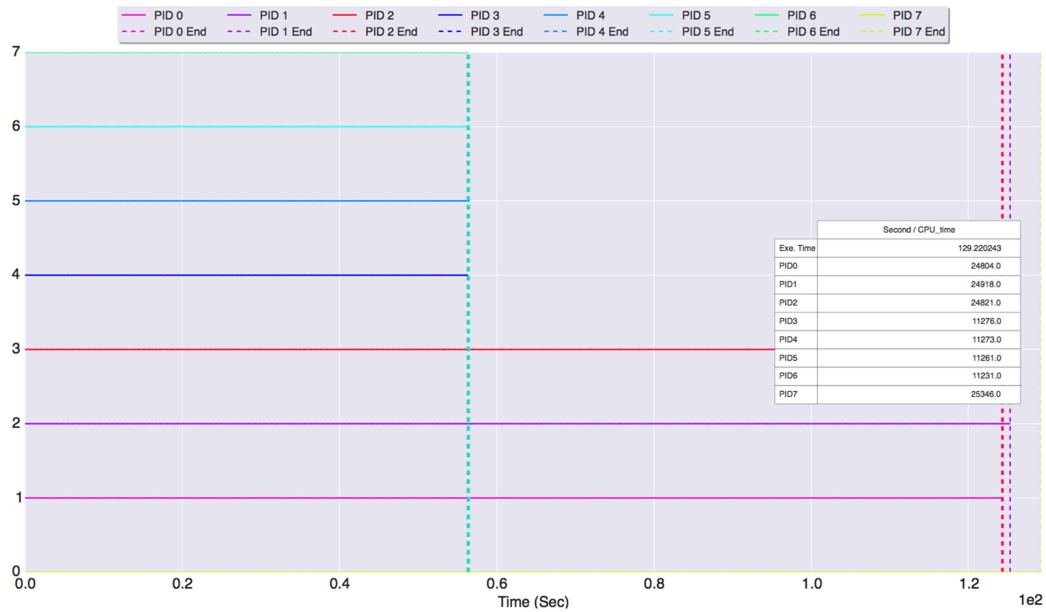

Figure 3.3: Setting affinity of each thread to one core for 8 threads of Blackscholes from PARSEC

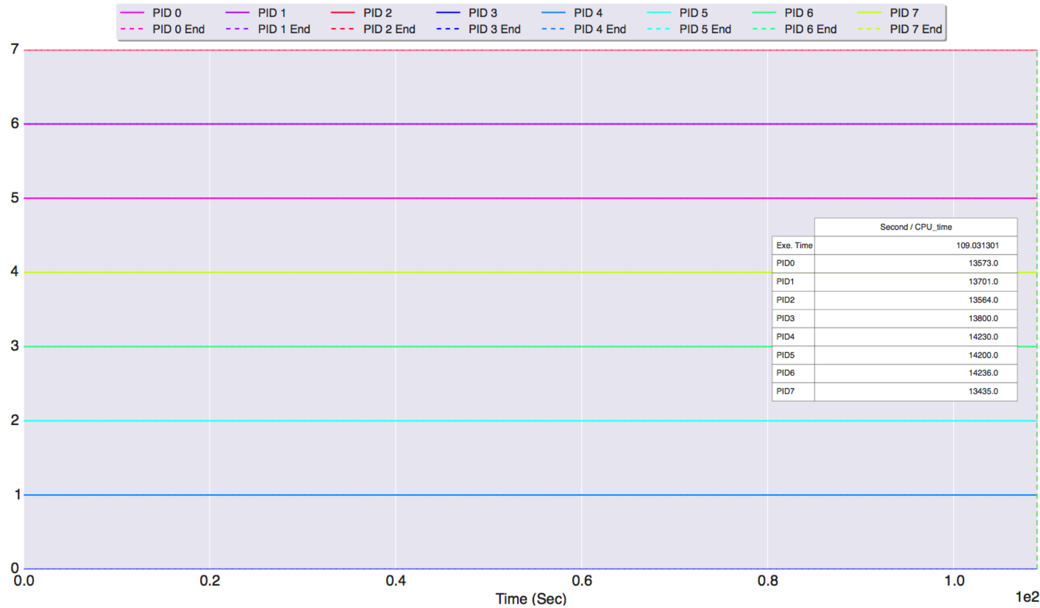

Figure 3.4: Setting affinity of each thread to one core for 8 threads of Vips from PARSEC

show the result. There is an interesting artifact observed in the figure 3.5 about the Linux scheduler. As long as a task is utilizing the resources well, like Blackscholes, the Linux scheduler refuses to migrate the task. This observation was an eye-opener. It helped me understand how to achieve the next scheduling scheme which works best among our tested approaches.

The last and most effective scheduling scheme implemented is periodic toggling between LITTLE and big for each thread. Obviously, there is a sweet point for how often it is best to toggle cores. The result is shown for toggling every 0.25 second in figures 3.7 and 3.8 for Blackscholes and Vips respectively. The result is that we obtain the faster executions of Blackscholes and Vips.

We did the same experiment on core-toggling for 3 different rates. Figure 3.9 wraps up the performance gain of different methods. The baseline is just monitoring. As it is shown, using our platform and the simple core-toggling scheme for the controller, we speed up the multi-threaded applications up to 14%. The proposed platform implemented as a Linux Kernel Module could be utilized in smarter ways using control theory and machine learning methods.

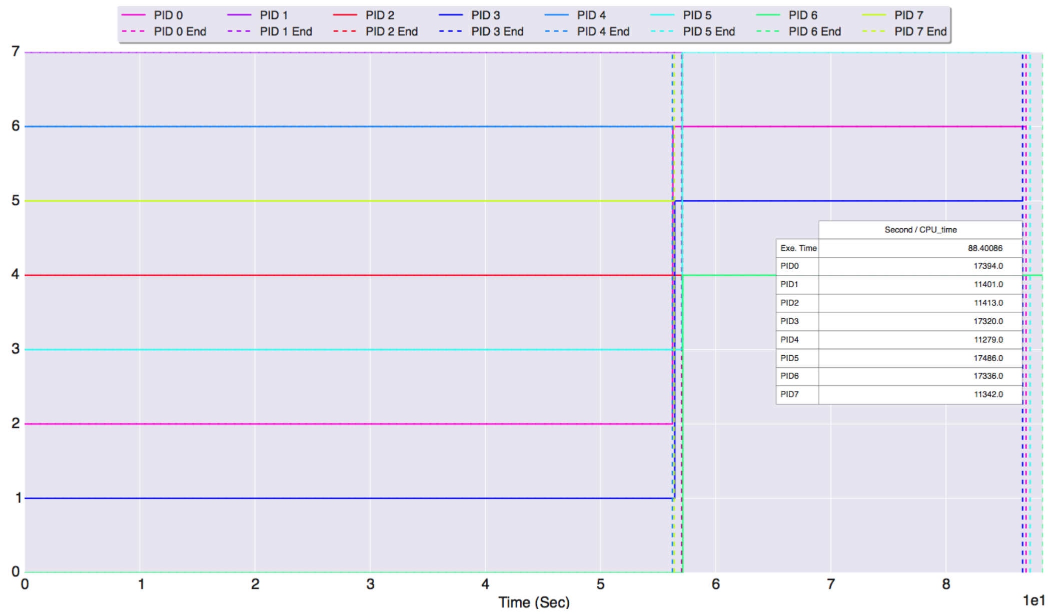

Figure 3.5: Setting affinity of each thread to two cores for 8 threads of Blackscholes from PARSEC

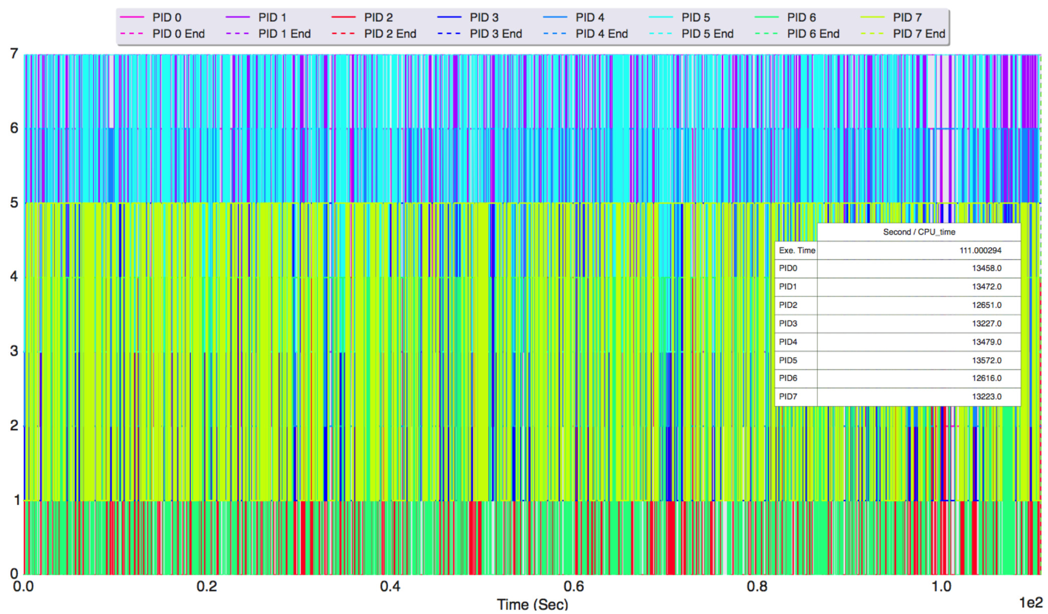

Figure 3.6: Setting affinity of each thread to two cores for 8 threads of Vips from PARSEC

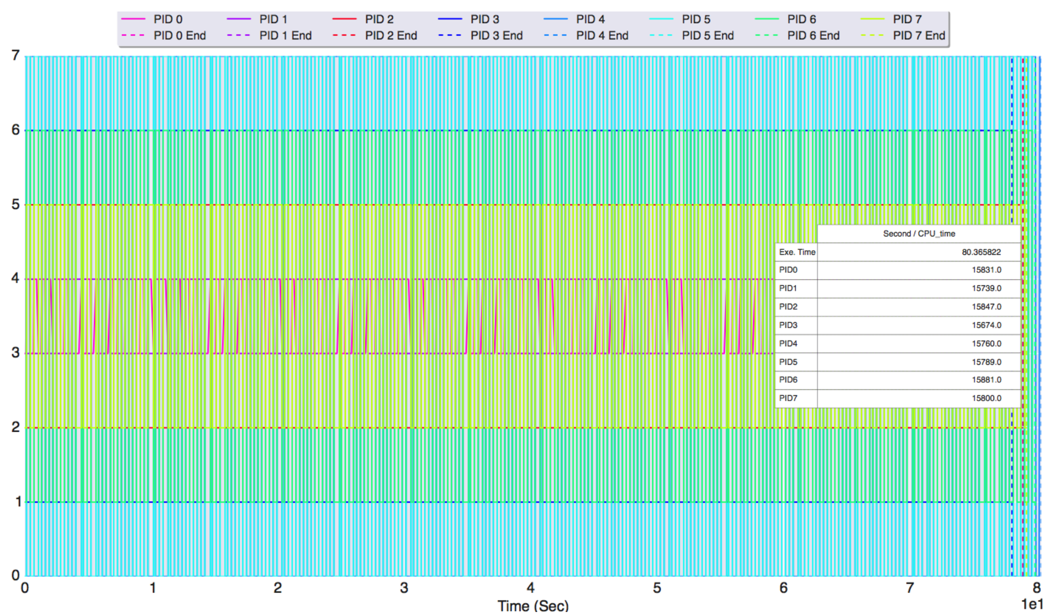

Figure 3.7: Core-toggling every 0.25 second for 8 threads of Blackscholes from PARSEC

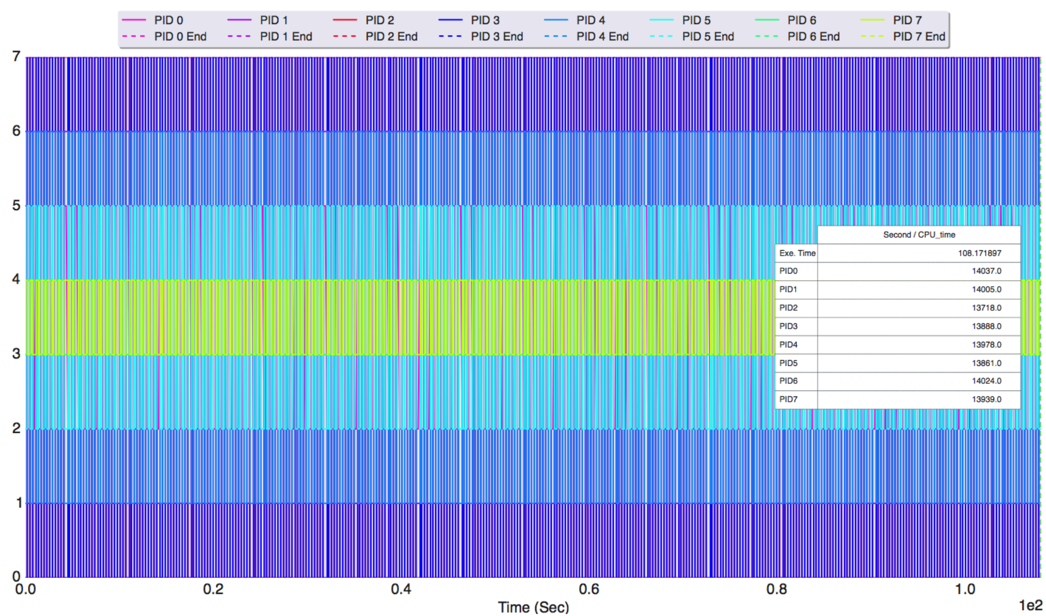

Figure 3.8: Core-toggling every 0.25 second for 8 threads of Vips from PARSEC

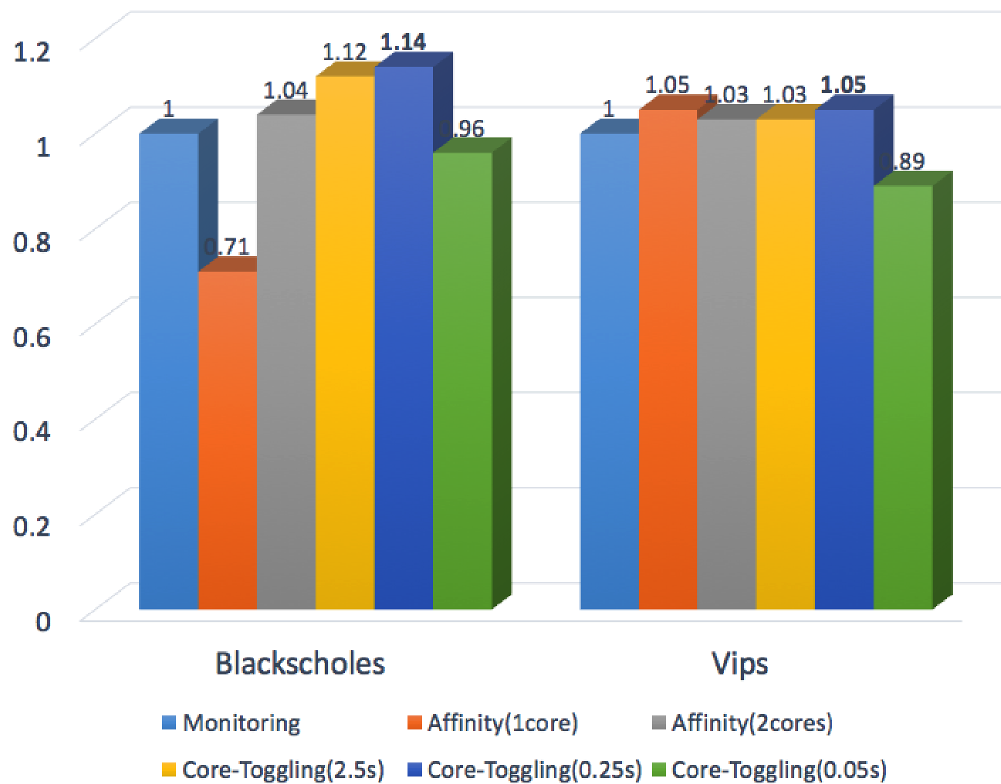

Figure 3.9: Speedup comparison among different implemented methods on Blackscholes and Vips from PARSEC, the best speeds are obtained by toggling each thread between big and LITTLE every 0.25s

4 CONCLUSION

The big.LITTLE architecture is the state of the art design strategy for mobile processors, yet there is an open question on how to best exploit this flexibility in hardware for mobile applications. A benchmarking study on TLP of Android applications shows an average TLP of 1.6 for all mobile applications. This low number is mainly caused by waiting for IO or UI, since most of the mobile application are deeply interactive with web and user. On the other hand, QoE is the ultimate goal of these systems. Single thread performance on demand, is essential for having a high QoE. All mentioned facts are contributing to the complexity of this challenging control problem.

Scheduling is the key part, since power and thermal management units are not flexible, and some vendors make power and thermal management units run as firmware for safety. They are hard to modify. Our idea is to control the scheduling in a way that cooperates with other controllers in the system, to achieve more efficient computing. An implemented LKM is proposed for offloading regular critical controlling tasks from user space to kernel space implemented by working queue threads. This helps the developer to test the controller without being concerned about Linux kernel scheduler interfaces. We studied several techniques to schedule multi-threaded application on a big.LITTLE system. We found that the best speedups are obtained by toggling each thread between big and LITTLE periodically, every 0.25s.

REFERENCES

[1] "The A10 website," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Apple_A10

[2] R. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 658–670, 2016.

[3] "The A9 website," 2015. [Online]. Available: https://en.wikipedia.org/wiki/Apple_A9

[4] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications." *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 302–313, June 2010.

[5] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C. J. Wu, "A study of mobile device utilization," *Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium.*, pp. 225–234, Mar. 2015.

[6] S. Patil, Y. Kim, K. Korgaonkar, I. Awwal, and T. Rosing, "Characterization of users behavior variations for design of replayable mobile workloads," *International Conference on Mobile Computing, Applications, and Services. Springer International Publishing.*, pp. 51–70, Nov. 2015.

[7] "Arm white paper on big.little technology: The future of mobile," 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[8] "Arm white paper on big.little technology moves towards fully heterogeneous global task scheduling," 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf

[9] "The Android Studio website." [Online]. Available: http://developer.android.com/tools/studio/index.html

[10] "The AnTuTu chip performance benchmark website," 2016. [Online]. Available: http://www.antutu.com/en/index.shtml

[11] R. Pothukuchi and J. Torrellas, "A guide to design mimo controllers for architectures," 2016. [Online]. Available: http://iacoma.cs.uiuc.edu/iacoma-papers/mimoTR.pdf

[12] A. Vega, A. Buyuktosunoglu, and P. Bose, "Smt-centric power-aware thread placement in chip multiprocessors," *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques. IEEE.*, pp. 167–176, Sep. 2013.

[13] J. P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quma, and A. Fedorova, "The linux scheduler: a decade of wasted cores," *Proceedings of the Eleventh European Conference on Computer Systems. ACM.*, p. 1, Apr. 2016.

[14] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," *High Performance Computer Architecture (HPCA), IEEE 19th International Symposium.*, pp. 13–24, Feb. 2013.

[15] K. Chronaki, M. Moret, M. Casas, A. Rico, R. M. Badia, E. Ayguad, J. Labarta, and M. Valero., "Poster: Exploiting asymmetric multi-core processors with flexible system sofware," *Proceedings of the International Conference on Parallel Architectures and Compilation. ACM.*, pp. 415–417, Sep. 2016.

[16] E. D. Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini, "Workload-aware power optimization strategy for asymmetric multiprocessors," *Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE.*, pp. 531–534, Mar. 2016.

[17] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.