A SOFTWARE FRAMEWORK FOR APPLICATION-GUIDED POWER-AWARE CONTROL SYSTEMS

A Thesis
Presented to
The Academic Faculty

By

Michael Joseph Giardino

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2019

A SOFTWARE FRAMEWORK FOR APPLICATION-GUIDED POWER-AWARE CONTROL SYSTEMS

Approved by:

Dr. Bonnie Ferri, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Aldo Ferri
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: January 9, 2019

Not everything that is faced can be changed,

but nothing can be changed until it is faced.

James Baldwin

If you dare to struggle, you dare to win.

If you dare not to struggle then damn it, you don't deserve to win.

Fred Hampton

For my father,

without whom I would not be here and without him, I am here.

ACKNOWLEDGEMENTS

This dissertation is the culmination of many years of work that were made possible by many people. First, I would like to express the utmost gratitude for my advisor Dr. Bonnie Ferri. From the very beginning she guided me through the inevitable ups and downs of the process. Her encouragement, insight, and motivation made all of this possible. Many thanks also to my committee, Drs. Abhijit Chatterjee, Aldo Ferri, and Sudhakar Yalamanchili, for their advice making this work much more complete. I would also like to thank the Enterprise and Big Data group at Intel Chandler, especially Dr. Kshitij Doshi. Dr. Dimitrios Charalampidis at the University of New Orleans first introduced me to engineering research and helped set me down this path.

My thanks also go to the Opportunity Research Scholars who helped throughout this research. I must give an extra acknowledgement to Eric Klawitter who put up with me for two whole years and did an enormous amount of work getting our robot up and running. Wayne Maxwell stayed up many a late night trying to help collect data and explain basic mechanical engineering concepts to me. Thanks to Kevin Pham and James Steinberg for all the parts and laughs I needed for the past years. The ECE academic staff at Georgia Tech are top notch and have helped me out of numerous binds. Extra thanks are due to Daniela Staiculescu, Tasha Torrence, and Chris Malbrue.

My time at Georgia Tech wouldn't have been as rich both personally and intellectually without a cadre of friends that gave me much more than I ever could repay, especially Drs. Hommood Alrowais, Sarp Satir, David Torello, Nortey Yaboeh, and Yusuf Yaras for coffee and killing time, Cypress and campfires; Becky Katz, Ashley King, and Elise Blasingame for their support, perspective, and understanding; There aren't words for my brother Barry Muldrey without whom, it is no exaggeration, I would not be writing these acknowledgements. Thanks must also go to my long-time friends who always had time for me especially Ryan Poirrier, Jordan Blaize, Neil Norton, and Martin Maxwell.

I am the person I am today thanks in large part to my wonderful and supportive family. Michael Comardelle was the archetype of character and curiosity, a little of both I like to think rubbed off on me. I was incredibly fortunate that Brenda Comardelle was so generous with his time when I needed it the most. Thank you also to the Coziers for dinners, laughs, and adopting me as a second son. Much love and thanks to my families: The Seltzers, Stablers, Ackers, and Holladays.

My grandfather, Joseph Giardino, imparted a positive vision of the world on me from a very young age. He has always been very kind to me even though I abandoned his beloved humanities and went into the trades. My brother Luca and sister Annie are a constant inspiration both personally and professionally. My second mom, Lyn, can always be counted on for love, compassion, and advice. My mother has been, from my earliest memory, my tireless advocate and font of love and understanding. My father was a man to whom I never stopped admiring, whose advice and encouragement made me seek the most from every facet of life. I wish you were here for this, Pop. Finally, my deepest gratitude to my wonderfully kind, supportive, patient, and loving wife Rachel. Rachel makes the hard times bearable and the good times incomparable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This dissertation describes a system for proactive management of power and performance trade-offs through greater cooperation between applications and hardware. To enable such a management system, a software framework for application-guided power-aware control systems was developed. This system allows an application to guide the underlying computing hardware through a reusable and modular software abstraction. This abstraction layer enables an application to avoid hardware-specific details while still requesting resources from the computing hardware using a generic quality-of-service (QoS) interface. The computing system, in turn, monitors its current power and performance state and notifies the application to adjust its computational load by changing its algorithms. This two-way communication between application and computing platform allows both application and system designers to create proactive strategies for managing power and performance states.

This dissertation describes mechanisms for system state estimation, prediction and management and introduces speculative threads for transient management in switched systems. Two methods for power and performance management are tested: a situational-aware governor and a Q-Learner-based quality-of-service manager (2QoSM). The implementation of the software framework was tested using an autonomous robot. The framework and QoSM allow for significant power savings with minimal performance penalty as well as the flexibility to explore different computing platforms and machine learning techniques in the future.

CHAPTER 1
OVERVIEW OF SYSTEM ARCHITECTURE AND CONTRIBUTIONS

Many sources, including the Semiconductor Industry Alliance, estimate that computing systems will consume 20% or more of the global energy budget by the year 2025 [1, 2, 3], making a strong case for the importance of research on power management in computers. The goal of power management is to reduce the amount of energy needed to complete a task while maintaining a certain level of performance. There are some basic power management strategies that are used across scales of systems, from micro-robots to supercomputers, even though these systems have very different tasks and power constraints. In embedded systems, such as those found in mobile robots and smartphones, power budgets are limited by batteries.

This dissertation describes a generic software framework that allows for application-guided control of hardware resources in order to better manage power and performance trade-offs. Applications are generally designed to work completely isolated from the underlying hardware. Similarly, the operating system and the hardware it controls are designed for the execution of arbitrary applications and thus takes a general approach to interaction with programs. When dealing with single-purpose applications, such as an autonomous vehicle or a enterprise or scientific computing system, this isolation becomes less of an advantage and more of a hurdle to optimizing performance.

This research explores a power and performance aware framework that allows for applications to communicate their needs to the hardware and hardware to inform applications about resource availability. This framework, a type of middleware discussed in detail in Chapter 5, attempts to balance isolation and communication between software and hardware. The experiments in this research are primarily focused on embedded systems but the software mechanisms described in Section 3.3 and future work in Chapter 7 make the case

for experiments using this framework in enterprise and high-performance systems as well.

Historically, controllers for physical systems have been designed using one of two approaches: designing the control algorithm without considering the underlying embedded computing system and then choosing a computing platform that can handle the worst-case operating conditions for that controller, or selecting a computing platform first and then designing the best quality controller that will run on that platform. These standard design approaches assume a fixed-computer hardware system and are generally designed without taking the computing system's run-time flexibility and reconfigurability into account, imposing false constraints on the computing hardware design space. The computing system itself is a dynamic system with response times and accuracy levels that can be controlled by trading off power and performance.

The design space of control systems implemented on embedded computing platforms can be made much larger by developing mechanisms that couple the power/performance controllers of the computing system to the physical system controller. Then, during various phases of operation of the physical system being controlled, decisions can be made at a high level to trade-off power and performance in both the physical system controller and in the computing platform. As a result, the overall design might be able to use a much lower power, lower cost, and lighter weight computing platform than is possible with the current design strategies. Power management features are either ignored or disabled when implementing a controller to ensure reliable and deterministic operation at the desired sampling rate.

Fundamentally, this research is the development of a runtime coordination methodology for the computing system controllers and the physical system controllers. This work allows coordination by approaching the problem from three directions: methods of state-observation and power-management (Chapter 3), mechanisms for managing challenges introduced power- and performance-aware control algorithms (Chapter 4), and creating a framework in which details of hardware and application can be abstracted away to allow for more powerful and universal power-management strategies (Chapters 5 and 6).

1.1 Motivation

Enabling the runtime interaction between the application and the computing system controllers means that decisions can be made at a high level as to when to perform power savings and when to require highest performance. Some motivating scenarios highlight the need to couple these systems. In each case, the computing system could save power by slowing down CPU speed and idling extra cores during periods when the application performance requirements can be lowered. Other times may call for high-performance execution during burst periods (since processors are not usually allowed to run at their peak performance for long periods of time due to core temperature limiters), for example:

- A mobile robot may have different user spaces in which the position and velocity specifications differ; for example, the mobile robot might have tighter tolerances as it moves more quickly or moves closer to obstacles versus moving in an uncluttered environment.
- A setpoint control algorithm operating near its target might allow for the sampling period to be increased and the processor put into a power savings mode.
- A robotic swarm might have operating regimes where they need to be closer together, thereby requiring faster response times and better performance from the processor.

1.2 Design of the Software Framework

The compute-aware control system must be built upon a framework that provides a communication channel between the low-level computing hardware and the application layer software running the physical system controller. Moreover, it must provide a compute-aware manager that makes decisions on the Quality of Service (QoS) that the computing system should provide to the physical system controller and what mitigation strategies must be in place for the physical system controller if that desired level of QoS is not achievable.

1.2.1 Design Considerations

From the bottom looking up, hardware is often unaware of application-specific needs. For example, in a traditional operating system, an application does not convey to the hardware its specific timing needs in order to ensure that its deadlines are met. Instead, the onus is on the process itself to make sure that it runs quickly enough to maintain its desired performance. Even in a real-time operating system designed to meet application timing constraints, tasks are manually assigned priorities and must set their own deadlines.

Another complicating factor of comprehensive power management systems is the lack of a communication between high-level processes and the device's hardware. There is no standard application programming interface for applications to communicate their needs to the operating system which can, in turn, adjust hardware to meet requirements. Except in very limited cases, the operating system, and thus underlying hardware, is entirely reactive to the needs of software. This framework allows for a proactive approach to be taken both by operating systems and applications.

In multi-programmable systems like PCs, servers, and mobile devices, an operating system is needed in order to manage the resource needs of multiple interactive programs, scheduling time on the processor, sharing resources, etc. However, given a single-application system such as a robot, it could be argued that the most efficient use of hardware would be complete control of the underlying hardware by the application. In this case, much of the resource-sharing and protection offered by the operating system is not needed and the reactive allocation of resources becomes clumsy. An ideal application could allocate resources proactively when entering different regions of execution.

In this case, there are a number of serious drawbacks. First, the added complexity of managing specific resources is a significant burden for an application. Second, each application must be written for a specific hardware down to the driver level, requiring a massive amount of work from the designer. Third, because of this close coupling of software to hardware, there is almost no portability across systems.

1.2.2 System Level Requirements

Given the design considerations discussed above, a desired feature of the compute-aware framework is that it be a compromise between application control and operating system control of resources. This application guidance would allow for proactive resource management that works in concert with the multiple levels of existing resource management that exists from the OS to the hardware.

Fundamentally there must be two system-level modifications. First, there must be changes to applications to allow for communicating needs to the underlying hardware. For example, an object tracking application that detects multiple objects requiring greater computational complexity must be modified to recognize this state in order to notify the operating system. Second, the operating system must have a way to receive guidance from the application and allocate resources accordingly.

The first step towards establishing a connection between hardware and software for the purpose of power management was to develop a high-level guideline for the interaction between subsystems. In order to successfully balance quality-of-service (QoS) and power management, it is necessary to implement two separate processes, one that is capable of monitoring and evaluating the QoS, and one that is capable of taking that evaluation and adjusting power use accordingly.

Note that the term "Quality of Service" is often used in the context of communication networks [4, 5] but can be used in a broader sense to mean the requested performance level of any discrete event system (DES) [6]. For example, QoS is used to quantify the performance of distributed applications [7], task scheduling [8], virtual machine placement [9], or mobile applications [10]. This is closely related to a service level agreement (SLA) in the context of enterprise applications [11].

Figure 1.1 shows the high-level architecture of our system. At the top is a Compute-Aware Application (CAA) which we define as an software program with multiple controllable performance modes or profiles. There are many examples of applications that are

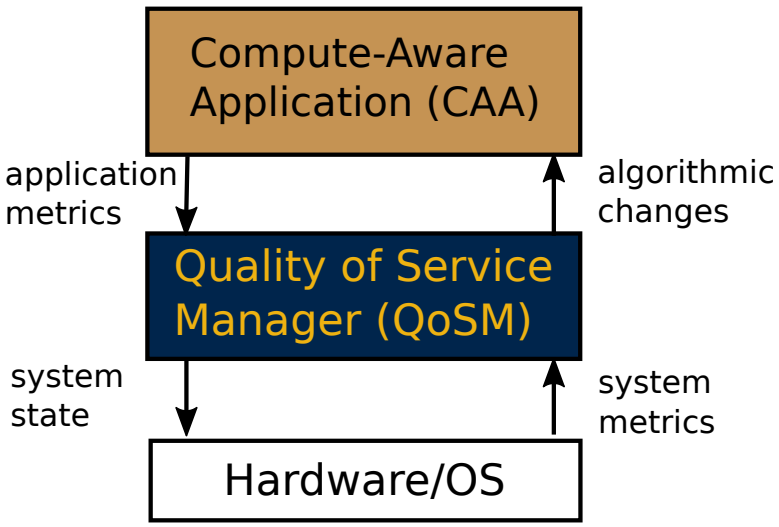


Figure 1.1: A high-level view of the software framework

created to be compute-aware. One familiar example is video decoders that adjust resolutions and framerates based upon available computing resources. These applications may have pre-defined discrete settings that allow for varied performance or they may be anytime or imprecise algorithms which improve performance monotonically as they are given more computational resources. No matter the type of application, it needs only two features to be integrated into our system: the ability to switch between algorithms of various computational complexity, and one or more metrics of internal performance.

The hardware shown in Figure 1.1 is abstracted and managed by an operating system. Much like the compute-aware application, the operating system must have two features: the ability to adjust resources and a set of metrics to describe power and performance states.

The quality-of-service manager (QoSM), shown in the middle layer of Figure 1.1, is the fundamental component in coordinating the application with the computing hardware. This can be broken into different modules, but it receives arbitrary power and performance metrics to determine its current state, and adjusts application and hardware to attempt to reach an overall power and performance target. Because of the modularity of the QoSM, the specific parts can vary tremendously in structure and complexity depending on the overall system needs and abilities. This dissertation covers both a situation-aware dynamic power

manager (see Chapter 5) and a more complex machine-learning algorithm (see Chapter 6).

What is essential to this framework is that as long as the required features are implemented (metrics and settings), the application, hardware, or middleware can be freely changed, even at runtime. This allows for a delicate balance of isolation and integration that is both useful and novel.

In order to create this framework, the following pieces of work were completed:

1. Mechanisms for System State Estimation, Prediction and Management (Chapter 3)
2. Speculative Threads for Transient Management in Switched Systems (Chapter 4)
3. Situational-Aware Methods for Power and Performance Management (Chapter 5)
4. Compute-Aware Management using Reinforceent Learning (Chapter 6)

The remainder of the dissertation describes this work in detail.

CHAPTER 2
LITERATURE SURVEY

2.1 Introduction

To place this research in the context of the state-of-the-art, it is necessary to cover a broad set of topics. Some background is included in the following chapters as necessary, but this chapter covers computer power management in depth (Section 2.2), application guidance (Section 2.3), quality-of-service management, middleware, and power-aware computing (Section 2.5), and imprecise and anytime control algorithms (Section 2.6)

Common embedded processors have undergone dramatic improvements in recent years, both in processing power and in reconfigurable features. Low-level power management and temperature control features, for example, previously only found in desktop and server CPUs, allow for dynamic voltage and frequency scaling (DVFS) of the processor [12, 13] and changes in its sleep or idle state [14]. In addition to DVFS and sleep state control, embedded processors adding symmetric multi-processing (SMP) which have the ability to migrate loads across cores and turn on and off cores. Since the power consumption is directly proportional to clock frequency, voltage, and activity, (see section 2.2.3 for more detail) adjusting these parameters can make a significant difference in power consumption.

2.2 Power Management

Power consumed by a computing system is

$$P = P_{CPU} + P_{MEM} + P_{DISK} + P_{PER} \tag{2.1}$$

where P_{CPU} is the power consumed by the CPU, P_{MEM} is the power consumed by the off-chip memory system, P_{DISK} is the power consumed by non-volatile storage, and P_{PER}

is the power consumed by perhipherals. The description of these subsystems can vary depending on the type of system, but generalizations can be made. P_{CPU} includes the power consumed by the processor and any on-die subunits (e.g. memory controllers and caches). In a system that uses graphical processing units (GPUs) as accelerators (e.g. high performance scientific computing), GPUs may be included in the P_CPU term, whereas in other contexts, it may be considered a perhipheral. In the context of this research, GPUs are considered computational units and thus can be included in P_{CPU}. CPU power is discussed in detail in Section 2.2.1.

The P_{MEM} term usually includes all off-chip byte-addressable memory such as dynamic random access memory (DRAM) or storage class memories (SCM). This excludes CPU caches (e.g. L3) as well as fast block-addressable storage (e.g. SSDs). P_MEM is discussed in more detail Section 3.3.1.

P_{DISK} and P_{PER} are mostly outside the scope of this research, though a number of the techniques referenced in this work have been used to manage these types of devices. In addition, our framework allows for the management of any hardware that meets very general requirements of observability and controllability.

2.2.1 CPU Power Management

The ubiquity of high-powered processors in nontraditional applications (e.g. mobile devices, embedded systems, IoT) has necessitated the need for comprehensive power management strategies. Computers use a broad set of methods to conserve power, from the CPU architectural level, board layouts and peripheral selection, through firmware and operating system support of low-power states, all the way up to user-controllable power modes. Each of these areas have been studied extensively for decades, and with the explosion of battery-powered devices from smart phones to mobile robots and ever increasing performance of processors, the need for more efficient and smarter power-management techniques continues to grow.

The power dissipated by the computational units in 2.1 is described by [15] as

$$P_{CPU} = p_t(C_L * V * V_{dd} * f_{clk}) + I_{sc} * V_{dd} + I_{leak} * V_{dd} \tag{2.2}$$

where p_t is the probability of activity, C_L is the capacitive loading, V is the voltage swing, V_{dd} is the supply voltage, f_{clk} is the frequency of the clock, and I_{sc} and I_{leak} are the short-circuit and leakage currents, respectively. The capacitive loading of the circuit C_L [16] and leakage current I_{leak} [17] are a function of the underlying digital design and thus are not applicable for dynamic power management. While the short-circuit current I_{sc} is caused by transitions and thus directly related to the frequency of the clock, it is based upon transistor design [18] and can be treated as a constant consumption. Therefore, when dealing with the dynamic power consumption, we can simplify [13] the equation to

$$P \propto V^2 f \tag{2.3}$$

Because voltage and frequency are the only two readily modifiable components of CPU power consumption, power management research has focused on voltage and frequency scaling. Chandrakasan *et al* described in detail the challenges and methods for static low-power digital design and dynamic voltage scaling [15]. Since power dissipation is dominated by the voltage term, much of the previous research focused on reducing voltage of a CPU both statically (in design) and dynamically.

The most obvious method of hardware power management is to simply turn off a machine when it is not being used. Initially, this was a manual process, necessitating a physical interaction to restart a system. This manual power-off was automated through firmware in the BIOS using Advanced Power Management (APM) [19] but this required platform-specific drivers in the OS. Eventually, this evolved into Operating System-directed configuration and Power Management (OSPM), an operating system sub-system that controls a computer's underlying hardware using a standardized interface, defined by *ACPI*. Advanced Configura-

tion and Power Interface (ACPI), first introduced in 1996, is a platform-independent set of standards for computer hardware power-management, configuration, and monitoring [14].

2.2.2 ACPI States

The most important aspects of ACPI for power management are system states. ACPI defines five primary types of states: global states (G-states), system states (S-states), CPU sleep states (C-States), DVFS states (P-states), and peripheral device states (D-States).

Global or G-states are user-visible power states enumerated from G0 to G3. In G0 (working), the system as a whole is running, although CPUs and devices may be in low-power or off states. G1 (sleep) is a sleep state, where the system maintains context but is no longer executing any instructions. When the system enters G2 (soft off), all context is lost and power usage approaches zero, but can be awoken via software such as wake-on-LAN (WOL). Finally, G3 (hard off) means that power is completely disconnected from the system and no power other than possibly a battery for the clock is consumed. System states (S-states) are also user-visible, enumerated from S0-S5. The S-states exist between G1 (sleep) and G2 (soft off). System state S1 is a low wake-latency state in which no CPU context is lost and coincides with G1. S2 is a deeper sleep state in which CPU and cache are halted, losing context in registers and cache, forcing data to be written back to main memory. State S3 is not very different from S2, only DRAM is refreshed (see section 3.3.1) less frequently. On many systems, states S2 and S3 are known as ”standby” modes. S4 is the lowest sleep state, known as hibernation, where main memory is saved to disk allowing a quick restore of the system without having to boot from a fresh state. S5 is soft-off, coincident with G2. Device states (D-states) are independent power-states of peripheral devices. Only two states are guaranteed to be implemented in a given ACPI-compatible device: D0 (fully running) and D3 (powered off).

Since the CPU is often the most configurable and high power component of the system, it has very fine grained power-management states: C-states and P-states. C-states, which

range from C0 to C3, define the operating status of a CPU. C0 means the CPU is currently executing instructions. If there are no instructions executing, the CPU is in state C1. The transition latency of state C1 is so low that the operating system does not have to consider whether to enter C1, meaning that as soon as there isn't an instruction that needs to be run on it, it can be put from C0 to C1. States C2 and C3 have a longer latency to enter and exit. In state C2, caches maintain state and consistency, while in C3 the OS is obligated to maintain cache consistency.

Schöne *et al* examine the implementation of ACPI states in Intel's Sandy Bridge and AMD's Bulldozer CPUs [20]. AMD and Intel have different implementation (and even naming) of their C-states, so in order for researchers and developers to create power-aware systems it's important to understand the platform-specific variation in transition latency and power reduction.

Finally, a CPU has fine-grained OS-controllable performance states. The highest performance state is always P0 and the lowest state is Pn. P-states are often implemented in hardware using *dynamic voltage and frequency scaling* (DVFS).

2.2.3 Dynamic Voltage and Frequency Scaling

Hong *et al* demonstrate the effectiveness of a voltage scaling heuristic through simulation of multimedia applications [21]. Pering *et al* developed a suite of mobile benchmarks and simulated a number of dynamic voltage scaling (DVS) scheduling algorithms [22]. The above research and commercial DVS focuses primarily on scaling only to a minimum voltage above the transistor's V_{th}. There is interest in near- or sub-threshold voltage computing, but this introduces significant performance and stability issues and requires support from both circuit-level design and semiconductor fabrication [23].

Since there is a practical limit to the minimum voltage of a CPU, clock frequency scaling has been examined extensively. Weiser *et al* explored scheduling techniques for reduced clock speed and were among the first to show the potential for significant power gains

from dynamic fine-grained frequency control by the operating system by using instruction traces from UNIX workstations [12]. Their work introduces what will become a common topic of optimization: a predictive algorithm for frequency scaling, in this case, using the amount of idle time in the past to predict future CPU needs. This technique was quickly expanded upon [22] as both CPU power and demand for mobile devices increased. Govil *et al* examine Weiser's PAST predictive technique and introduce a number of their own predictive scheduling heuristics [13] while Yao *et al* introduce a more formal analysis of the scheduling problem from a mathematical standpoint [24].

In 2000, Burd et al designed and fabricated an ARMv4-based CPU that could change the operating voltage [25]. Also in that year, AMD released their first DVFS processor, the low-power version of the K6-2E+, which allowed for control of the CPU frequency and voltage from a software settable machine specific register (MSR) [26]. By the mid-2000s, more finely-grained frequency controls were commercially introduced into all CPUs, allowing for implementation of the scheduling algorithms discussed previously. Nearly all modern CPUs and GPUs have a form of DVFS, allowing software control of frequency, and thus power consumption. The ARM Cortex-A15, one of the processors used in our experiments, ranges from 200MHz to 2GHz in 100 MHz increments [27].

Another related DVFS use case available in commercial processors are bursts of high-power or "turbo" modes that allow processor cores to run faster than the base operating frequency if the processor is operating below rated power, temperature, and current specification limits [28]. While the entire set of CPUs cannot necessarily run at the highest possible frequency at the same time or all of the time, because each processor core can have a different C- and P-state, an individual core can increase performance if there is adequate cooling and power available [29].

Looking at a higher level than operating system scheduled DVFS, modern Intel processors have a feature known as running average power limit (RAPL) that allows for a user defined power budget to guide the CPU's low level operating modes [30]. RAPL features

have also been used to estimate memory power [31, 32].

2.2.4 Operating System Power Management

The operating system uses the above described lower-level mechanisms to construct higher-level paradigms for power management. Given the resulting fine-grained range of power and performance states, the challenge then lies in choosing the correct state for a target application and system power. This problem, in general is known as dynamic power management (DPM) [33]. Determining the frequency is often done in a very simple manner. For example, the on-demand governor in Linux determines frequency entirely by the system load. [34, 35].

Rizvandi *et al* demonstrate that there is an optimal frequency for a given task [36] and Ahmed *et al* show experimentally it is possible for the OS to predict this frequency for periodic real-time tasks [37]. Recent work has looked to not only find the optimal DVFS state [38] but also, when dealing with heterogeneous computing, the optimal core [39].

More sophisticated power management strategies have been developed for controlling DVFS including integrated run-time systems such as CPU MISER [40] or CoScale [41], or Intel's RAPL system for setting energy quotas in hardware [31]. Another paradigm for saving power using DVFS is known either as race to idle [42, 43] or race-to-halt [44]. This technique relies on modern CPUs having extremely low transition latency and low-power idle states so that the CPU can run at a very high performance state to complete the work quickly and then transition to idle.

The strategy of using DVFS states to manage power and performance relies on the assumption that the task is *compute bound* as opposed to *memory bound*. A computationally-bound process is limited by the number of instructions being executed and thus benefits from speeding up the processor's frequency [32]. If the program is memory bound, increasing the CPU performance will not speed up execution, and so other techniques must be used such as memory frequency scaling [45], offlining memory pages [46], reducing refresh rates

[47] as well as heterogeneous memory systems [48, 49]. Memory power management is discussed in detail in 3.3.1.

With some data showing diminishing returns from DVFS [50], researchers have looked at race-to-halt [51]. Race-to-halt, that is running at maximum frequency to service a task and then going to sleep, has shown promise [44].

2.2.5 Machine Learning Techniques for Dynamic Power Management

Due to the low latency of transitioning P-states combined with the significant power variations between states, there is a strong motivation for smarter power management strategies. Thanks in part to increased computational capacity, researchers now have the ability to use modern machine learning techniques to develop dynamic power management using neural networks and reinforcement learning. These techniques will be discussed in detail in Chapter 6.

2.3 Application Guidance

In the end, much of the power management is either reactive to coarse metrics of load or predictive based upon models. An alternative, especially when dealing with computing platforms that only have a single primary task, is to increase the interaction between the application and underlying hardware. While single-purpose systems often have multiple background processes running for secondary tasks, their primary objective is maximum performance from a single application. In this way, the management of enterprise databases or autonomous robots differ from a standard user-facing general purpose computer which often has a multitude of applications in various states of being run.

It has been shown in many contexts that general purpose operating systems by definition make compromises for flexibility that can hurt specific application's performance. Operating system implementations of caching [52], scheduling [53], virtual memory [54], and file systems [55] have all been shown to be not ideal for certain applications. Engler *et al* attempt

to overcome these shortcomings by exposing hardware resources to the application directly in Exokernel [56]. In Dune, Belay *et al* similarly give applications direct access to hardware resources but in a more protected manner by using modern virtualization features [57].

When dealing with memory, additional techniques can be used to give the operating system guidance in manageing hardware. In order to enable software-guidance of memory access patterns, `madvise()` is a system call that allows programs to advise the kernel as to the anticipated access characteristics of a given range of virtual memory [58]. This advice can be used by the kernel when prefetching data and freeing pages. For example, a program can specify `MADV_SEQUENTIAL`, which informs the kernel that it can aggressively read-ahead and discard the used pages when they are finished. When dealing with heterogeneous memories, the decisions the kernel makes have many potential benefits but it becomes a more complex decision. Instead of having a single piece of advice, Soft2LM implements a set of flags that can, like GFP flags, be combined to describe multiple characteristics of a page.

Jantz et al developed a framework to "color" pages in order to provide application guidance to the operating system for physical page placement [59]. Using *trays* to organize and place data on specific physical DRAM pages based upon application guidance showed 55% power improvements on a synthetic benchmark. The modifications to the kernel API in Soft2LM are similar but provide additional options for applications to identify data in heterogeneous memory systems.

`jemalloc` is a scalable concurrent `malloc` replacement which uses isolated regions of memory called *arenas* to reduce lock contention in CMP systems. It also uses `madvise` to release pages back to the system. Building upon `jemalloc`, memkind extends the well-known application allocation API (`malloc`, `free`, etc.) with the kind of memory requested [60]. By using the arenas of `jemalloc`, memkind is able to allocate memory of the desired type in a specific region of memory corresponding to the requested kind.

2.4 Hardware-Software Codesign

Using dynamic power management techniques in control systems applications can they impart nondeterminism in the response times of the computing process.

Most of the work on implementing control systems focuses on hard-real-time computing [61, 62, 63] where there is a strong emphasis on achieving determinism in the response times of the computer. Suppressing the power-savings and reconfigurability mechanisms of computing systems in order to achieve determinism both increases the power usage and wastes one of the best features of the computing system. Moreover, striving for hard-real-time behavior may cause computing systems to be over designed (in terms of speed) in order to handle the worst-case timing situations [64], which might arise only rarely.

The penalty for hard-real-time overdesign of computer speed is most felt for self-powered systems such as mobile robots, aerial robots, and underwater robots since additional processor speed generally requires more power and more weight. Relaxing the hard-real-time requirement to soft-real-time and providing methodologies, such as anytime control, to account for the rare worst-case timing scenarios may achieve virtually the same performance with much slower processors [65]. Being able to integrate smaller computer capacity without sacrificing performance leads to smaller size, weight, and cost [66]. Adding coordinated control/computer performance and power management strategies further reduces the total energy needed. The result is that the mobile device can operate longer and, in the case of robots, perhaps be more maneuverable. An extreme example is in dirigible vehicles in which small increases in payload result in large changes in the size/buoyancy in order to remain afloat [67, 68]. Another application is in the desire to make micro mobile robots.

2.5 Middleware for Quality-of-Service Management

The idea of software abstraction layers, also known as *middleware*, for managing application quality-of-service have been explored by a number of researchers. An early work by Li

et al was an attempt to balance the objectives of system and application in a distributed video system by the introduction of an application-aware QoS middleware [69]. Zhang *et al* developed a system, ControlWare, for QoS management in distributed real-time systems and introduce *convergence guarantees* that lie between hard and probabilistic guarantees [7]. Their use of feedback-control theory to manage resources based upon a QoS is similar to our system in concept but differs in application and scope. ControlWare is focused on distributed systems and thus requires coordination entities not needed for a single-application system. CoAdapt allows for dynamic coordination of accuracy-aware and power-aware systems [70].

Imes *et al* have developed multiple hardware and software agnostic frameworks for power management. POET, their C-based framework, minimizes energy consumption while maintaining soft real-time constraints of commonly used benchmarks [71]. They expanded upon POET to create Bard, a framework that allows for changing between power and performance constraints at runtime [72]. This work is the most closely related not only due to their framework design but also their use of the ODROID-XU3 as their test platform.

Our middleware diverges in some important ways. First, the performance target is not a computational metric, but rather performance of the physical system itself. For example, in the test platform used in this research, the performance target is the deviation from the desired pat of a mobile roboth, but this metric is not integral to the framework since any performance metric of the physical system could be used. While application performance is not linearly related to CPU power states, it does have a monotonic and predicable relationship. Physical-system performance, on the other hand, is more closely related to a network application in that the performance is not just a function of computational work completed (i.e. instructions retired), but rather a product of actuators and sensors interacting with a physical dynamic environment that may be non-deterministic.

Second, our example test platform is a multithreaded autonomous robot with more complex and strict requirements for performance. This robot application, while making

extensive use of sampled calculations, displays both periodic and aperiodic behavior.

Third, in addition to meeting application timing constraints, the computing system itself informs the application of excess or limited resources, giving it the ability to preemptively prepare by changing algorithms or operating modes. In the opposite direction, the application can request the hardware via the framework to an increase or decrease in resources when anticipating a changing operating mode

A related topic to application management and guidance of hardware resources is the concept of *power-aware computing*. This general paradigm attempts to include power budgets into hardware and software goals of "smaller, faster, cheaper" [73]. Certain mechanisms to enable this paradigm have already been discussed in this literature review, however there has been research into more comprehensive power-aware systems [74, 75]. Quality-of-service (QoS) is a common term meaning the performance of a network or telecommunications system, especially as seen from the end-user's point. Our usage of QoS in the context of compute-aware control systems is analogous in the sense that it is a measure of the performance as seen by the end-user, in our case, the high-level application. QoS has been used in the context of application performance with respect to storage systems [76]. A closely related avenue of research involves the implementation of a service level agreement (SLA) governor to monitor database performance metrics and adjust the hardware settings to minimize power while maintaining a given QoS [11] This idea was expanded upon into general-purpose productivity aware frequency scaling (PAFS) [77]. Power-aware QoS agreements have also been explored in MapReduce cloud systems [78], grid computing [8], and virtual machines [9] but there is little research into the use of QoS in control systems or generic software-guided frameworks.

2.6 Imprecise and Anytime Algorithms

When an application cannot successfully predict or control its operating environment, it is important to have algorithms which can give valid output under non-ideal conditions. In most

cases, computer computations are only guaranteed to have a valid output at the completion of the algorithm and any interruption of the calculations will result in incomplete or invalid output. *Imprecise computing* is a class of algorithms in which a computation may be aborted prematurely and a valid answer is still obtained [79]. Building upon imprecise computing are *anytime algorithms*, algorithms which give increasingly better results the longer they execute [80]. An iterative algorithm with a monotonically converging solution is an example of an anytime algorithm and can be considered an ideal candidate for imprecise computing. Of fundamental importance in designing an anytime algorithm is a well-defined quality measure to monitor the progress of the computation and allocate computational resources therefore Zilberstein defines three general metrics for constructing anytime algorithms: certainty, accuracy and specificity [81]. *Certainty* is a measure of the certainty of an answer expressed p]robabilistically, set building, or other methods. *Accuracy* is a measure of how close the current, approximate value is to the exact answer. *Specificity* is a measure of the level of detail in the results. Zilbersetein also describes seven properties of anytime algorithms: measurable quality, recognizable quality, monotonicity, consistency, diminishing returns, interruptibility, and preemptability. In order to determine the quality of the current answer, one method is to develop *performance profiles*, that is the expected output quality as a function of time [81]. Anytime algorithms span a broad set of applications; however, we are primarily interested in anytime implementations of path planning, computer vision, and sensor fusion. Boddy and Dean use performance profiles to determine the output quality of path planners [82]. Likhachev *et al.* developed an anytime version of the A* path-planning algorithm with provable bounds on the suboptimality of the solution [83]. The ability to determine the bounds and measure of suboptimality makes this useful as an anytime algorithm. Iterative image processing algorithms are good potential candidates for anytime implementation.

2.7 Transient Management

Undesirable transients can occur in control algorithms or digital filters whenever the controller or filter is switched abruptly from one configuration to another. This presents a significant challenge when starting a new filter or controller which has memory, because on execution, any states must be either loaded from default, zeroed, or calculated. No matter what the state population technique, any transients must either anticipated and handled or allowed to settle before the filter is activated. There has been a lot of work done in the area of transient management in reconfigurable digital filter and control systems [84, 85, 86] Most of this work considers the case when the structure of the filter or controller remains constant, only the parameters change.

However, in the domain of digital filters and controls, much more drastic changes can occur due the architecture of the individual algorithms being used. Namely, in situations where resources change sporadically, filters and controllers can switch between algorithms of very different complexity. Transient management then becomes even more important, especially in those cases when switching from a low-order filter to a high-order filter, since the initial conditions need to be chosen appropriately.

One approach to transient management in switched systems is generally termed "bumpless transfer," which is a transient management strategy that aims to maintain a smooth output response in the presence of abrupt changes in the control or filter implementation; see, for example [87], [88], and [89]. In essence, the bumpless transfer technique uses various methods to compute the appropriate initial conditions for the new algorithm that would match the output of the old algorithms over some time interval. This eliminates "bumps" in the output, but comes at the expense of computational effort, which grows higher as the filter order increases. This can result in delays and/or increased latency in the switching time.

CHAPTER 3
SYSTEM STATE ESTIMATION, PREDICTION AND MANAGEMENT MECHANISMS

In order to make informed power and performance-aware decisions, certain mechanisms are required. The work in this chapter addresses three of these needs: system state estimation, system state prediction, and heterogeneous memory management

System state estimation is critical for power management because accurate measurements or models of power consumption and performance are needed to make decisions. In Section 3.1 we examined a number of commonly used metrics to evaluate their relationship to measured power consumption and performance. The research, published in part in [32], makes a case for *stall-cycle ratio* as a strong metric for estimating system state as well as an evaluation methodology for determining the best metrics for a given system configuration.

After successfully estimating the current state of the system, it is equally important to predict the effect that hardware changes (e.g. dynamic voltage and frequency scaling) will have on power and performance. Section 3.2 takes the data from Section 3.1 and uses an artificial neural network (ANN) to approximate a function of the relationship between a set of metrics and power and performance. This model allows for a dynamic power manager such as our quality-of-service manager (see Figure 1.1) to predict the changes to power and performance due to its actions.

While there is an understandable focus on CPUs as a target for dynamic power management, memory systems also consume a large portion of a computer's power budget. Different memory technologies (e.g. phase change memory) have varied power and performance characteristics and, especially when integrated into a heterogeneous memory system, can benefit from a power and performance-aware memory manager. Section 3.3 describes in detail the design philosophy and mechanism developed for power and performance-aware

memory management. This work was published previously in [49] and is expanded upon here.

3.1 CPU/DRAM Power and Performance Metrics

Fundamental to power- and performance-aware computing is determining the current state (i.e. power consumption and performance) of the computing system. Power is measured in Watts and can either be measured directly using specialized hardware, or estimated based upon CPU temperature, or modeled using measured performance. While power is a physical unit of measurement, "performance" can be different measurements depending on the context. Therefore the goals of the research described in this section are to identify which metrics are most important, examine correlations between metrics, and create models to estimate the state when available data is incomplete.

In order to determine the current state of the system, performance metrics are collected of the running system, and are used to measure and estimate different aspects of the computing system's performance. These metrics are collected using performance monitoring units (PMUs). PMUs are specialized on-CPU hardware for counting both hardware events (e.g. instructions retired) as well as software events (e.g. page faults) [90, 91]. These events can be either sampled periodically to evaluate run-time performance or they can be accumulated to determine the overall profile of an application [92]. From these measured events, a number of useful derived metrics are used to determine different aspects of the behavior of a system. For example, if cycles and instructions retired are measured, it is possible to divide instructions by cycles to create IPC which is a measure of the amount of time it takes to complete an instruction [93]. In addition to direct and derived hardware metrics, there are also software metrics available to determine the state and performance of applications. Various additional metrics have been proposed for different purposes (e.g. virtual instruction count for time invariant measurements [37]).

To enable an intelligent software framework, we would like to be able to estimate the

effects of changes to voltage and frequency to both power consumption and application performance. During much of the history of computing, there has not been a way to precisely measure in real time, the power consumption of a CPU or memory system. Any measurements that were taken were of the entire system, or large subsystems (e.g. everything on the computer's motherboard) and required significant external instrumentation. To estimate power in real time, models were developed to map performance metrics to power consumption [94]. Running average power limit (RAPL), introduced by Intel in their Sandy Bridge line of processors [31], allows researchers and system designers to obtain detailed estimates of energy consumption of a computer's subsystems. While some modern Intel processors have DRAM power-consumption domains available via the RAPL interface [95], most devices, especially low-power architectures, do not have separate DRAM power domains.

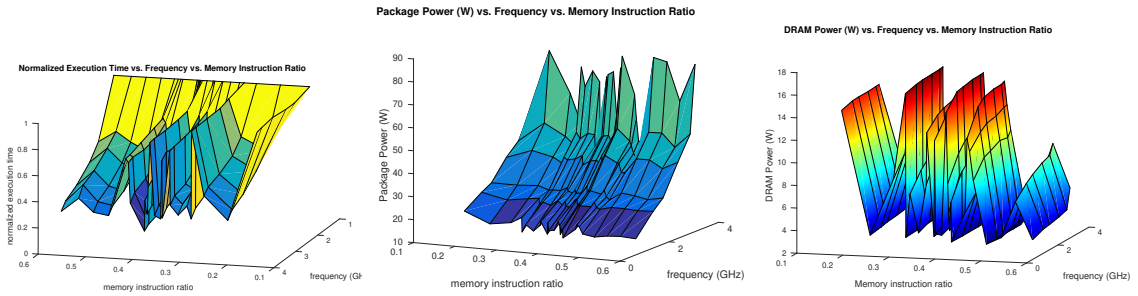
One portion of this work [32] evaluated metrics for estimating power consumption of DRAM from standard performance events available in most processors. Previous research relied heavily on models to validate power-performance relationships.

In addition, while finding a metric (or metrics) that can be used to estimate both CPU and DRAM power consumption, we also want a strong metric for estimating *boundedness*. Boundedness is an abstract concept that describes the qualitative characteristics of the execution of a program. An application is considered memory-bound or memory-intensive when its execution is limited by long-latency memory accesses, forcing the CPU to stall for data. A program is CPU-bound or computationally complex when the limit to execution is the maximum retirement rate of instructions. When a program is CPU-bound, any increase in the performance of the CPU will show an improvement in execution time.

Experimental results were obtained by running the SPEC CPU2006 benchmarks [96] while collecting power and performance metrics using the CPU's hardware counters. Data was collected on instructions retired, memory instructions executed, cache misses, and stall cycles. These measurements were taken at multiple frequencies (fixed during the execution

of an experiment) in order to quantitatively measure what the effect of frequency scaling has on the power consumption and performance.

In the following figures, benchmarks are ordered according to the metric along the front (y-axis), and the results for different frequencies of each benchmark are shown along the size (x-axis).



(a) Memory Instruction Ratio vs. Relative Execution

(b) Memory Instruction Ratio vs. Core Power

(c) Memory Instruction Ratio vs. DRAM Power

Figure 3.1: Memory instruction ratio (loads + stores divided by total instructions retired) [32]

Because the amount of time spent waiting for memory accesses is the primary determiner of the boundedness of a program, a naive choice for determining both memory power and boundedness would be the number of memory instructions compared to total retired instructions. It would seem that if a program was extremely memory intensive, this would be reflected in a higher number of memory instructions than computational instructions. Figure 3.1 shows the relationship between memory instruction ratio, that is the number of load and store instructions divided by the number of total retired instructions. Figure 3.1a shows the measured memory instruction ratio versus measured relative execution time (or speedup) at different frequencies. If a program is only bound by the number of instructions is can execute (i.e. compute bound), we would expect to see a inverse linear relationship between relative execution time and frequency. In other words, as the frequency increases, we should see a corresponding decrease in the time it takes to complete a given task. On the other hand, if a process spends the majority of time waiting for data (i.e. memory bound), we would expect a minor decrease in execution time as frequency increased.

Figure 3.1a shows a decrease in memory instructions compared to non-memory instructions from left to right and no clear relationship to speedup. The benchmark with the smallest memory ratio on the far right (libquantum with 0.17) shows the among the smallest reduction in relative execution time, while the benchmark with the largest memory ratio on the far left (hmmer with 0.57), shows a 70% reduction in execution time.

Memory instruction ratio is an attractive metric from a collection perspective because it can be simply calculated given a recorded instruction trace. However, this work shows that it is not an indicator of when an application benefits from frequency scaling. Furthermore, as Figures 3.1b and 3.1c show, it is not a good predictor of CPU or DRAM power consumption.

The obvious explanation of the failure of memory instruction ratio is the heavy use of caching and thus the benefits of *memory locality*, specifically whether the memory accesses show either *temporal locality* or *spatial locality* [97]. Temporal locality is when a given piece of data is accessed multiple times while still cached, amortizing the long initial memory access by having a number of low-latency cache hits. Spatial locality is due to a relatively large amount of data being pulled from memory with every single access. In DRAM-based systems, when a byte is accessed from memory, an entire page is pulled from a bank into a row buffer [98]. If a program demonstrates spatial locality, successive memory accesses will use data also found on the same row, again amortizing the cost of the memory access.

One way to indirectly measure the memory locality of a system is to collect the amount of cache misses and divide them by the number of cache accesses. The inverse of this ratio $1 - \frac{cachehits}{cacheaccesses}$ is known as the cache hit ratio.

As Figure 3.2 shows, there is not a direct relationship between the percentage of instructions that are memory instructions and the cache hit ratio. An application with relatively few memory instructions can have a very poor hit rate (e.g. libquantum) while an application with a much higher number of memory instructions can have very good memory locality (e.g. povray).

Figure 3.3 orders the benchmarks by cache hit ratio. In Figure 3.3a, we can see that a

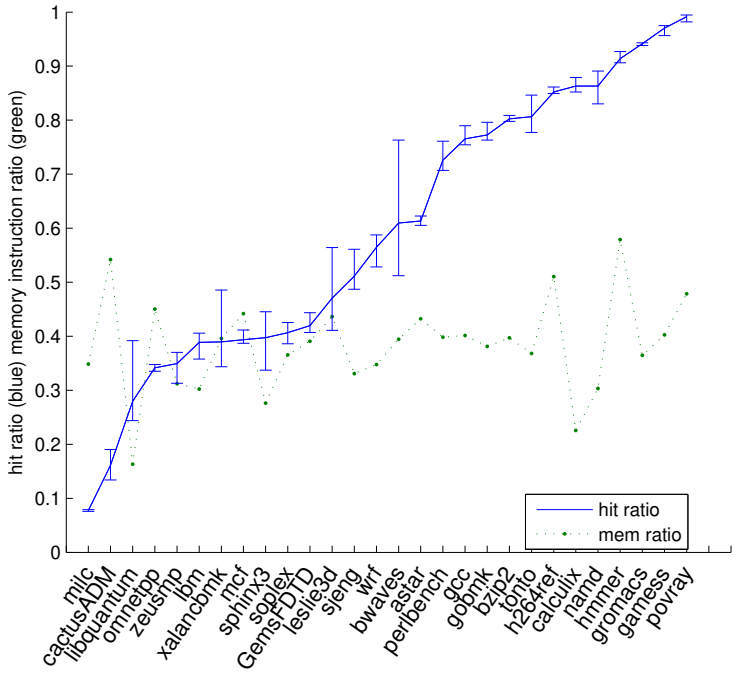
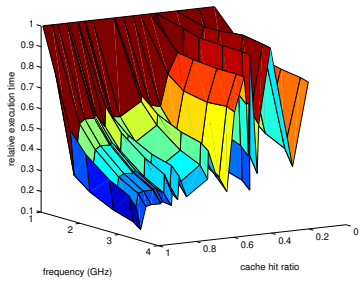


Figure 3.2: Memory Instruction Ratio vs. Cache Hit Ratio

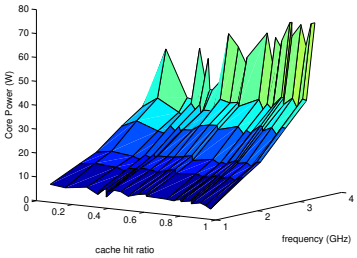
relationship begins to show. The benchmarks that show the higher cache hit ratio, show a greater speedup from a higher clock speed. There are still a few benchmarks towards the right side (low cache hit ratio) that show a substantial speedup, but across the SPEC benchmarks, we can predict that if the cache hit percentage is greater than 50%, frequency scaling will have a strong positive effect on execution time.

As for CPU power, a higher cache hit rate does correspond to higher CPU power (i.e. more CPU activity) but the relationship isn't strong enough to predict CPU power from this single metric. DRAM power is even less well correlated.

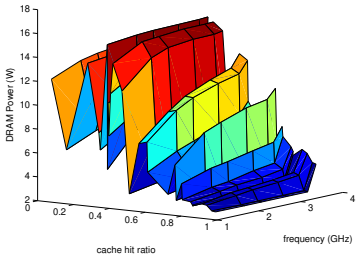
Another commonly used metric for evaluating the performance of a system is cycles-per-instruction (CPI) or its multiplicative inverse, instructions-per-cycle (IPC). If a CPU spends a lot of time waiting for memory accesses, the number of cycles per instruction will climb, indicating a more memory bound state. Figure 3.4 shows the CPI data presented in the same manner as the previous data. Immediately, it is possible to see that one problem when using CPI is that the metric is not normalized between $[0, 1]$. Because a multicore



(a) Cache Hit Ratio vs. Relative Execution

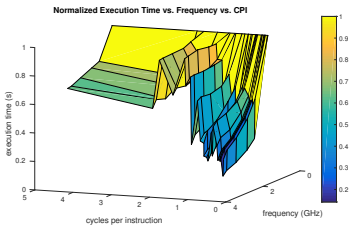


(b) Cache Hit Ratio vs. Core Power

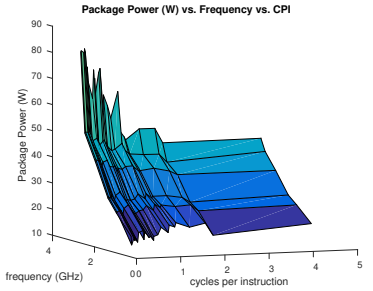


(c) Cache Hit Ratio vs. DRAM Power

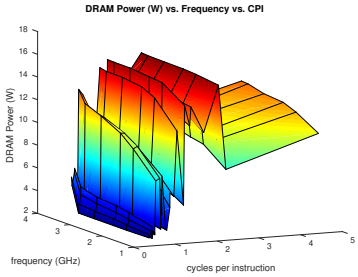
Figure 3.3: Cache hit ratio of SPEC benchmarks and associated speedup and power.



(a) Cycles per Instruction vs. Relative Execution



(b) Cycles per Instruction vs. Core Power

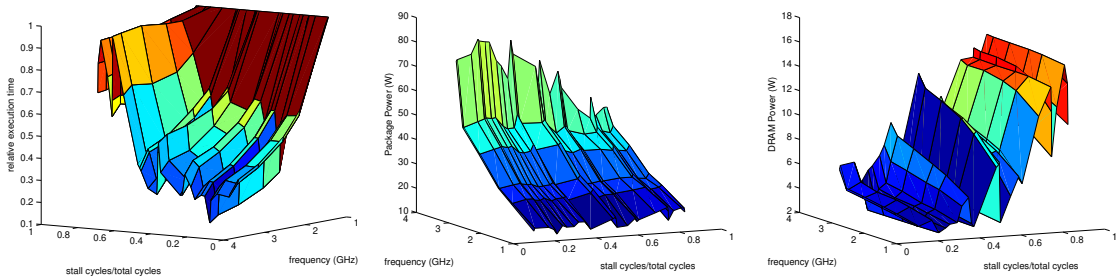


(c) Cycles per Instruction vs. DRAM Power

Figure 3.4: Cycles per Instruction (CPI) of SPEC benchmarks and associated speedup and power.

superscalar processor can execute more than one instruction per cycle, and technically there is no upper bound on the number of cycles it will take to execute a given instruction, it is not possible to get a normalized value. However, CPI does show slightly better, if compressed, prediction of speedup (3.4a) and CPU power (3.4b). Memory power, shown in Figure 3.4c, remains hard to estimate.

Finally, there is the most accurate single metric for estimating system performance and power, the *stall-cycle ratio*. A stall cycle is a cycle in which a CPU is waiting for data from memory, stalling the pipeline and stopping computations. The results indicate that the *stall-cycle ratio*, which we defined as the ratio of stall cycles to total cycles, is a strong measure of a program's boundedness. Figure 3.5a shows the relationship between the number of cycles a CPU is waiting to do work with the benefits of increasing the frequency



(a) Stall Cycles per Cycle vs. Relative Execution

(b) Stall Cycles Per Cycle vs. Core Power

(c) Stall Cycles per Cycle vs. DRAM Power

Figure 3.5: Experimental Results [32]

of the CPU. Thus the stall-cycle ratio, used as a proxy for measuring boundedness, can be a useful tool for management of DVFS.

In addition to a strong correlation between potential speedup and stall cycle ratio, this work showed promising results correlating DRAM memory consumption to stall cycle ratio. Figures 3.5b and 3.5c show a much closer relationship between stall cycle and power consumption of the CPU and DRAM respectively. In Figure 3.5b, as the ratio of stall cycles increases, the power consumption of the CPU core decreases, coinciding with decreased work done by the CPU. Figure 3.5c is noisier but it is clear that there is a correlation between an increase in stall-cycles and an increase in power consumption by the DRAM.

One problem with stall-cycle ratio is that while most Intel and AMD processors have this metric available, some commonly used ARM processors do not. Without this exact metric, we would like to evaluate the ability of multiple metrics to be used together to estimate power and performance. Because these relationships are non-linear, machine learning, specifically artificial neural networks, can be used to approximate a multidimensional function to estimate power and relative execution time.

3.2 Using Artificial Neural Networks to Predict Boundedness and System Power

For a number of reasons, artificial neural networks had not been used for optimizing the DVFS states for general purpose workloads. First, training is a computationally intensive

task that could potentially use as much CPU resources as the application itself. Second, workloads are dynamic and may contain multiple applications each running in multiple phases.

However, due to increased processing power and the amount of potential power savings from using machine learning techniques to set DVFS states, researchers have begun to explore the use of machine learning to predict power, thermal characteristics, and performance of computer systems.

Imamura *et al* used artificial neural networks to dynamically optimize two knobs: DVFS and thread allocation on NUMA (non-uniform memory access) architecture [99]. This work found that they could improve DVFS significantly. Narayana used ANNs to predict on-chip temperature [100]. Hesse *et al* use ANN to predict efficient network-on-chip DVFS states [101]. This work is a more direct approach to prediction, leaving the actual setting of DVFS states to the Q-Learner discussed in detail in Chapter 6.

3.2.1 Neural Network Design

The previously collected data (see Section 3.1) were used to train an artificial neural network to find a relationship between the metrics and system power and performance. The neural network was built using the Python TensorFlow [102] library with Anaconda 3.6 as a working environment.

The neural network will take metrics as inputs and estimate power and speedup as outputs. Because neural networks can find non-obvious and non-linear relationships between metrics, we decided to use as much available data as possible for our inputs. The chosen input metrics are shown in Table 3.1. This gives the neural network 9 inputs and 3 outputs. Since this is a dimensionality reducing problem, two smaller hidden networks were used to reduce the size of the input space to the output space in two steps. In the first experiments 7 and 5 internal nodes were used.

The loss function used was the L2 norm which was chosen because of it's ubiquity and

Table 3.1: Inputs and Outputs to the Neural Network

Input	Output
CPU Frequency	CPU Power
Cycles-per-Instruction (CPI)	DRAM Power
Cache Hit Ratio	Speedup/Boundedness
Load inst. per inst.	
Store inst. per inst.	
Memory inst. per inst.	
Load inst. per memory inst.	
Store inst. per memory inst.	
Stall Cycle Ratio	

straight-forward relationship. Future work could look at different loss functions.

Initially, the now shuffled 168-row dataset was split into 118 training points, 30 testing points and 20 validation points. This was chosen as an approximately 70/30/10 split, but because it isn't a large amount of data, round numbers were chosen. Generally when training neural networks, it's important to normalize the data, that is $x^{(n)}, y^{(n)} \in [0, 1]$.

A number of different learning rates were tested using batched gradient descent ($\alpha = \{0.001, 0.005, 0.01, 0.05, 0.1, 0.2, 0.3, 0.5\}$). The test cases learned for 1000 epochs which took approximately 500 s each test case.

Then, because the performance of neural network depended so much on the initial data selected (i.e. our results varied tremendously between different runs), we implemented k-fold cross validation. Since the data was already a multiple of 6, it was divided into 6 folds, running 1000 epochs, and saving the model with the best performance. Because of how long it would take to run all possible learning rates, we chose the best performing learning rate from the batched gradient descent.

In order to determine how well the neural network predicted, the output data was denormalized to see how well it matched the actual data.

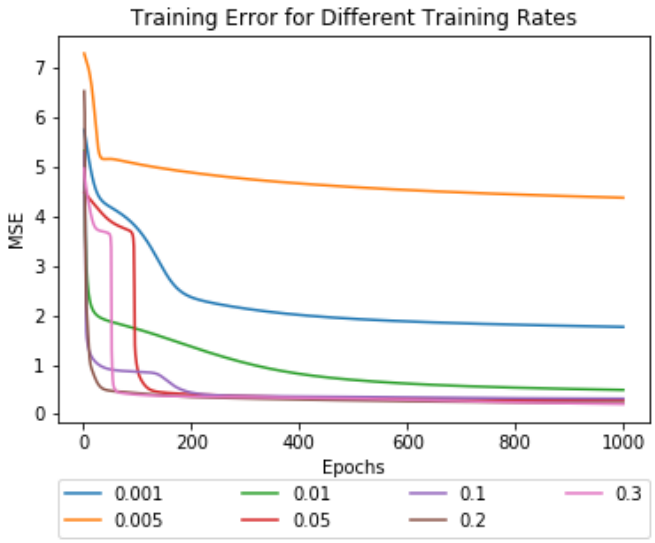


Figure 3.6: Error of different learning rates

3.2.2 Training and Evaluation of the Neural Network

The first step was evaluating the different training rates of the neural network. Figure 3.6 shows the change in error for various learning rates. When using a very small learning rate ($\alpha = 0.001, 0.005$), the error is much higher, never reaching the lowest levels in 1000 epochs. That the 0.005 is worse than 0.001 is likely just a function of where the gradient descent began, and it was trapped in a local minimum. With a learning rate of 0.01, it begins to converge to the higher rates. Once the learning rate reaches 0.1, it converges within the first 50 epochs and doesn't improve much past that. The best training error obtained with a $9 \times 7 \times 5 \times 3$ ANN was 0.268.

Next, Figure 3.7 looks at $\alpha = 0.2$ up to 400 epochs. While the training rate continues to improve, there's a slight increase in the test data error, likely showing the beginning of some overfitting.

As seen in the graphical depictions of various metrics compared to DRAM (Section 3.1), the data for DRAM power is extremely noisy compared to CPU power and speedup. This lack of correlation could be part of the reason why the neural network training error remained so high. In an attempt to better approximate the better correlated data, we removed

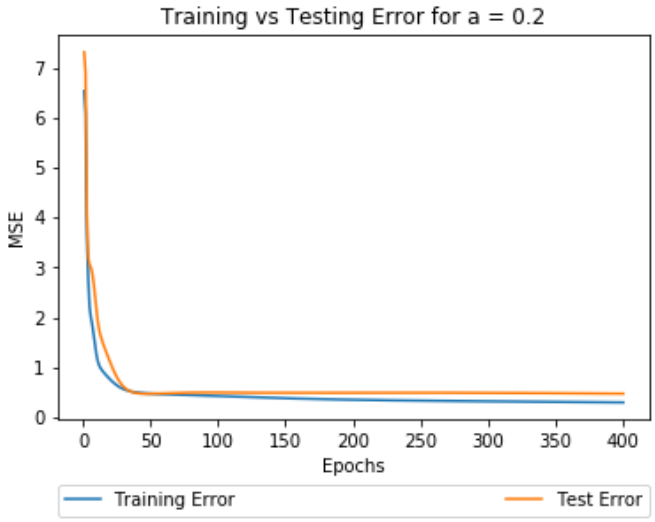


Figure 3.7: The beginning of overfitting

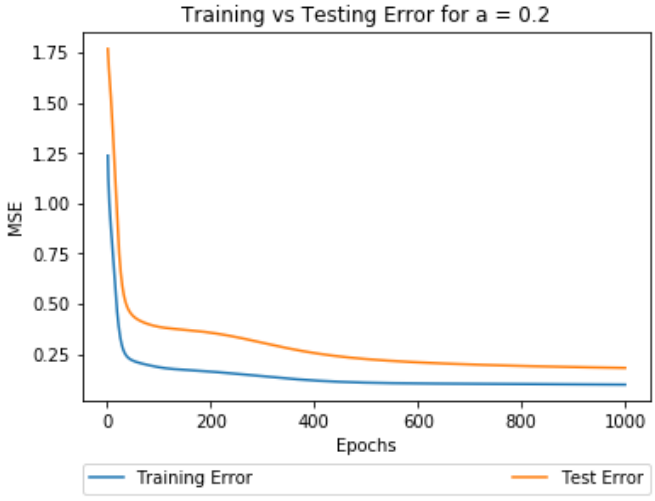


Figure 3.8: Results without DRAM Power

the DRAM power from our output Y and reran the simple neural network with a training rate of $\alpha = 0.2$. This experiment is shown as Figure 3.8. Without the very noisy DRAM power, there was a training error of 0.098 and a test error of 0.181. This seems to confirm that it is much more difficult to learn the DRAM power given our input space and the size of the neural network.

To focus simply on the DRAM power, we ran the neural network again training only with the DRAM power. In 1000 epochs, the DRAM-only training reached a training error

(a) Speedup Only (b) CPU Power Only (c) DRAM Power Only

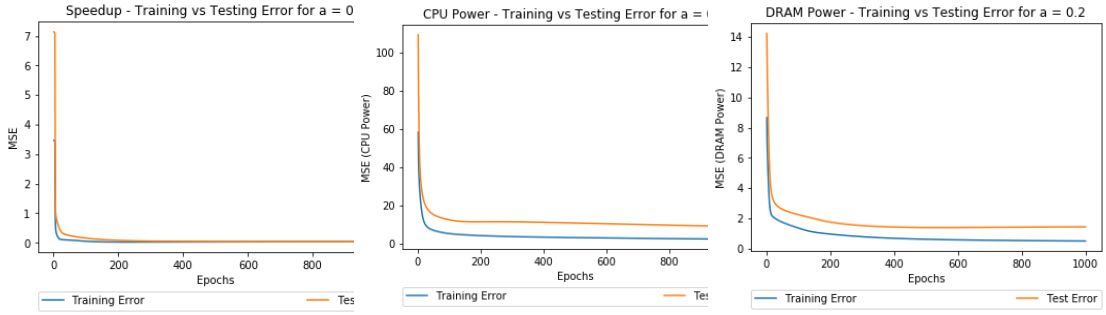


Figure 3.9: The performance of single-output training.

of 0.032 and a test error of 0.089. This gives a denormalized error of 1.439 W. This is a significant improvement and, given the noise in the data, the error of the combined output is more than the sum of it's parts. Since working on just the DRAM power did much better than the sum of the parts, we re-ran the training on only a single output for each.

The CPU Power training (Figure 3.9b) was similarly improved with a training error of 0.034 and a testing error of 0.124 giving a slightly less impressive denormalized error of 9.08 W. Finally, the speedup or relative execution time (Figure 3.9a) also improved, though it didn't do as well in it's training but did much better in it's testing. This is an interesting result and suggests that it is has created a better model for speedup than it did for either power, even thought the training error was slightly higher.

Because our previous neural network performed much better with a single output, the difference could be the amount of neurons associated with each output, so increased the hidden neuron count by 3x. This made a neural network of $9 \times 21 \times 15 \times 3$. It took longer to run, taking about 2000 s to train over 1000 epochs. But the overall performance was much better, obtaining a training error of 0.028 and a test error of 0.041.

The accumulated results of these tests are shown in Table 3.2.

The training of these large neural network take a relatively large amount of time, however the very act of collecting the data over multiple runs across different benchmarks takes many hours. Once the data is collected, it may take a few hours to train a neural network but

Table 3.2: Best Outputs of Individual and Group Neural Networks

Output	Training MSE	Testing MSE	Testing MSE (W)
DRAM Power	0.032	0.089	1.439
CPU Power	0.034	0.124	9.080
Speedup	0.039	0.043	-
All	0.268	0.347	-
All (3x)	0.028	0.041	-

it then can be used indefinitely for accurately estimating the potential speedup and power consumption as long as the underlying hardware configuration does not change. While this isn't a strong candidate technique for online learning or reconfigurable hardware, for a stable, static system, once trained it can be a valuable tool.

3.3 Software-controlled Two-Level Memory (Soft2LM)

The final enabling mechanism for this software framework is a method for handling heterogeneous memory architectures. Heterogeneous memory is an architecture with different physical types of byte-addressable main memory that share an address space (see section 3.3.1). Because memory technologies have different latencies, capacities, bandwidth and power-consumption characteristics, we would like the ability to place an application's data based upon not only the system's power and performance budgets, but also an application's own knowledge of the usage characteristics of its data.

Software-controlled Two-Level Memory (Soft2LM) [49] is a software policy for memory management in heterogeneous systems to improve the trade-offs between performance and power consumption. Soft2LM introduces a hardware agnostic approach that allows for region-based allocations, transparent page migrations, and an API for application guidance. This policy enables the application to request specific regions of memory based upon expected use cases and the operating system to allocate and migrate pages based upon their use.

3.3.1 Memory Power Management

While general power management techniques were discussed in detail in Chapter 2, to place this mechanism in context, we must explore in detail memory consumption, hybrid memory, and techniques for managing both.

Dynamic random-access memory (DRAM) is the most common type of computer main memory. DRAM has two primary qualities that make it attractive for main memory: low latency and fine-grained memory word accessibility [103]. Research has shown that that DRAM can consume 20% [104] to 40% [105] of system power with memory power consumption overtaking CPU as both DRAM capacities and CPU efficiency increase [46]. Kumar *et al* showed a reduction of memory power consumption with the application of frequency scaling techniques to memory [45].

A large portion of this power consumption is present even when the machine is idle due to *refresh power*. Because DRAM is dynamic, the entire memory space must be refreshed constantly by every row being read to the row buffer and rewritten. In most modern DRAM this refresh must occur every 64ms [106, 107].

Hardware techniques for reducing refresh power include keeping track of when a row has been read (and thus does not have to be refreshed) [47], timing refresh operations based upon predicted access patterns [108], and the inclusion of multi-bit error correcting codes to tolerate longer refresh periods [109].

Because there is significant variation in the physical retention properties of each memory cell [110] researchers have proposed identifying long and short retention pages and placing data accordingly [111] as well as offlining unneeded memory devices dynamically [46].

Owing to its high capacity, low latency, and persistence, storage class memory (SCM) can be integrated into current machine organizations in a multiplicity of ways: for example, as memory, as storage, as a hardware or software managed cache for either, and various combinations thereof. A consensus is growing around what is known as a hybrid memory system [48, 112, 113]. In these systems, NVM is logically inserted into the memory

hierarchy between DRAM and SSD, where it used as a transparent hardware-managed disk cache or as a shared-address space main memory, managed by the operating system's memory manager. The work discussed in Section 3.3 expands upon the DRAM-as-a-cache for SCM usage in two ways: it implements an efficient type of memory-to-memory paging in software, and in doing so, it also assigns to application software a steering role so that data is promoted or demoted between DRAM and SCM based on a combination of application input and application reference behaviors.

As with mixed SCM and DRAM arrangements, memory access latencies and bandwidths also vary in DRAM-only NUMA designs in which different latency and bandwidth characteristics apply for intra vs inter domain access [114]. These asymmetries, combined with SCM tier's persistence and power characteristics encourage a more informed scheduling of memory on the basis of joint participation by applications and the runtimes that host them in heterogenous memory. Shin et al expand on existing NUMA architecture in the Linux kernel, assigning NVM and DRAM to different NUMA node IDs, allowing the operating system to use existing NUMA migration to move pages between NVM and DRAM [115]. Adjacent physical pages are allocated into groups and to determine hotness of data, they use unused bits in the page table entry to store a weighted history of the pages' dirty bit. Their results show that even using separate NUMA domains, the overhead of migration in total execution time is only 1.14%, and they get a 19-36% decrease in energy consumption. The work discussed in Section 3.3 bypasses separate NUMA nodes, further reducing the codepath of different regions of memory. This also leaves NUMA available to build onto a heterogeneous system in the conventional way.

Lee describes a technique for using hybrid memory management that uses a hypervisor to scan and extract page access histories for tasks in guest virtual machines [116]. Based on these, the hypervisor can perform intra-VM and inter-VM allocation of capacity in on-chip stacked DRAM modules, so that relatively expensive off-chip DRAM accesses can be minimized for frequently referenced pages. Our technique is similar but does not require

a hypervisor as an intermediary, and therefore generalizes to single as well as multitenant execution. Hardware based techniques for managing stacked DRAM modules are described by Sim [117] and Chou [118]. Sim proposes a Part-of-Memory (PoM) architecture in which a page activity tracker guides hardware in remapping hot data to on-chip tier [117] , while Chou proposes a CAche-like MEmory Organization (CAMEO) for migrating referenced cachelines into on-chip tier while furnishing the total capacity of on-chip and off-chip DRAM to software [118] . A common characteristic of both approaches is that the latency and capacity of the stacked DRAM are comparable to those of last level caches, making the hardware approaches more fitting. The software-based approach taken throughout this research makes it possible for applications to benefit from both the larger capacity of the near tier and persistence of the far tier as well as the ability to draw upon a long history of page access patterns to make informed decisions about allocation and placement.

Intelligent page placement is also explored in the context of hybrid systems comprising CPUs and GPUs by Agarwal [119] and Li [120]. Agarwal developed bandwidth-aware page placement driven by both compiler extracted insights and explicit hints from software is used to show 35% improvements in GPU performance [119]. Their experiments show a marked improvement, but because hardware amenible to their algorithm was not available, they were forced to conduct their experiments in the simulator. Results from Li show that in comparison to a hardware managed approach, static assignment of hot data to on-chip DRAM doubles performance and cuts power consumption in half [120].

For hybrid memory systems comprising off-chip DRAM and off-chip SCM modules, a hardware memory controller is proposed by Ramos [121]; the controller monitors access patterns and remaps pages, while maintaining its own address translation table to keep such data movement transparent to application and operating system. Using simulation for such hybrid systems, Meza et al show power consumption and performance improvements under a hardware-based but software-assisted blended memory management approach – with significant gains resulting from displacing a hard disk drive with SCM, and further

gains arising from eliminating software overhead of file system calls [122] . This is in line with the findings presented in this research–that in lieu of using SCM as a faster paging device, it is a better choice to eliminate the I/O and serialization overheads by using SCM as extended memory into which colder virtual addresses are mapped. Using trace driven simulation, Seok et al conclude that it is imperative to reduce write accesses and energy consumption by taking into account the non-uniform latency and endurance of SCM [123]. Others have examined low-level hardware modifications to improve performance, reduce power consumption, and improve device lifetime: Yoon explored row-buffer aware caching policies [124], Qureshi optimized DRAM cache architecture for latency, even at the expense of hit-rate, by both reducing the associativity and streaming data and tag in a single data burst, and introduced a memory access predictor [125] while Meza added a small cache for recently used metadata in DRAM [126]. The approach proposed and evaluated in [49] and expanded upon in Section 3.3 is driven from a similar perspective but different vision: that due to their non-uniform power, latency, and bandwidth characteristics, SCM accesses need to be reduced to a minimum and that such reduction can be pursued sooner with software approaches. while minimal hardware extensions (e.g., in data access instrumentation) that can assist software are also identified sooner as a side effect.

Hybrid memory architectures can use DRAM as a buffer to SCM [48] or side-by-side [112]. The simulated system proposed by Dhiman *et al* show a 40% reduction in power consumption [112] while the hybrid system proposed by Park *et al* can reduce power due to refresh from 23-94% [127].

3.3.2 System Overview

Soft2LM proposes two additions to page management: the ability to guide page placement and monitor page usage, and a mechanism for transparent page movement.

Application Guidance

Application guidance of page placement requires an API that allows for anticipated use of memory allocations to be relayed to the operating system. This differs from `madvise` in that `madvise` is called after the mapping has been made. Our modifications pass information to the operating system at allocation time. We propose the needed information be passed to the kernel as additional flags in the existing `mmap()` call. Initially, the modifications are requests for specific regions of memory, `MAP_PREF_NVM` and `MAP_PREF_DRAM`, which map to the modifications in page allocation flags outlined in table 3.3 and discussed below. There is a lot of potential for use-based flags that can then be interpreted by the operating system based upon it's available physical memory.

If the application only prefers a specific type of memory (`MAP_PREF_*`), then the allocation will only fail if no memory is available when a page is actually allocated. Because `mmap()` uses lazy allocation and only brings in data when there is a page fault, the available memory may change during runtime, using preferences (as opposed to requirements) will prevent future allocations from failing.

If an allocation is required to be in a specific region, then we can specify that as well. If an application requires persistence of data (e.g. critical logs) `MAP_REQ_NVM` will ensure that data is written to non-volatile memory. If the application needs low-latency access or is doing heavy but temporary writing (such as a scratchpad), `MAP_REQ_DRAM` can be selected. When used in conjunction with the existing flag `MAP_POPULATE`, the page table will be prepopulated and the file will be read-ahead. This should ensure that, at allocation time, `mmap()` either gets the memory in the required region or fails immediately. If `MAP_POPULATE` is not used, there is the potential for a failure when a page fault occurs later.

Ideally, we would like for an application to be able to simply specify the expected use pattern for a given region of memory and have the operating system make allocations based upon the physical memory available, potentially leveraging many types of heterogeneous

Table 3.3: Software Advise Flags for Page Allocation Using `mmap()`

Basic, fail only with ENOMEM	
`MAP_PREF_NVM`	prefer NVM but accept DRAM
`MAP_PREF_DRAM`	prefer DRAM but accept NVM
Basic, fail if no required memory is available	
`MAP_REQ_NVM`	require NVM
`MAP_REQ_DRAM`	require DRAM
Usage based flags	
`MAP_READ_ONLY`	Write triggers protection fault and reflagging
`MAP_READ_MOSTLY`	Latency consideration in allocation, try to keep clean (writeback)
`MAP_WRITE_MOSTLY`	Latency insensitive, placement in far, persistent, and low-energy memory
`MAP_SCRATCHPAD`	For temporary data, latency and physical endurance primary concerns
`MAP_LATENCY_SENSITIVE`	Data without specific read and write patterns, but expected to be latency sensitive, similar to `reg` in C
`MAP_REQ_PERSIST`	Require placement in persistent memory or force write-through
Anticpated Use Patterns	
`MAP_HOT`	Data that is part of a critical set, allocate in near memory and attempt to keep close to CPU
`MAP_COLD`	Data that is expected to be used infrequently, allocate in far memory

memories. Common use patterns could be defined by the flags in table 3.3. Read only pages could be moved into DRAM as needed, but then moved down to NVM as they cool, and eventually being discarded without writeback when space is needed. Read-mostly pages, anonymous or file-backed, when located in DRAM could be occasionally synced with NVM or disk when dirtied, but otherwise kept close to the CPU for low latency accesses. Write-mostly regions of memory can be reserved for those where data is written consistently but without the need for low-latency reads, and thus can be allocated in far memory.

Scratchpad regions of memory are those which store temporary data used for ongoing calculations or processing. These data may experience a lot of writing and reading and thus should remain low-latency for both reading and writing, and in physical memory with high endurance. Latency sensitive memory regions are those which are known to be accessed frequently but may not have the high write access patterns, thus endurance is not a factor in selecting the underlying pages. If data needs to be kept persistent, the require persistence flag can force all allocations into non-volatile regions of memory, or in non-heterogeneous memory force write-through.

In addition to specific use patterns, the actual access frequency of the data can be specified as hot or cold. Hot data is given a preference for near memory, however the operating system may move data around as it sees fit. Cold memory is allocated in far memory and will not be moved to near memory without good reason (e.g. `madvise` update or operating system monitoring).

Migration Mechanism

The migration mechanism is built upon the Linux's memory management. It contains three primary changes to page management: split physical memory, epoch-based page migration, and region-aware memory management.

The physical address space is split into two contiguous regions, each one having its own data structures including per-order free lists, per-cpu active/inactive LRU lists, and

Figure 3.10: A sample epoch showing the large computational period and the smaller migration-related periods. [49]

page caches. *Epoch-based migrations* are used to amortize the cost of moving pages and remapping, as well as reduce thrashing. An epoch, shown in Figure 3.10, allows the memory management system to work in stages, collecting performance and power data during the execution of applications, using this data to make decisions about page movement, selecting pages to move, copying the data, and then committing the movement at the end of the epoch. Performance data is currently collected by the memory subsystem and performance counters, however it is expandable to receive data from other sources such as hardware sensors, daemons, as well as an application's own knowledge of its performance.

The decisions about data movement are similarly flexible. The initial implementation uses DRAM capacity thresholds to trigger a migration of data from near memory to far memory. The migration mechanism looks on the inactive lists for pages that can be moved to NVM. It is important to note that since the second level memory is byte-addressable, while there may be a performance penalty associated with the additional latency of second level, there is not the danger of swapping or having to re-read a page from disk. The page, once the migration is complete, will be accessible by applications without suffering a page fault.

If the stall cycles of a CPU begin to increase indicating a program becoming increasingly memory bound, the migration mechanism can work in reverse, taking pages from the second-level memory's active list, and transparently migrating them to DRAM. Future work will develop more sophisticated methods of identifying candidates for migration.

Whenever a shared page is remapped, any CPUs which have the page mapped must

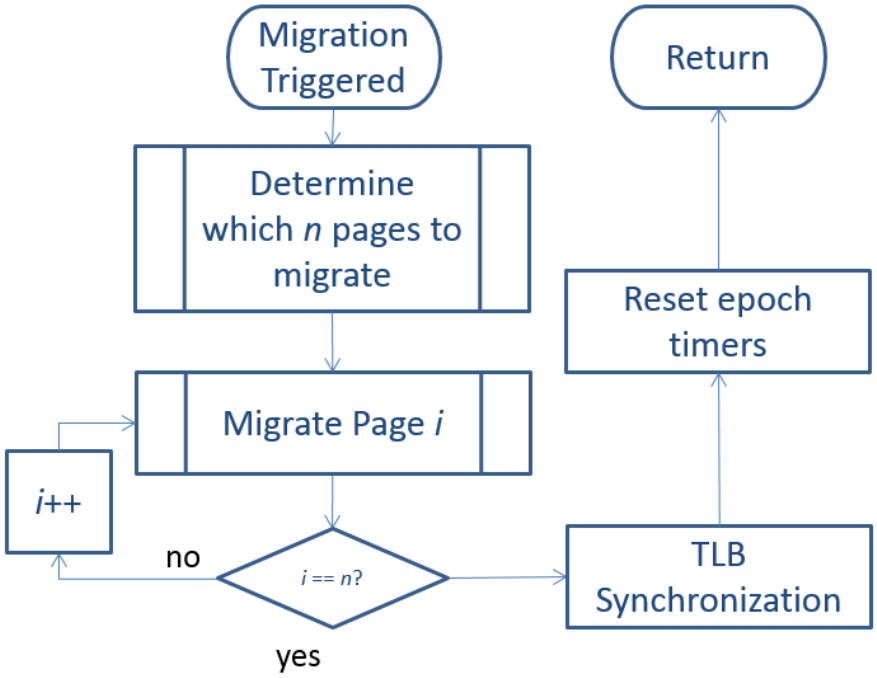


Figure 3.11: Flow chart of migration portions of epoch.

remove from the TLB the cached entry. In order to ensure that no stale mappings are used, the remapping CPU issues a *TLB shootdown* to remove the stale entry from all processors. This synchonization requires an inter-processor interrupt (IPI), an expensive operation that stalls the issuing CPU until all CPUs that receive the IPI acknowledge that they have removed the TLB entry. Currently, the actual act of migrating the data occurs right before the batched remapping, however the system could be expanded to perform data migrations in the background, staging the moved pages for a batched remapping at a later point.

Page Selection

The migration begins upon the expiration of a high resolution timer whose duration can be set via a `/proc` file. This timer interrupt places the migration task on a workqueue and returns. When the workqueue is executed, the selection of pages begins. The kernel checks the threshold of DRAM as well as the maximum number of pages to be moved, both set via `/proc` filesystem entries, to determine how many pages need to be moved. Much like `vmscan` scans LRU vectors `lruvec` to age and free inactive pages, the migration

mechanism scans these vectors looking for migration candidates. These LRU vectors are divided into four LRU lists: inactive file-backed pages, inactive anonymous pages, active file-backed pages, and active anonymous pages. In this order, the page vectors are scanned, first looking for unmapped pages that are clean and can be migrated asynchronously. Inactive, clean, unmapped file-backed pages may be discarded without penalty. However at this time, they are migrated under the assumption that second-level memory is large enough to accommodate most if not all of the total memory footprint. In addition, due to the persistent nature of NVM not requiring energy to refresh as in DRAM, once data is written back to NVM, there is no cost in keeping it there indefinitely until there are capacity constraints.

If moving unmapped pages is unsuccessful in getting the needed number of pages, the list is scanned for mapped, clean pages. Finally, the list is scanned for dirty pages. It is important to keep in mind that we are not writing back dirty pages when we migrate; there is no reason to worry about the page state when moving them to another active region of main memory. We are however using the dirty state of the page as additional information on the activity of the page. Since `vmscan` regularly passes through main memory, looking for pages to free, we can use the fact that it is dirty to assume that the page may be recently used. If the threshold can be met by simply moving cold/inactive pages to SCM but keeping them mapped, the pages are collected and staged to be moved. If there are not enough pages in the inactive list, pages are collected from the active lists. Once the candidate pages are selected, each page is copied to its new location. When the TLB flush is acknowledged by all nodes, the epoch timer is reset and the kernel cedes.

Migration Process

The detailed process of page migration is shown as Figure 3.12. It follows closely the migration used in Linux already by compaction, memory hotplug, and memory failure paths, however there are significant differences in implementation, due to different failure modes, allocation sources, and page selection. Once we have found candidate pages to migrate from

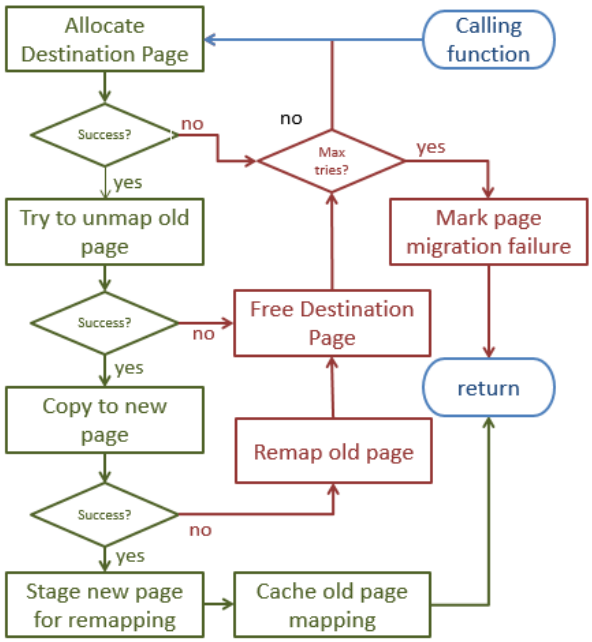


Figure 3.12: Detailed flowchart of migration. [49]

a region, the calling function attempts to allocate a page in the needed region using a pair of new page flags: `GFP_NVM` and `GFP_MIGRATE`. If we are unsuccessful in allocating a page, going all the way from page caches, down to the buddy system, we unwind and try again. Once we obtain a page, we attempt to unmap the old page. If this is unsuccessful, the destination page is freed and we try to allocate a new page. Once the page is unmapped, all the data and meta data are copied to the new page. While rare, it is not impossible for the data to not copy correctly, so if the data copy fails the page is remapped, and the destination page freed. If the copy is successful, the page is remapped, the old page mapping is cached, and the migration is complete. The caching of old page mappings is an important addition that will allow for *page aliasing*. Since a page, once migrated, is not immediately freed, but rather cycled through various states of cleanliness and activity, there is a good chance that at least for a while, the page may exist in two places at once, that is DRAM and NVM. This provides an interesting opportunity for an even more lightweight case of migration and an extremely powerful use of Soft2LM. Take for example a page that is NVM, but has been accessed often and thus should be moved to DRAM. If we have the old page mapping

cached and the NVM version is clean, instead of writing the page back from DRAM, costing bandwidth and write power, we can simply use the cached mapping to stitch the page back into the page table, and move the DRAM version of the page to an inactive list for future freeing.

3.3.3 Experimental Evaluation

There are three facets of evaluation to consider. First, given the added features of enumerated regions of memory and the modifications of memory management allocation and freeing mechanisms, we measure the introduced overhead by comparing the performance of a split-memory tiered memory manager with an unmodified kernel. Second, in order to show the benefits of a shared address space over block-based storage, we compare our tiered-memory manager with swap placed on a DRAM-based ramdisk. The choice of a ramdisk eliminates any device latency penalty that swapping would normally introduce, comparing only the costs of the mechanisms. Third, in order to evaluate the overhead of moving active memory pages during benchmark execution, we migrate 100 and 1000 pages per second, and compare the results to swapping.

Because our focus is on applications with large data needs, we chose the PARSEC benchmarking suite that contains 13 varied memory intensive workloads. [128]. Experiments were carried out in a KVM/qemu-based virtual machine running on an Intel Haswell i7-4770. Virtual machines with 4 Haswell cores (2 physical, 4 logical) and 6 GB of DRAM were created running Debian 8 and a modified version of the Linux kernel v3.14.39. Since there is no consumer-grade SCM available, our experimental evaluation was run using two regions of DRAM, and for the comparison with Linux's paging, a RAMDISK used as a swap partition.

PARSEC

The PARSEC v3.0 suite was selected for its diverse set of memory intensive workloads and its use in current research. All benchmarks were run with 4 threads in an attempt to reduce

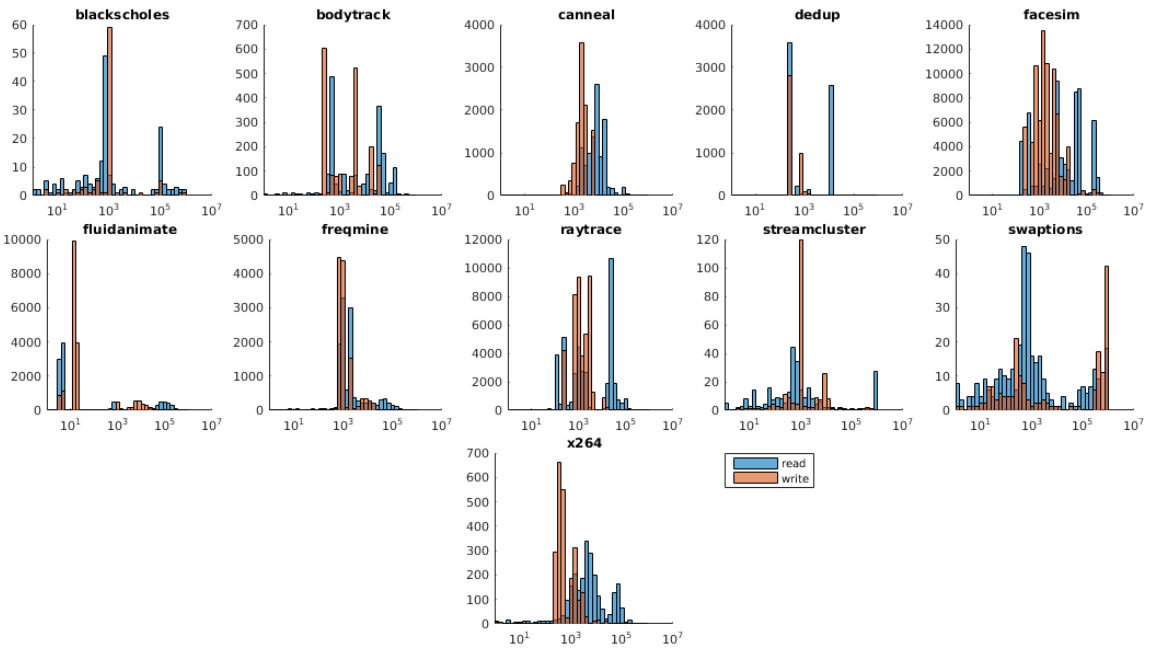


Figure 3.13: This figure shows the page access histograms for the PARSEC 3.0 suite of benchmarks running the `simsmall` datasets. Blue indicates page accesses for reads, while red shows writes. The distribution of page accesses across an extremely large spectrum, as well as the different access patterns motivates the need for intelligent page placement when dealing with heterogeneous memories.

the potential for CPU bottlenecks, which would reduce the stress on the memory system. The `native` size dataset was used for all runs. PARSEC measures the execution time of the benchmarks by using the Unix `time` command, giving us three time values for each run. The **real** time is the actual wall time elapsed during the execution of the benchmark, the **user** time is the time the process was executing on the kernel, and the **system** time is time spent in the kernel.

Memory Footprint of `native` PARSEC benchmarks

In order to determine the actual memory footprints of the PARSEC suite, during the runtime of all the benchmarks, the resident set size (RSS), virtual memory size (VSZ), and major and minor page faults were sampled every second and logged to a file. In the output of `ps`, RSS is the amount of an application's memory that is allocated and actually resident in RAM (i.e. non-swapped) while VSZ is the total size of the virtual memory including code, shared

Table 3.4: Memory Characteristics of simsmall and native PARSEC Benchmarks

	native (MB)		simsmall (unique pages)			
Benchmark	RSS	VSZ	unique	read	write	inst (M)
blackscholes	611	652	190	188	95	106
bodytrack	31	380	2096	1884	1868	301
canneal	938	1239	10630	10628	10430	607
dedup	1681	2621	6778	6777	4107	764
facesim	302	552	80326	65669	80073	11848
ferret	102	1330	2409	2394	1574	519
fluidanimate	597	642	19039	10176	18851	440
freqmine	705	941	12271	12231	12038	921
raytrace	1124	1444	42920	40261	42602	9519
streamcluster	106	223	419	417	232	420
swaptions	4	229	428	410	192	697
vips	41	361	1997	1588	1184	966
x264	181	310	2546	2440	2361	218

libraries, and swapped out portions of code. Table 3.4 shows the measured footprint while running the `native` size dataset. What is interesting about these measurements is how a significant portion of them do not have especially large datasets. For example, swaptions has only 4MB resident in RAM at any given time, while both bodytrack and vips have less than 48MB. Further work is being done to better profile these benchmarks to understand both their temporal access patterns and implementation of application guided allocations.

PIN-based traces of `simsmall` PARSEC benchmarks

In addition to running the full benchmarks on a full system, we examined the access properties of this benchmark suite. We used PIN [129] to capture memory traces of the PARSEC suite. Since the `simsmall` datasets generated nearly 700 GB of traces, the longest being over 10 B instructions, it wasn't possible to directly compare these data to the `native` datasets, however, we believe it gives some insight into the page access patterns, further motivating the need for intelligent memory management.

The right half of table 3.4 shows the unique pages recorded, and the number of pages read and written, as well as the number of memory instructions collected for the entire

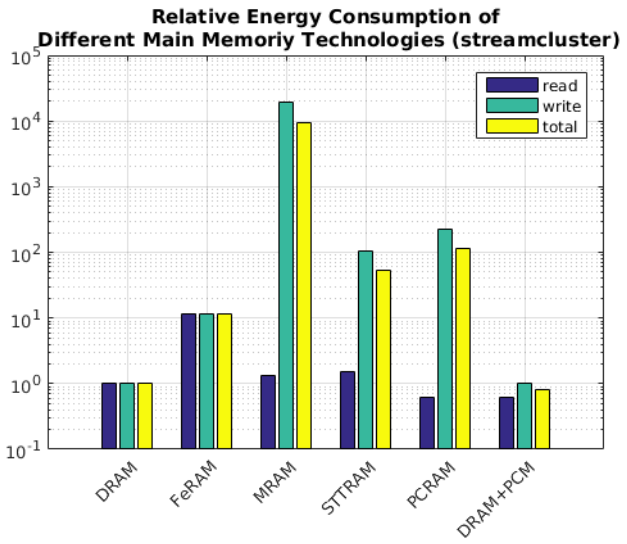


Figure 3.14: This shows the energy consumption of different types of memory on the streamcluster workload relative to DRAM. [49]

simsmall run. Here we can see the differentiation of written and read pages across all benchmarks. For example, in `canneal`, 10,630 total unique pages are accessed, with 10,628 read and 10,430 written, so nearly all data is consistently read and written back. Compare this to `fluidanimate` or `facesim` where nearly every page is written to, but 18% and 46% of these pages are not read during the execution of the respective applications. These differing access patterns not only emphasize the advantage of application/programmer guidance in identifying page use, but also the need for intelligent page placement.

Figure 3.13 shows the page access distribution for both reads and writes. The x-axis consists of bins of access frequency of a given page on a log scale, while the y-axis shows the number of pages in each bin. Here again we see much different access patterns of pages across benchmarks. Benchmarks such as `dedup` and `fluidanimate` have a large number of pages with very similar access counts, indicated by the sparse but tall spikes in the graphs, while `facesim` and `blackscholes` have wider distributions of page access frequencies. This is further evidence of the aid that applications can be in identifying the access patterns of pages.

Because we wanted to evaluate the entire execution of the benchmarks instead of taking

a computable section of the memory traces, we were unable to use a full memory system simulator such as NVMain [130] to estimate the power and performance on different memory topologies. Using energy consumption estimates for upcoming versions of different memory technologies [131], we calculated the power consumption of the workloads using the collected memory traces. Figure 3.14 shows representative data from streamcluster. This ignores the effect of the CPU caches, since the same application will access the same data across all memory technologies, and instead uses the estimated 2017 joule/bit data found in Moreau for all memory accesses.

With the exception of DRAM and FeRAM, all the candidate SCM technologies have asymmetric energy consumption for reads and writes. With a goal of minimizing energy consumption, DRAM+PCM data uses simple inclusive caching to service reads through PCRAM and writes on DRAM, both allowing each technology to operate in its lowest power region as well saving the endurance of writes to PCM.

3.3.4 Introduction of Multiple Regions on Performance

Moving to Soft2LM implementation, we will first examine the partitioning of physical memory in the kernel. In this section, we show the low cost of an active Soft2LM kernel in both in overall benchmark runtime as well as time spent executing kernel code, versus the unmodified Linux kernel v.3.14.39. The basic split-level memory management system does not migrate pages, it simply allocates pages in the lowest-latency region of memory available, and uses vmscan to handle the aging and removal of pages. Our results, Figure 3.15 show that there is very little performance regression by the additional codepath.

The mean real slowdown with unoptimized code was 1.5% (0.9857), with the worst regression being 8.4% in streamcluster, and the biggest improvement being 3.6% in dedup. There was a greater slowdown in the kernel time, however we don't believe this is a great concern.

Given the added benefits of tiered memory (e.g. lower power consumption, persistence

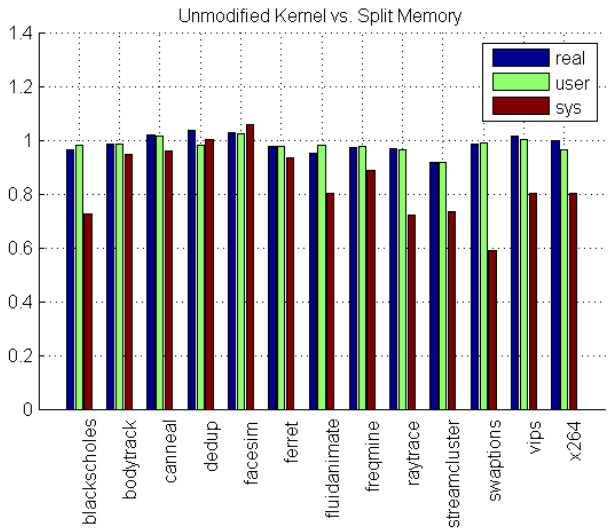


Figure 3.15: Speedup of an unmodified (stock) kernel to the Soft2LM kernel. We compare the real, user, and system times (described in 3.3.3) to determine the cost of longer codepaths. [49]

of data, etc.), there is only a minor regression in the performance of these memory-intensive benchmarks. Also, the amount of time spent in the kernel is very small compared the application. Averaged across all PARSEC benchmarks on a stock kernel, the CPU spends 153x more time executing the average application than in the kernel. In addition, compared to swapping as shown below, the performance regression is significantly smaller using Soft2LM. Finally, the current kernel modifications have not been optimized, and are currently very flexible for testing purposes.

3.3.5 Comparison to swapping to a RAMDISK

Instead of our method, a simpler use of SCM is to use it is a very fast block-storage device and swap to it. In this section we examine the benefits of using tiered-memory over swapping. Because using a traditional SSD/HDD-based swap device would be an unfair comparison with access time dominated by disk latency, we chose to create a swap device in RAM, thus only testing the actual overhead of the paging mechanism.

Mirroring the layout of tiered memory, we created a 5 GB swap file in DRAM. While

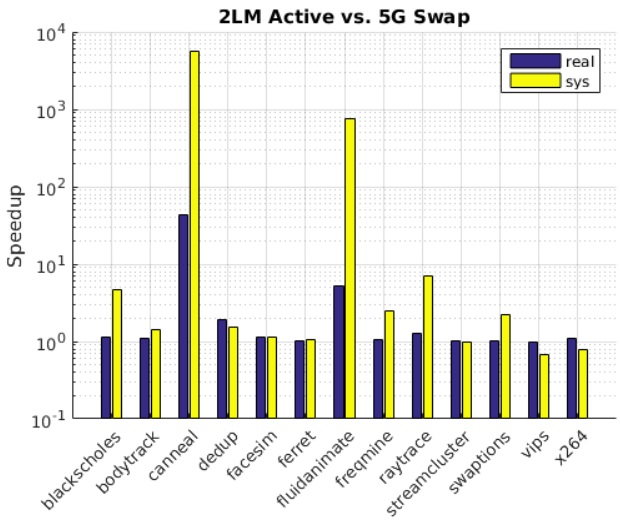


Figure 3.16: Data comparing the system with two active regions of memory, one 1GB and one 5GB versus a system with 1 GB RAM and 5 GB RAMDISK backed swap.[49]

we expected much better performance from tiered memory since the this code does not require page fault-triggered remapping, our data show that using the kernel's paging system drastically degrades performance, no matter how low the latency of the underlying device.

We ran the benchmarks on two separate swap configurations, one with 1 GB RAM and 5 GB of RAM disk-backed swap which matches the layout of the Soft2LM kernel, and one with 2 GB RAM and 4 GB RAMDISK-backed swap.

The results for a 5 GB swap device are shown as Figure 3.16. Note that because the speedup over the ramdisk is so large for some benchmarks, the speedup is displayed using a log y-axis. The mean and median speedup the real execution time of benchmarks using tiered-memory over a 5 GB Ramdisk are 4.7x and 1.116x respectively, while the kernel code showed a 1.53x median speedup. The performance of `canneal` was incredibly degraded when swapping, showing a 97.8% decrease in performance (43x speedup when using Soft2LM). `dedup` and `fluidanimate` both saw speedups of 1.5x and 5.2x respectively. Both of these benchmarks have rather large memory footprints as shown in Table 3.4 Both `vips` and `x264` showed a marked increase in time spent in the kernel (47% and 24% respectively), most likely due to the smaller data sets not needing to be swapped to disk, thus

eliminating the cost of the swapping portion of the kernel code. It is important to note that even though there was an increase in kernel latency in these two benchmarks, the overall performance of the benchmark was not notably changed. In the remainder of benchmarks, the time spent in the kernel was significantly decreased when using two regions of memory versus swapping, with a mean speedup of 1.5x. Only `vips` had a 0.8% overall slowdown due to the higher amount of time spent in kernel code.

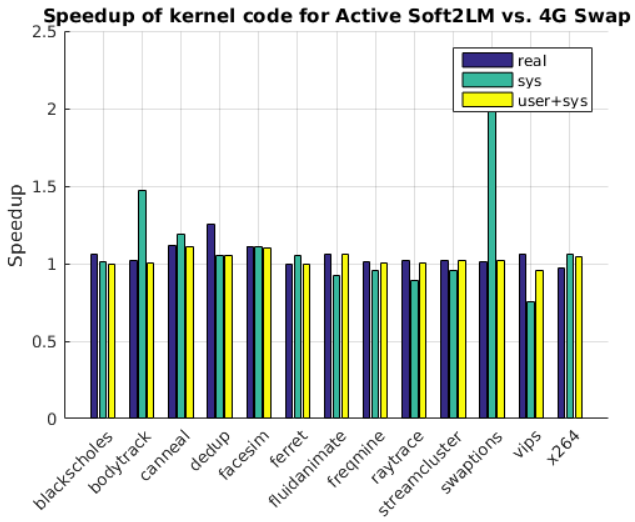


Figure 3.17: Data comparing the system with two active regions of memory, one 1GB and one 5GB versus a system with 2GB RAM and 4GB RAMDISK backed swap.[49]

Because a number of these benchmarks performed so badly with a 5 GB swap device, especially `canneal` and `fluidanimate`, we reran the benchmarks with 2 GB main memory and a 4 GB swapfile to test more modest swapping. These results are shown in Figure 3.17. We can see that with more DRAM, improvement using the tiered-memory manager is decreased, however there is still on average a 6.3% real time speedup over swapping, with a 13% speedup in kernel code.

3.3.6 Active Migration with Benchmark

While using two regions of memory shows an improvement over any amount of swapping, we also wanted to evaluate heavy migration on top of managing multiple regions of memory.

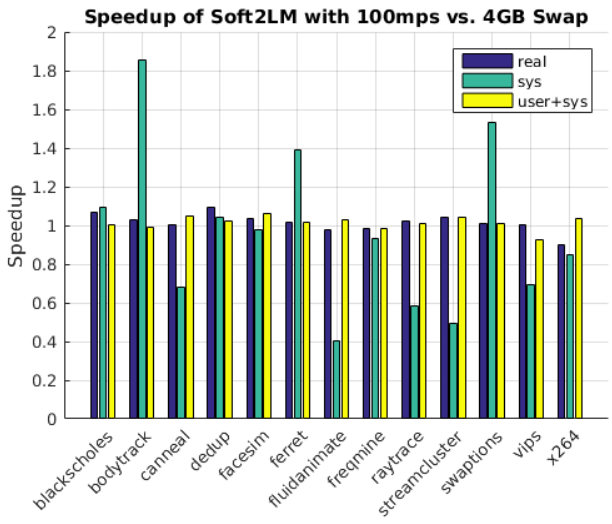


Figure 3.18: Data showing the speedup when migrating 100 pages per second versus swapping. [49]

Further tests were conducted while migrating 100 and 1000 pages every second, not due to memory pressure based upon a threshold, but to simply maximize the amount of migrations in order to determine the expected overhead of significant migrations. By moving pages from DRAM to SCM even when not under memory pressure, we allow new pages to be allocated in DRAM, thus implementing a basic system of page caching: new, hotter pages, are allocated first in DRAM, then as they cool are moved to SCM. The results of moving 100 pages per second are shown in Figure 3.18.

Migrating 100 pages per second is, given 4 KB pages, 400 KB of migrations per second. Compared to the less aggressive 4 GB swapping case, tiered migration made a slight average benchmark improvement of 1.8%, while exhibiting a 6.7% slowdown in kernel code execution. Some benchmarks such as `bodytrack`, `ferret`, and `swaptions` had significant improvements in kernel execution times.

Putting even more pressure on the migration system, we increased the number of pages to migrate per second to 1000. This very aggressive migration averaged 75,000 pages moved per benchmark run, which with 4 KB pages is 300 MB moved over the course of each benchmark. Comparing this migration number to the memory footprints of the benchmarks,

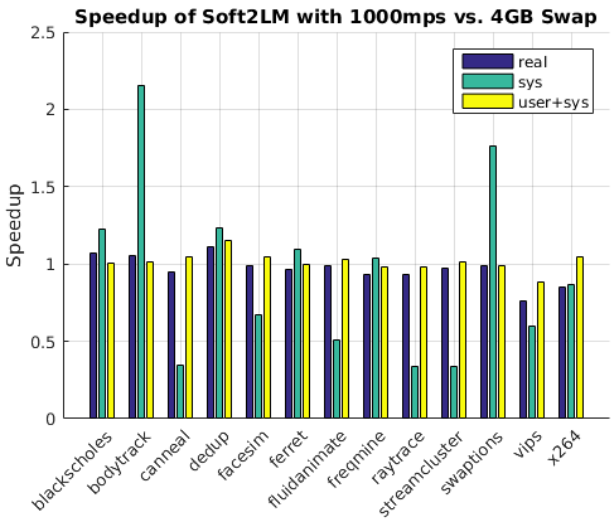


Figure 3.19: Data showing the speedup when migrating 1000 pages per second versus swapping. [49]

it's almost certain that a large number of these applications pages were moved from under it, with almost zero effects on the runtime performance of the programs. Because we are forcing migrations without being under memory pressure, not only are the inactive lists cleared out, but the active lists are heavily scanned as well.

The data comparing 1000 page migrations per second versus a 4 GB swapfile is shown as Figure 3.19. These results are very similar to those in the 100 migration case (Figure 3.18). Even by increasing the number of migrated pages by 10x, Soft2LM still improves over this limited swapping by an average of 0.7%, though spending nearly 6.3% more time in the kernel. The performance scales well, and when in some cases moving the entire benchmark's dataset, performance is still favorably comparable to 4GB RAMDISK-based swapping, to say nothing of its massively better performance against 5 GB swapping.

3.4 Conclusion

This chapter describes the enabling work that allows for system-state estimation, prediction, as well as mechanisms for power- and performance-aware memory management. System-state estimation is an essential part of metric selection by the Q-Learner in Chapter 6.

In order to properly train a reinforcement learner, of which the Q-Learner is a specific type, the learner must be able to identify the state in which it is currently and measure the instantaneous reward. Therefore, it is critical to define the system state and reward with metrics that best reflect the actual power and performance of the system. While not every metric is available on every platform and different metrics are useful in different scenarios, having a stable of known quality metrics such as stall-cycle ratio, instructions-per-second, and cache-hit ratio allow us to create better quality-of-service managers.

The developed methodologies and mechanisms support the software framework shown in Figure 1.1 for a variety of compute-aware applications; however, not all of these techniques are applied in the final experimental platform due to complexity and capabilities of that platform. For example, the neural network-based function approximation was not implemented in the final system for two reasons. For one, the added complexity made it more difficult to evaluate the performance of the Q-Learner. Second, the neural network was trained on a different system for a different application than our final test platform. While we believe it would be an effective addition, there are different metrics available in the ARM-based test platform therefore the design would have to be modified to take these into account. Similarly, Soft2LM was developed with the intention of using heterogeneous memory which is not available on the test platform. The API and settings that are available via Soft2LM can be added to the Q-Learner, however without real heterogeneous memory, it would only add complexity.

CHAPTER 4

SPECULATIVE THREADS

This chapter discussed the motivations for and the implementation of transient management strategies in a compute-aware manner. Undesirable transients can occur in digital controllers or filters whenever the controller or filter is switched abruptly from one configuration to another. In contrast, switching between standard applications in a multiprogrammed computing environment is done completely transparently by a scheduler [132]. Any special considerations such as critical sections of code must be handled through the use of algorithmic techniques such as mutexes, semaphores, or memory barriers [133]. These methods are well-studied and well-understood by computer scientists. However, switching between physical system controllers adds additional challenges not found in purely software systems, such as the management of transients.

This chapter discusses the implementation of transient management strategies in a compute-aware manner. A strategy embraced in this paper is to design the control system for digital filter software to be "compute-aware", that is, to monitor and take advantage of the dynamic reconfigurability of the processor when implementing digital filters or controllers. The result is a system that is designed with the flexibility of reconfiguration in both the controller or filter algorithm and in the processor.

This chapter lays out the case for transient management strategies in our system and examines experimental results for a filter bank and a switching motor controller. A portion of this chapter was published as [134].

4.1 Introduction

There has been much work done in the area of transient management in reconfigurable digital filter and control systems [84, 85, 86] Most of these papers consider the case when

the structure of the filter or controller remains constant, only the parameters change.

However, in the domain of digital filters and controls, drastic changes can occur due to the architecture of individual algorithms used. Namely, in situations where computational resources change sporadically, filters and controllers switch between algorithms of very different complexity. Transient management then becomes even more important, especially in those cases when switching from a low-order to high-order filter, since the initial conditions need to be chosen appropriately.

One approach to transient management in switched systems is termed "bumpless transfer," which is a transient management strategy that aims to maintain a smooth output response in the presence of abrupt changes in the control or filter implementation; see, for example [87], [88], and [89]. In essence, the bumpless transfer technique uses various methods to compute the appropriate initial conditions for the new algorithm that would match the output of the old algorithms over some time interval. This eliminates "bumps" in the output, but comes at the expense of computational effort, which grows higher as the filter order increases. This can result in delays and/or increased latency in the switching time. This paper proposes an alternate approach that is more amenable to real time applications.

A methodology called "speculative threads" can be extended for improved implementation of bumpless transfer. Speculative threads are traditionally used in computing applications and refer to a method where serial tasks that may have data dependencies are computed speculatively based upon predicted data [135, 136, 137, 138]. If the data were predicted correctly, then the work done by the thread is committed, and execution is accelerated. If, on the other hand, the data were not predicted correctly, then the work is discarded and the execution proceeds as it would have without the speculation. In this paper, the meaning of "speculative threads" is expanded to describe a technique used to decrease latency in a processor by predicting when a thread might be needed and spawn it ahead of time in order to reduce the warm-up time.

Speculative threads can be used as a means of implementing a predictive bumpless

transfer method to reduce transients when switching dynamic algorithms. A computationally simple predication algorithm is developed to determine if a switch is imminent, and then it automatically spawns a thread that is used to perform transient mitigation strategies. At the same time, the computational resources are automatically increased so that there is no loss of performance on the running algorithm such as added latency.

The speculative thread development was combined with a means to manage the processor resources and the dynamic algorithm in a coordinated manner in order to come up with an overall compute-aware transient management strategy. In particular, CPU cores are idled or turned on by the digital filter application as computational load changes in an effort to maintain a constant latency (or time delay) of the main controller or digital filter algorithm. One solution that is possible with this general framework is to turn on an idle core or speed up a processor in order to compute the initial conditions for the bumpless transfer. While improving the transient performance, this method still results in some delay in the switching time of the filter. Another solution, based on the speculative thread concept, is to predict when a switch is imminent and then to turn on the idle core to run the new filter in parallel with the old one so that the switching transients die out sufficiently by the time that the new filter is switched in. The prediction algorithm is very simple and has very little computational load.

The general framework for using speculative threads for transient management methodology for switching dynamic algorithms is described in Section 4.2 while the practical means of implementing the method are described in Section 4.3. Experimental results are given in Section 4.4 for a digital filter.

4.2 Speculative Threads for Transient Management

As mentioned in the background section, improper transient management when a digital filter or controller reconfigures can have a dominant effect on the system performance [80]. A method that can be used to reduce induced transients is to run the new filter off-line

and in parallel with the old one prior to the reconfiguration; that way, the new filter is in steady-state when it is switched on. There are two problems that must be overcome for this to be a viable strategy. First, it has a significant computational overhead, requiring two filters to be implemented concurrently. Second, waiting until the new filter reaches steady-state can cause a significant delay in the switching time, resulting in degraded performance. This chapter shows how to solve these problems by developing a prediction scheme that anticipates when a switch is likely to happen, and launches a speculative thread to run the new filter if sufficient computational resources are available at that time.

A simple, low-overhead predictor can be developed by modifying the existing switching condition calculations to include a component that determines a prediction if the switching boundary will be reached within a preset number of time steps. Then the algorithm is switched when it reaches that boundary with no delay, which may improve the response over other bumpless transfer methods that do incur a delay. The contributions in this area apply to any reconfigurable control system or digital filter. As the numbers of cores increases, using unutilized cores for speculative threads can reduce the performance penalties to the primary program and reduce the latency of switching the algorithm.

To describe the methodology, consider a digital filter with two possible configurations, Filter 1, $f_1(x, u)$, and Filter 2, $f_2(x, u)$, where x represents the states of the filter and u represents the filter inputs. This system has a switching condition, $g(x)$, that mandates when the filter switches from one configuration to another:

$$y = \begin{cases} f_1(x, u) & g(x) > 0 \\ f_2(x, u) & g(x) < 0 \end{cases} \tag{4.1}$$

where $g(x)$ is the switching function.

A state machine shown in Figure 4.1 can be used to represent the state transitions for the simple switching implementation where no transient management strategy is used, that is, where the initial conditions for the new filter are chosen statically and *a priori*.

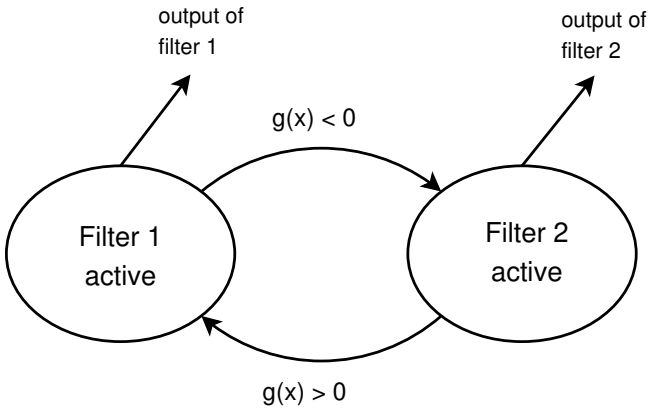


Figure 4.1: State machine for a reconfigurable controller with no transient management strategy. [134]

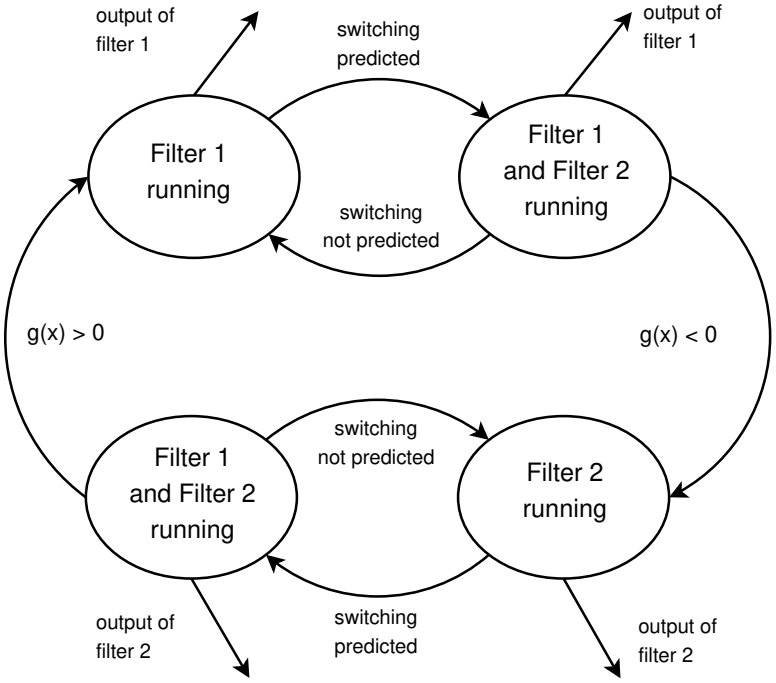


Figure 4.2: Example of a state diagram depicting two filter states and two prediction states. [134]

To implement speculative threads, the state machine is enlarged with secondary switching conditions as shown in Figure 4.2. If it is predicted that the switching surface will be breached in a small amount of time, then a speculative thread will be spawned to start implementing the secondary filter. That filter will become the primary filter when the switching surface is actually breached.

To determine the predication algorithm, a linear approximation is used to predict $g(x[n+m])$, m steps into the future. For example, suppose $g(x[n]) < 0$ so that Filter 2 is running. The linear approximation for the value of the switching function m time steps in the future, $\tilde{g}(x[n+m])$ is given as

$$\tilde{g} = g(x[n]) + (g(x[n]) - g(x[n-1]) \cdot m \tag{4.2}$$

where the finite difference $g(x[n] - g(x[n-1])$ is the instantaneous slope of the change in g. The approximation in equation 4.2 is used to predict if $g(x[n+m]) > 0$, that is, if the system would switch m time steps in the future. Thus, the condition in equation 4.2 results in a simple prediction algorithm that is used to spawn a speculative thread to start implementing Filter 1, $f_1(x, u)$, on a new CPU core if

$$g(x[n])(m+1) > g(x[n-1])m \tag{4.3}$$

The value of $N = m$ is a design parameter selected based on the time constant of the filter. If the controller or digital filter has a time constant of M steps, then N might be chosen as $N = M$ or $N = 2M$ in order for the transient to decay sufficiently before the switch is made. Note that $g(x[n])$ is being calculated anyway to determine when to switch controllers, so the incremental calculations needed for prediction are very minor. This design parameter might be called a prediction horizon since it relates to how far in advance of a switch that the prediction is made.

To summarize the secondary switching conditions that pertain to the prediction shown in

Figure 4.2: If filter 1 is running, that is, $g(x) > 0$ then switching is predicted if $g(x[n])(N + 1) < g(x[n-1]) \cdot N$. If filter 2 is running (i.e $g(x) < 0$) then switching is predicted if $g(x[n](N+1) > g(x[n-1]) \cdot N$.

4.3 Generic Implementation Framework

The goal of this research is to create a portable library for speculative multi-threading that can be used on any symmetric multi-programmable system running Linux. Our library was developed entirely in C, with the intent of making it fast, modifiable, and portable across any OS that can use POSIX threads (pthreads). By leveraging pthreads we were able to build speculative threads on a robust base, allowing other researchers and software engineers to easily use and modify the framework.

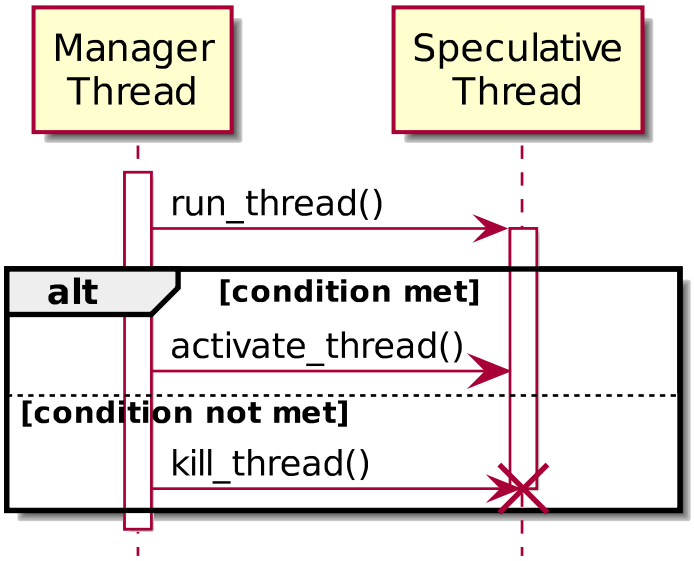


Figure 4.3: UML sequencing diagram for a generic speculative thread. [134]

The software implementation using speculative threads can be illustrated using a Unified Modeling Language (UML) sequencing diagram. UML is a graphical representation for designing and analyzing software systems [139]. The sequencing diagram is especially useful for depicting the interaction of multiple processes (or threads) including data dependencies, external events and inputs, and the sequencing of each process. Figure 4.3 shows a

sequencing diagram for a typical speculative thread used in general non real-time applications. The diagram shows two threads, a manager that controls the other threads, including the speculative thread. The threads are shown being created at the top of the diagram, then time runs downward showing the sequence of events such as when each thread is running and what events cause a change in the threads. The thin rectangular blocks below each thread indicate when that particular thread is running; the manager thread is always running. The speculative thread is spawned when it is predicted it might be needed. In the figure, this is indicted by the `run_thread()` command. The box labeled ”alt” shows the alternate paths that a branch (or if-then) decision might take; one branch is above the dashed line and the other is below it. For the first branch, the condition is met that the speculative branch is no longer speculative, that is, it is needed in the operation and an `activate_thread()` command is used. The other branch is that the condition is not met for that speculative branch to continue, so it is terminated with the `kill_thread()` command.

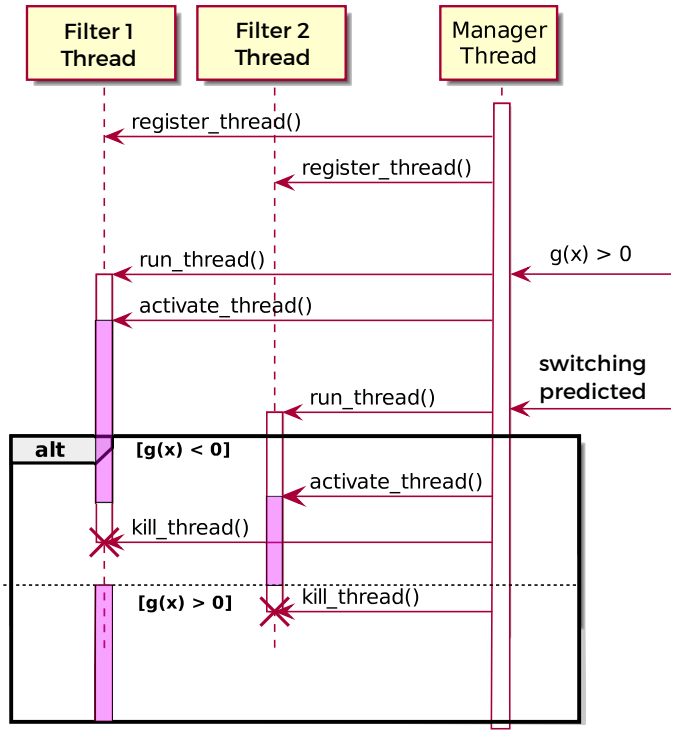


Figure 4.4: UML sequencing diagram for a speculative threads to implement transient management. [134]

Figure 4.4 shows a UML sequencing diagram for the speculative threads that implement the digital filter system transient mitigation state diagram in Figure 4.2. There are three threads that can run simultaneously: the Manager thread, Filter 1 (computes $f_1(x, u)$), and Filter 2 (computes $f_2(x, u)$). The Manager thread not only computes $g(x)$ and the switching conditions for the prediction, it controls when to run the other threads and when to reduce or increase computational resources. The computational resources are increased when both threads are running so that the execution time does not change for the primary filter, the one that is outputting the filter response. Resources are controlled by changing the frequency of the processor or by starting up or shutting down additional cores on a multi-core processor.

The diagram in Figure 4.4 can be explained by describing the various parts. Again, the sequence of events goes downward. In the initial state, Filter 1 is running since $g(x) > 0$. When the prediction is made, the Filter 2 thread is started with the `run_thread()` command. Both threads continue until a branch is reached where either the primary switching condition is met, $g(x) < 0$ indicating that the Filter 1 thread should be terminated, or that the prediction is no longer valid and that the primary switching condition has not been met so that the Filter 2 thread should be terminated. In the figure, the "alt" block shows the decision logic between two possible ways to exit the prediction state; these alternative branches are both shown but are separated by a dashed line. The primary switching and prediction switching conditions used to change states in Figure 4.2 are shown next to the Manager thread in Figure 4.4. The corresponding actions are to run or stop threads and to change the computer resources.

4.4 Application to Reconfigurable Filter Banks

The speculative thread method to reduce transients was tested on a filter bank consisting of two filters: a third-order and a second-order Chebyshev low-pass filters. The filter bank and the speculative thread framework were implemented on a ODROID-XU4 single-board computer, which contains ARMs big.LITTLE heterogeneous processor containing

eight cores (4 Cortex-A15 and 4 Cortex-A7). The big.LITTLE architecture is an attractive platform because of its mainline Linux support for scheduling threads on both high- and low-power cores contained on the same die [140].

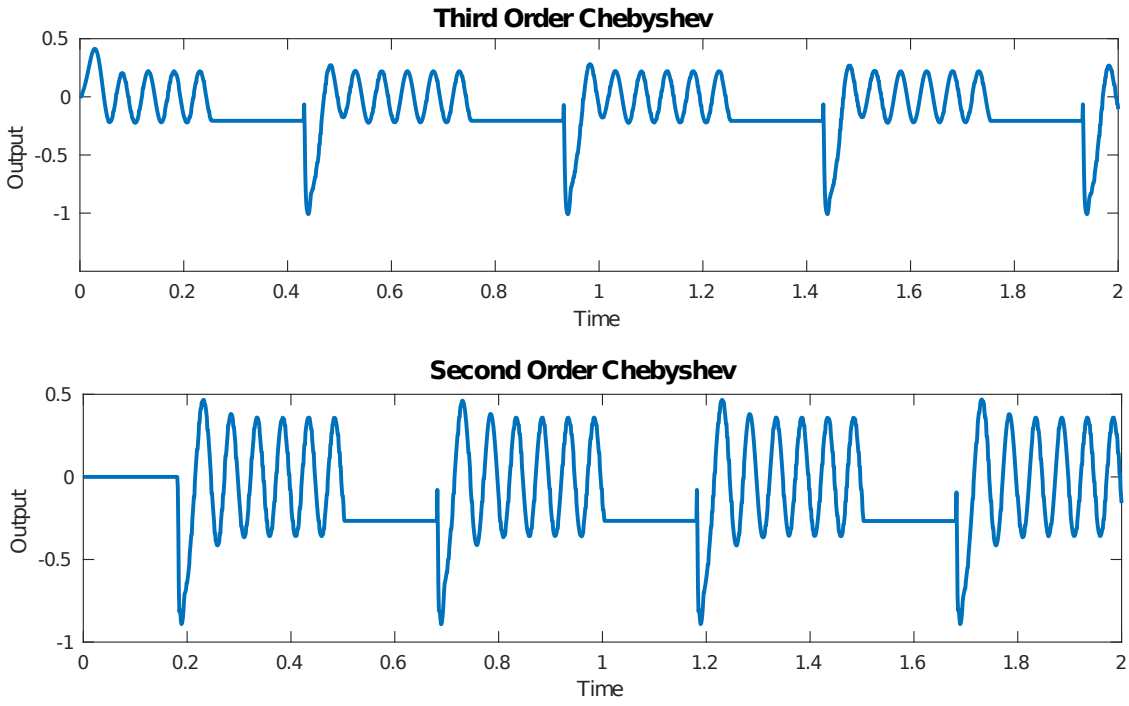


Figure 4.5: A newly activated filter when enabled with zeroed states demonstrates large transients before settling.

The two filters were run independently in a sampled data manner on the ODROID. Figure 4.5 shows their responses to pulsed sinusoids with zeroed states during the off times. These plots are meant to show the transients of both the 2nd-order and 3rd order filters that would appear during a cold switching of a filter bank between the two filters where the states were initialized to zero. A decision variable is needed to determine which filter to use during a hybrid operation. This decision variable might represent an external sensor signal or an internal signal related to the available computational resources. For example, when resources are high then a more computationally intense filter, such as the higher-order filter, might be used. When resources are scarce, then a less costly low-order filter might be used. For repeatability, the decision variable was defined for this experiment as a low-frequency

sinusoid representing a time-varying computational load that causes a sinusoidal variation in the computational resources $\frac{1}{2}(1 + sin(t))$. When the simulated load was less than 50% (ie, the sinusoidal value was less than 0.5), the high-order filter was run, and when the load was greater than 50%, the low order filter was run. The input to the filter was a higher frequency sine wave.

4.4.1 Speculative Threading

We examined three different scenarios of switching between the two filters, where they all used the same decision logic based on a sinusoidally varying load described above. In the first scenario, the benchmark for comparison, both filters were run constantly and multiplexed to a single output according to decision logic. There are no transients due to initializing the filters, but then the computational resources are doubled. In the second case, the filters were switched cold, that is, initialized at each switch with zeroed states and no time for the transients to decay. In the final scenario, the speculative thread method was used. The filter threads were launched when the computational load was predicted to pass the 50% threshold. Comparison of each switching case to the benchmark always-on filter bank case shows significant improvement as the amount of warm-up time increases.

The experimental results are shown in Figure 4.6 for cold switching and for two speculative thread cases. The vertical dashed lines indicate the times when the filters are switched from 2nd to 3rd order (or vice versa). The cold switching case switches the filter by setting the initial conditions for the new filters to be zero. The speculative thread cases spawn the new filter thread 30 or 100 samples prior to the predicted switching time and then switch to the new thread when the switching condition is actually reached. An indication of the time spent during speculation execution (when both filter algorithms are being computed) is shown at the bottom of the speculative thread plots. The benchmark case, with no switching transients, would show nearly pure sinusoids from about 0.1 seconds onward, with only a change in amplitude and phase due to the switching of filters. Of the three cases shown in the

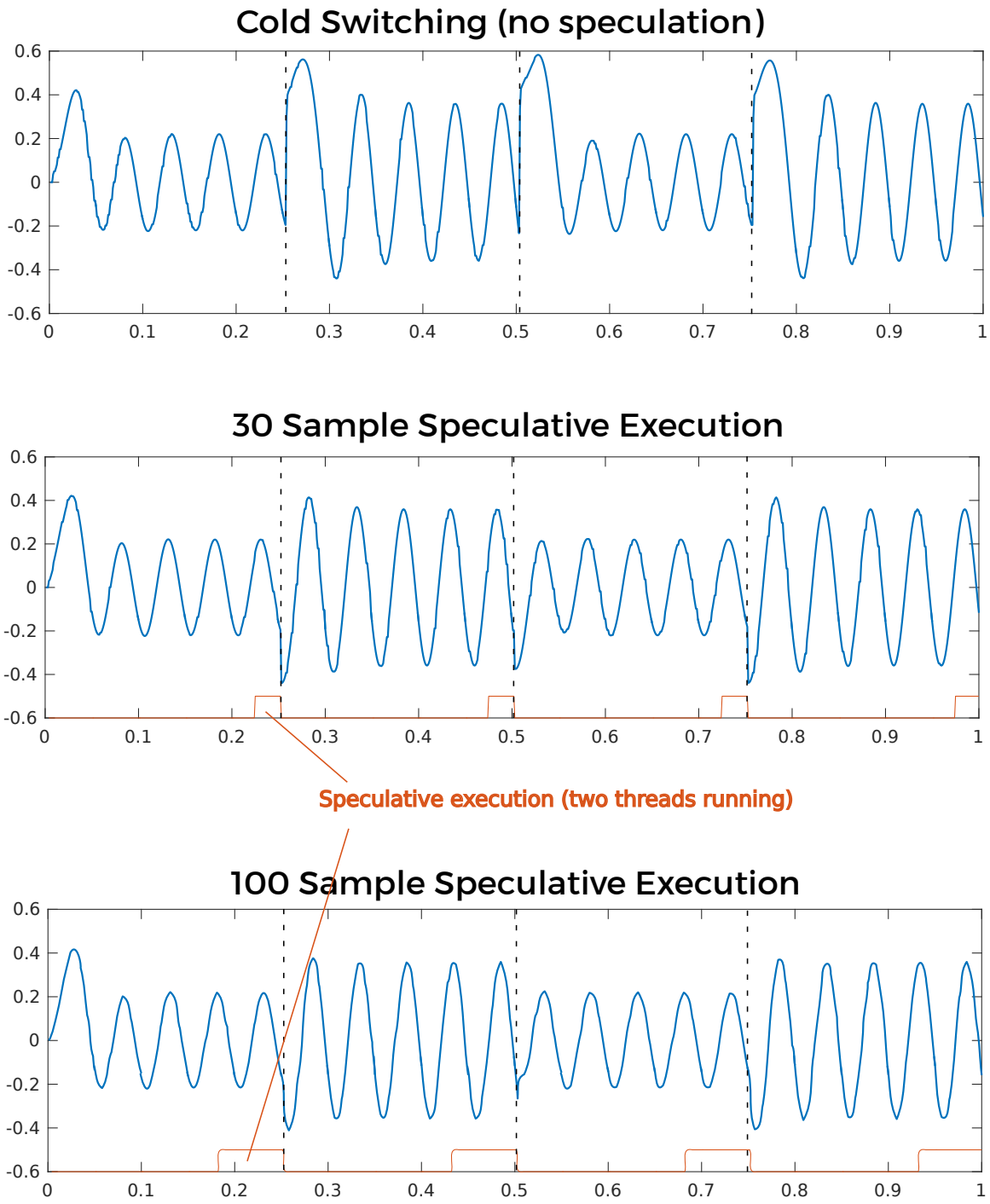


Figure 4.6: Experimental results of the output of the filters versus time for cold switching and for two cases using speculative threads, with 30 sample and 100 sample prediction horizons. [134]

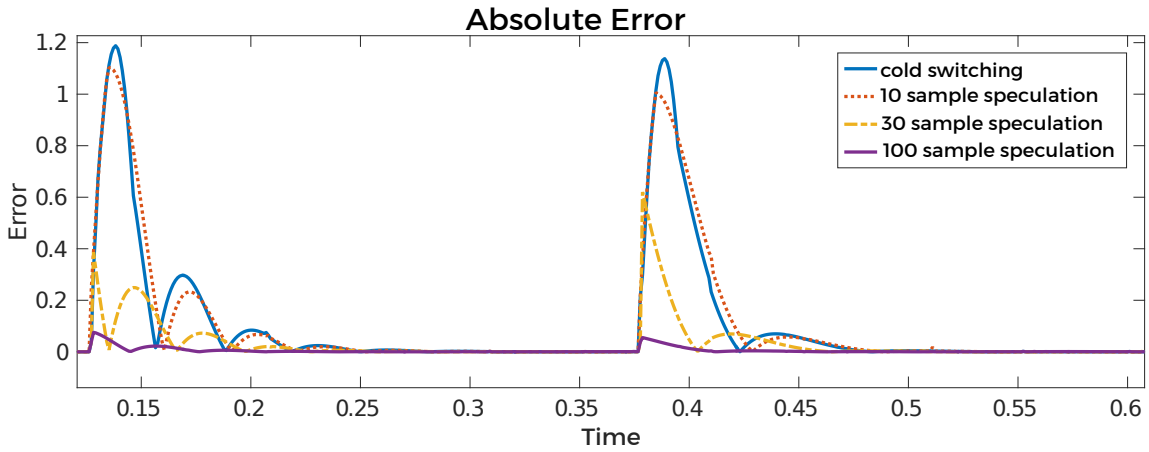


Figure 4.7: Comparison of each switching case to the benchmark always-on filter bank case. [134]

figure, the 100-sample speculative thread case is closest to the ideal benchmark case. The time constants for both of these filters are approximately 30 samples. Thus, the 30-sample prediction case allows the filters to run for one-time constant prior to switching, while the 100-sample prediction case allows the transients to decay for approximately three time constants prior to switching in the filters.

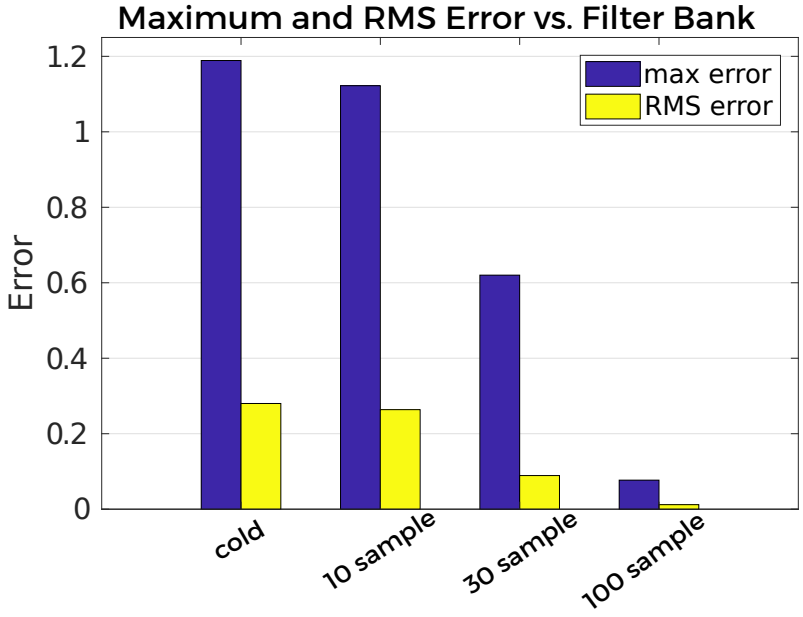


Figure 4.8: The maximum and RMS error for the different threaded experiments.

For a quantitative examination of the technique, Figure 4.7 plots the error magnitude, that

is the difference between the speculative and cold switching scenarios and the benchmark case. The performance of speculatively launched filters improves significantly over the cold switched variant, especially as the length of the warm-up period increases.

In addition, Figure 4.8 shows the maximum and RMS error. Cold switching has a maximum error of 1.189 and an RMS error of 0.2801. Adding 10-cycle speculation only reduces the maximum and RMS error by 5.63% and 5.82% respectively. A 30-sample speculation improves maximum and RMS error by 47.86% and 66.22%. With a 30-sample speculation, two filters run simultaneously only approximately 6% (60 of 1000 samples) of the total execution time but it shows a tremendous improvement. If we extend speculation to 100 samples (approximately 20% execution overlap), we obtained a 93.52% reduction in maximum error and 95.72% decrease in RMS error.

4.4.2 Introducting Hysterisis

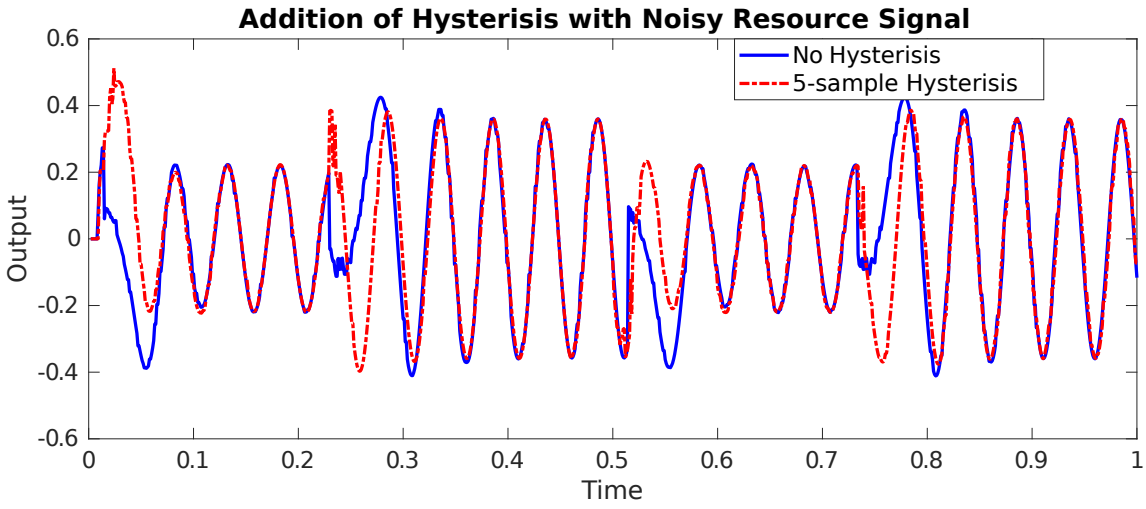


Figure 4.9: Influence of the hysteresis on switching transients with a noisy resource/switching signal.[134]

The above scenarios were run with a resource/switching signal that is a pure sine wave, while a noisy or irregular switching signal offers different challenges. For example, a noisy signal might oscillate across a switching surface, that is, trigger the switching condition repeatedly in a short amount of time. In switched systems, this is known as chattering. In

this situation, the prediction condition as well as the switching condition might chatter. In terms of speculative threads, chattering might cause the speculative thread to be alternatively killed and then reactivated repeatedly. When it is reactivated, the filter in the speculative thread is set to zero initial conditions. To alleviate this chattering, we introduce a time-based hysteresis to stop the premature termination of a speculative thread by keeping a deactivated thread alive for a specified amount of time. Experimental results suggest a hysteresis delay of 10% of the speculation period, that is, the value of N, is a good rule of thumb.

Figure 4.9 shows an example of the problems introduced by a noisy switching signal. Gaussian noise ($\mu = 0.1$ and $\sigma = 0.5$) was added to the switching sinusoidal signal. Without the hysteresis mechanism, the speculative filter threads are killed immediately when the prediction is no longer valid. As seen in Figure 4.9, by adding time-based hysteresis, the transients are mitigated even in the presence of a noisy switching signal.

4.5 Conclusions and Future Work

Speculative threads is a promising transient management implementation method that employs reconfigurable computing mechanisms available on many embedded system platforms. The speculative thread framework developed in this paper handles transient mitigation for switching between dynamic algorithms such as digital filters or digital controllers. The prediction algorithm that triggers a speculative thread to be started is very simple and has little computational overhead. The fact that the algorithm is predictive means that the computations are done prior to the switching time resulting in smaller latency in the switching time compared to traditional transient management strategies that are implemented after the switch condition is met. Also, the compute-aware nature of the developed framework allows for the control of the processor resources such that the processor speed may be increased or idle cores started in order to run the speculative thread in parallel with no loss in resources or performance for the active filter thread.

Experimental results run on an ODROID platform demonstrates the feasibility and

benefits of the speculative thread framework. A design parameter, N, called the prediction horizon was introduced that dictates when a speculative thread is triggered to start. The larger the value of N, the longer the the speculative thread runs prior to a switch. Correspondingly, the larger the value of N, the smaller the switching transients in the experimental results. A hysteresis mechanism introduced to avoid chattering keeps speculative threads alive briefly after the prediction of an impending switch ends in case the prediction is retriggered in a short period. This hysteresis method improves the performance of the speculative threads in environments when the switching signal is noisy.

Speculative threads were demonstrated on linear digital filters, but it is equally applicable to other dynamic algorithms.

CHAPTER 5

A SITUATION-AWARE RECURISVE QOS GOVERNOR

At the core of this research is a compute-aware framework of power management allowing bidirectional guidance between the hardware layer of the computing system and the application layer of the physical system controller. In other words, the application layer is aware of its hardware-defined power constraints and can adjust its algorithms accordingly. In addition, the application is able to request additional computing resources as needed.

The motivation for this framework as well as a high-level view of the architecture is described in Chapter 1. The high-level view is shown in Figure 1.1 and will be expanded upon in more detail below. The software framework is a quality-of-service manager (QoSM), located above hardware and OS and below the compute aware application (CAA).

This work can be found in [141] and is expanded upon below.

5.1 Introduction

5.1.1 Power Management Strategies

Power management strategies begin at the chip level and continue in peripheral hardware, firmware, operating systems, and application and algorithmic features. Multiple power states were introduced and later standardized by the Advanced Configuration and Power Interface (ACPI) in 1996 [14] in order to give the operating system the ability to manage the power usage at run time and with a granularity more fine than just an on/off decision on components. In particular, there is a trade-off between high performance states of operation and low power states. For example, many people are familiar with the power management strategies in a laptop or mobile phone. In the high performance state, the processor and peripherals are operating at their highest clock speed and voltage levels. At the lowest

performance state, the processor and peripherals go to sleep quickly due to inactivity and run at slower speeds with lower voltage levels.

The most direct way to increase the performance of a CPU is to increase the clock frequency, thereby increasing the number of instructions completed in a given amount of time. The CPU power is approximated in [13] as

$$P \propto \alpha V^2 f \tag{5.1}$$

where α is the activity of the processor, V is the voltage, and f is the clock frequency. An increase in frequency is directly proportional to an increase in power consumption. To match the processing power of a CPU to the workload, modern CPUs can use dynamic voltage and frequency scaling (DVFS) to find a suitable power-performance state [13, 27].

From the application's side, algorithms that have options to reduce performance often have the potential of reducing power. Therefore, we would like to leverage these types of algorithms with the goal of reducing power. Examples of this include anytime algorithms and imprecise computing [142, 79]. All of these solutions, however, exist at very specific layers of the process architecture.

The purpose of this chapter is to describe a compute-aware framework allowing bidirectional guidance between the hardware layer of the computing system and the application layer of the physical system controller. This means that the application layer is aware of its hardware-defined power constraints and can adjust its algorithms accordingly, and is able to adjust computing resources as needed. The goal is to build this software framework with a manager layer that exists on top of existing power management layers and below the physical control system application layer.

Section 1.2 explains the motivation for and design of the architecture of the application-aware framework. Section 5.2 describes our experimental platform in detail, with Section 5.2.1 explaining the hardware and Section 5.2.2 detailing the control algorithms. Experimental results are given in Section 5.3 that demonstrate a situation-aware governor running

on a mobile robot as it does trajectory planning with obstacle avoidance.

5.1.2 Middleware for QoS Management

The idea of software abstraction layers or middleware for application QoS have been explored by a number of researchers. One of the early works by Li *et al* was an attempt to balance the objectives of system and application in a distributed video system with the introduction of an application-aware QoS middleware [69]. Zhang *et al* develop a system, ControlWare, for QoS management in distributed real-time systems and introduce *convergence guarantees* that lie between hard and probabilistic guarantees [7]. Their use of feedback-control theory to manage resources based upon a QoS is similar to our proposed system in concept but differs in application and scope. ControlWare is focused on distributed systems and requires coordination entities not needed for a single-application system. CoAdapt allows for dynamic coordination of accuracy-aware and power-aware systems [70].

Imes *et al* have worked on hardware and software agnostic frameworks for power management. POET, their C-based framework, minimizes energy consumption while maintaining soft real-time constraints of commonly used benchmarks [71]. They expanded upon POET to create Bard, a framework that allows for changing between power and performance constraints at runtime [72]. This work is the most closely related not only due to their framework design but also their use of the ODROID-XU3 as their test platform.

Our work diverges in some important ways. First, the performance target is not a computational metric, but rather performance of the physical system itself. In the experimental example presented in this paper, the performance is how closely a mobile robot can track a desired path, but this metric is not integral to the framework. Any performance metric of the physical system could be used. Second, our example test platform, a multithreaded autonomous robot, has more complex and strict requirements for performance. This robot application, while making extensive use of sampled calculations, displays both periodic

and aperiodic behavior. Third, in addition to meeting application timing constraints, the computing system itself informs the application of excess or limited resources, giving it the ability to preemptively prepare by changing algorithms or operating modes. In the opposite direction, the application can request the hardware, via the framework, to increase or decrease the resources when it anticipates a changing operating mode.

5.1.3 Design of the Software Framework

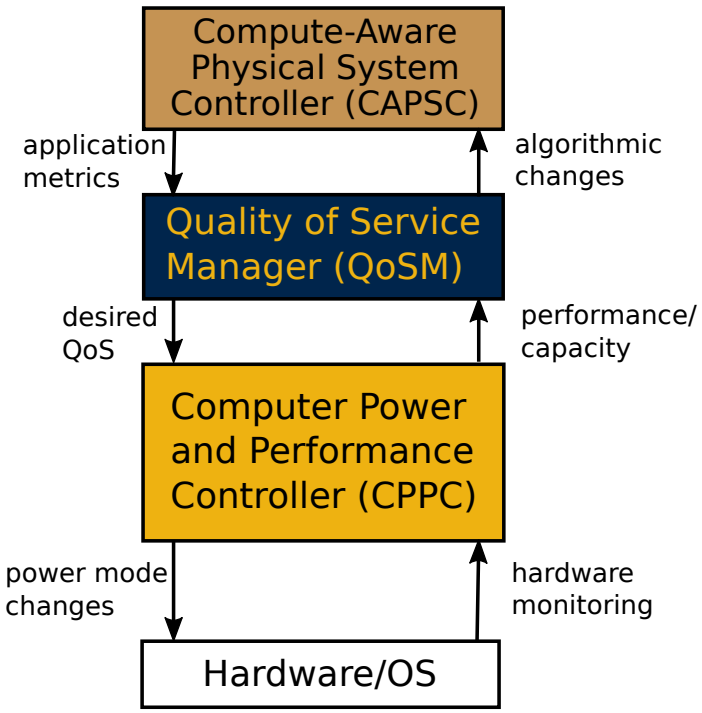


Figure 5.1: A high level view of the software framework architecture.[141]

To satisfy the design considerations and meet the architecture requirements described generally in Chapter 1, we developed layered architecture shown in Figure 5.1. It has three basic components that are the core of the work: *compute-aware physical system controllers*, *computer power and performance controller*, and a *quality of service manager*.

Compute-Aware Physical System Controllers (CAPSCs) are standard software controllers reformulated into anytime algorithms, which can be stopped or reconfigured at any time due to sudden limitations in the computer resources. Multiple CAPSCs can be

composed into a single application, with each physical system controller communicating it's metrics to the QoS Manager (QoSM). The QoSM receives the monitored metrics (e.g. SNR, control system error signals) and converts this into a QoS that can be passed to the computer power and performance controller (CPPC). The CPPC takes the desired QoS and converts it into available physical power/performance modes and relays them to the hardware/OS.

On the return side, the CPPC monitors the OS/hardware performance, identifying capacity excesses or shortages and relays this information to the QoSM. The QoSM informs the CAPSCs of possible algorithmic changes that can be made based upon the computing system. For example, if the computing system is set to the lowest power and performance modes and there is still excess capacity, the QoSM can inform the CAPSC to use higher performance algorithms (e.g. higher resolution image capture or deeper searches). On the other hand, if capacity is low and the CPU is struggling to keep up, the QoSM can notify the CAPSC of its limitations, allowing application performance to degrade gracefully as opposed to simply running out of resources.

Most importantly, these communication pathways allow both application (CAPSC) and hardware to make informed decisions about how to use available resources in the most efficient manner possible.

5.1.4 Situation-Aware Governor

We developed a situation aware (SA) governor to act as QoSM, which changes the CPU clock frequency in response to situations perceived by the physical control system application. There are both discrete and continuous mechanisms for determining the appropriate frequency (and power state) of the CPU. If the control system perceives that it is in a situation that requires high performance or that it is in a computationally intensive region (such as recalculating the entire ADA* path), it sets the CPU performance state to the highest power/performance mode $P0$.

In other circumstances, the SA governor attempts to reduce the power in a more continu-

ous manner by feeding back the performance error in the following algorithm:

$$Q[n] = K_p \cdot Q[n-1] + K_e \cdot e[n-1] \tag{5.2}$$

where $Q[n]$ is the quality-of-service at the current time, $K_p \in [0, 1]$ is a power-down coefficient that determines how aggressively the CPU's power is decreased, $Q[n-1]$ is the quality-of-service at the previous time, K_e is an error gain, and the previous physical system performance error is $e[n-1]$. Equation 5.2 represents a generic SA governor. To apply the algorithm to the mobile robot in the experiment, $Q[n]$ is mapped to discrete CPU P-states (voltage and frequency) and $e[n]$ is the path error of the robot from the ideal path.

The goal is to put a downward pressure on the frequency of the processor, reducing the power consumption until the reduction in performance causes an increase in the measured error, which then puts upwards pressure on the frequency, keeping the performance high enough to meet error targets. Different combinations of K_p and K_e were evaluated to validate the qualitative behavior of the governor as well as to determine the ranges of performance achieved and power consumed.

5.2 Experimental Platform

The compute-aware framework shown in Figure 5.1 was implemented on a mobile robot platform in order to demonstrate the feasibility of dynamically controlling the computing platform processes in response to situational awareness seen by the robot.

5.2.1 Hardware Platform

The initial test platform of the framework is a custom mobile robot built upon the DFRobot Cherokey 4WD. The computational unit is a hybrid, stacked, heterogeneous architecture consisting of two primary computational units. The lower-level CPU is a ATmega328/P-based Arduino variant, the DFRobot Arduino Romeo. This board contains the needed

hardware for motor controls as well as analog and digital GPIO pins which are necessary for collecting data from sensors. In addition, running a low-level controller on the Arduino allows for basic data collection and computations to be handled by a very low power microcontroller (20 mA @ 20 MHz). This board runs the low-level PID motor controllers and odometry and is connected via SPI to the upper-level CPU.

The ODROID XU4 is based upon the Samsung Exynos5422 Cortex-A15/A7 Octacore SoC and is among the most powerful single-board computers [143]. This ARM big.LITTLE heterogeneous multiprocessing architecture (HMP) allows for transparent thread scheduling between high-performance higher-power cores (Cortex-A15) and lower-performance lower-power cores (Cortex-A7) [144] and per-cluster DVFS [145]. The ODROID is running Ubuntu Linux with the 4.14.5-92 kernel which allows for the capture of metrics from the hardware performance monitoring units (PMUs).

The use of a full single-board computer as opposed to a simple microcontroller allows for more complex algorithms such as computer vision and high-resolution pathfinding. In addition, multicore processors allow us to use aggressive multithreading to control multiple subsystems independently. Functions are divided into multiple threads to allow for scheduling among the heterogeneous cores either using the OS's scheduler or by manually pinning the cores based upon power or performance needs.

The most computationally complex of the threads is our implementation in C of Anytime Dynamic A* (ADA*), a path planning algorithm that can replan during run time [146]. ADA* is a good candidate for testing this system for a number of reasons. First, it is anytime, so after a single iteration of path-finding, the robot has a path to its goal that monotonically approaches the ideal path with further processing. Second, it can update the path when encountering a new obstacle, allowing for a more realistic test of a dynamic environment with limited sensor range. Third, the complexity can be adjusted based upon the resolution of the map.

5.2.2 Control Algorithms

In autonomous robots, the calculation of the safe and feasible trajectory is known in general as *motion planning* [147]. Motion planning is often broken into related but different problems: path planning and trajectory planning [148].

Path-planning is concerned with determining a path in a given environment and safely traversing obstacles from a start point to an end point. These paths are time-independent and consist of a continuous curve [148] or a graph [149].

Trajectory planning is defined either as a time-parameterized function prescribing the configuration of the vehicle in time [147] or as taking the solution from the path planner and determining how the robot should move along the path [148]. In this work, we are using the second definition, an algorithm that takes the path generated by the path planner and calculates the desired wheel velocities to reach a given waypoint.

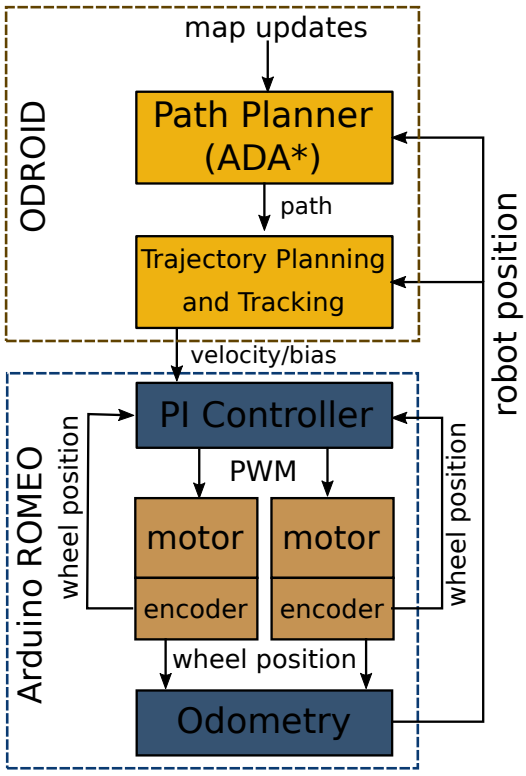


Figure 5.2: The interaction between the controllers used in the experimental plaform. [141]

Our control algorithms are shown in Figure 5.2. On the ODROID platform, the ADA*

[146] stores a map of the environment, represented by a grid, and computes the path. This map and associated matrices as well as mathematical functions needed for the ADA* algorithm were implemented using the GNU Scientific Library, a C library for scientific computing [150]. Once the first iteration of ADA* is completed, a feasible (but possibly not optimal) path is calculated, and the set of waypoints $P = \langle p_{start}, \ldots, p_{end} \rangle$ where each $p_n = (x_n, y_n)$ is passed to the trajectory planner . It determines its next waypoint by first finding the point on the path P closest to its current position with the addition of a optional lookahead parameter ℓ.

Given the robot's current position (x_c, y_c) and the path P, the trajectory planner calculates the Euclidean distance between the current point and each waypoint, saving the closest one $p_i = (x_i, y_i)$. It then iterates ℓ points ahead selecting $p_{i+\ell}$ as the next target.

The difference between the current position $p_c = (x_c, y_c)$ and the target or goal waypoint $p_{i+\ell} = p_g = (x_g, y_g)$ is given in equation 5.3.

$$\begin{aligned} \Delta x &= x_g - x_c \\ \Delta y &= y_g - y_c \end{aligned} \tag{5.3}$$

Using this distance, we can calculate the angle to the goal θ_g using the four-quadrant arc tangent

$$\theta_g = \arctan\left(\frac{\Delta y}{\Delta x}\right) \tag{5.4}$$

The error in heading is simply $\Delta\theta = \theta_g - \theta_c$. The trajectory planner then calculates the needed velocity and turning angle and passes these values to a differential steering PI controller [151] implemented in the Arduino platform.

The position of the robot is calculated using odometry from the motor encoders, and this position is made available to both the path planner (to prevent planning from the beginning)

as well as the trajectory planner (to determine the next needed maneuver).

5.2.3 Experimental Environment

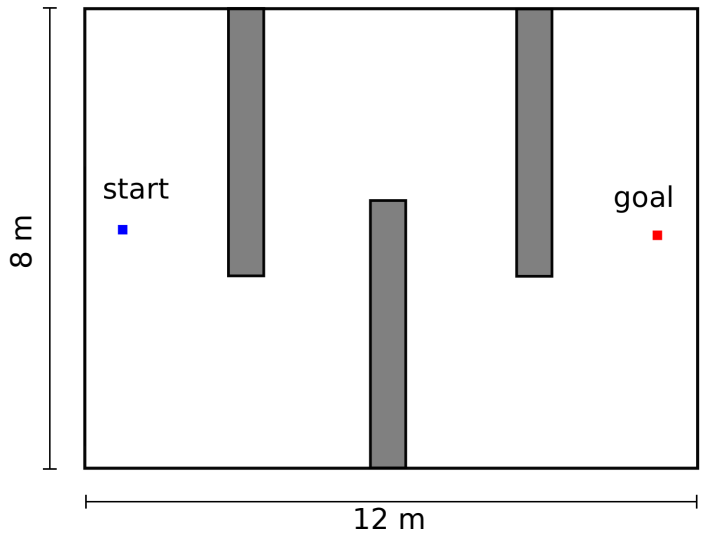


Figure 5.3: The testing area was 12 m by 8 m divided into 960,000 squares of 1 cm each. Three obstacles forced the path of the robot deterministically. [141]

The testing area for the robot is illustrated in Figure 5.3. It is 8 m x 12 m, subdivided into 960,000 1 cm squares. The start and goal are directly across from each other. There are three obstacles that force the robot's path to slalom. The robot begins on the left side of the map, as shown in Figure 5.3. It calculates an initial path, not seeing any obstacles, directly for the goal on the right side of the map. The robot has a "vision range" of 1 m (100 squares), so as it approaches the first obstacle, a portion of it becomes revealed. ADA* begins replanning the route, passing the new path along to the trajectory tracking controller when it is completed.

Current and performance metrics are monitored by another thread. There is an `ncurses`-based [152] monitor thread that shows the robot's position and current metrics for determining the robot's progress, Finally, we have a separate logging thread for writing data to a file.

5.3 Experimental Results

The experimental results are analyzed with respect to a set of performance metrics of the physical system as it traverses the path as well as a time-series evaluation of the computing system parameters. To evaluate the performance of our situation-aware governors, we compared their performance to that achieved with a static-low-power state setting, a static-high-power state setting, and the default Linux on-demand governor. While the DVFS settings in Linux are enumerated by a frequency value, in implementation, each discrete frequency setting has an associated discrete voltage: higher voltages with higher frequencies and lower voltages with lower frequencies. The on-demand governor scales the CPU frequency and voltage dynamically according to the load, measured by the amount of idle time the processor experiences [35]. The static power setting cases used in the experiments give upper and lower bounds on both performance and power. The static-high-power state set the cores at their maximum frequency, 1.6 GHz and 2.0 GHz on small and large cores, respectively. The static-low-power state is a frequency setting of 400 MHz on both large and small cores. This was the lowest setting in which the path following task was achieved. Lower settings resulted in the robot failing to reach the final goal or hitting an obstacle.

5.3.1 Performance Metrics for Path-Following Task

The performance is evaluated qualitatively and quantitatively by examining the actual path taken by the robot as well as a set of averaged metrics. Figure 5.4 plots experimental data representing sample paths of the robot taken during representative runs for the different governors, with the inset showing a close-up of the behavioral trends. The static-high-power state and the on-demand governor very closely match, and there is very little overshoot in their performance. Correspondingly, these cases resulted in the smallest time to complete the task. The results with the situation-aware governor, run with $K_e = 0.001$ and $K_p = 0.85$, show that there is a small latency at the start of a turn. This latency is due to the delay from

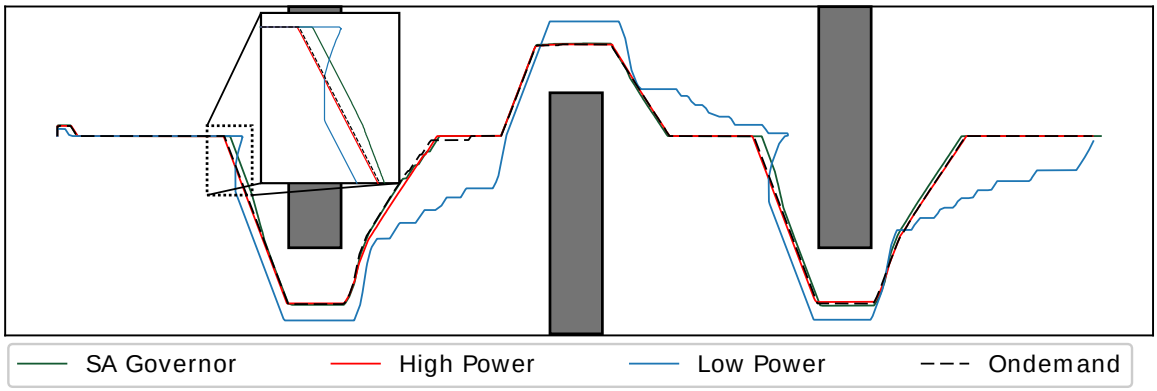


Figure 5.4: Experimental results showing the obstacles and paths taken by the robot when the computing platform is running under one of four cases: the static-low-power setting, the static-high-power setting, the situation-aware (SA) governor, and a the default on-demand governor. The inset shows a close-up of the paths when the obstacle is first sensed. [141]

the time that the robot senses the new obstacle to the time that the CPU ramps up to full performance in order to compute the new path using the computationally intensive ADA* algorithm. The static-low-power setting results in a long execution time to compute the new path, and thus the robot often overshoots or takes a non-ideal path. These trends are similar throughout the path of the robot.

There many different metrics of performance that can be used for a mobile robot including accuracy of path following, time to complete the task, energy consumption, or maximum power. Data was collected while the robot traversed the path shown in Figure 5.4.

Figure 5.5 shows RMS Path Error, average power consumption for the ODROID computer, and time to complete the task, all averaged over ten runs for each of the cases: the high and low static power settings, the on-demand governor, and five situation-aware governors with different K_p and K_e values. The RMS error in this case is defined as the deviation from the average path followed by the robot in the static high-power state. The three green, crosshatched bars in Figures 5.5 and 5.6 correspond to an error gain of $K_e = 0.001$ and different power-down coefficients, $K_p = \{0.80, 0.85, 0.90\}$, where lower values give more aggressive frequency reduction. The two brown starred and dotted bars examine the impact of changing the error gains. These bars correspond to settings of $K_p = 0.85$ and error gains

of $K_e = \{0.0005, 0.0025\}$, where the larger values place more weight on physical system performance. The static-low-power state setting results are shown by the blue slashed bar while the static-high-power state setting results are shown as the red back-slashed bar. The on-demand governor results are displayed as yellow circles.

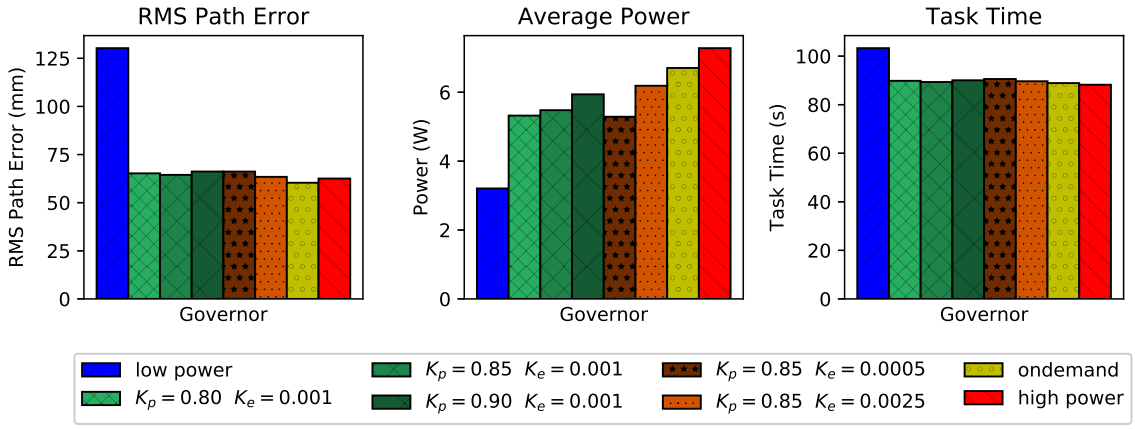


Figure 5.5: These graphs show the averages of the metrics collected during the experiments. [141]

The average path error is very high in the static-low-power state, owing to the extended length of time needed to complete a single iteration of the ADA* algorithm. In contrast, the situation-aware governors do very well, nearing the performance of the static-high-power state, with only 3.0% increase in RMS error. The best performing situation-aware governor ($K_e = 0.0025$) sees only a 1.3% increase in RMS error compared to the static-high-power setting, demonstrating the ability for system designers to prioritize either better performance or better power savings based upon the control gains of the situation-aware governor.

Next, looking at average power, there is a much larger variance between the governors. The static-low-power state consumes less than half the power than when running in the static-high-power state. The power savings of the on-demand governor over that of static-high-power state is 7.9% while even the least aggressive situation-aware governor ($(K_p, K_e) = (0.85, 0.0025)$ sees a 15.0% reduction in power from the high-power state. More power-conscious governors (lower K_p) reduce power by 26.9%.

The task time, that is, the time that it takes the robot to traverse the path from the

starting point to the goal, was relatively flat across most of the governors. The low-power governor takes about 14% longer than the other governors due to slower processing time, which then leads to higher latency and more path deviance in the entire control system. The situation-aware governors saw 1.3-1.8% longer task times versus 0.7% for the on-demand, when compared to the static-high-power case.

In general, a lower K_p and K_e reduce power consumption at the cost of slightly increased error and task time, whereas higher values give higher power consumption with better performance. These values could be tuned to the system designer's specifications or could be changed dynamically based upon power and performance targets.

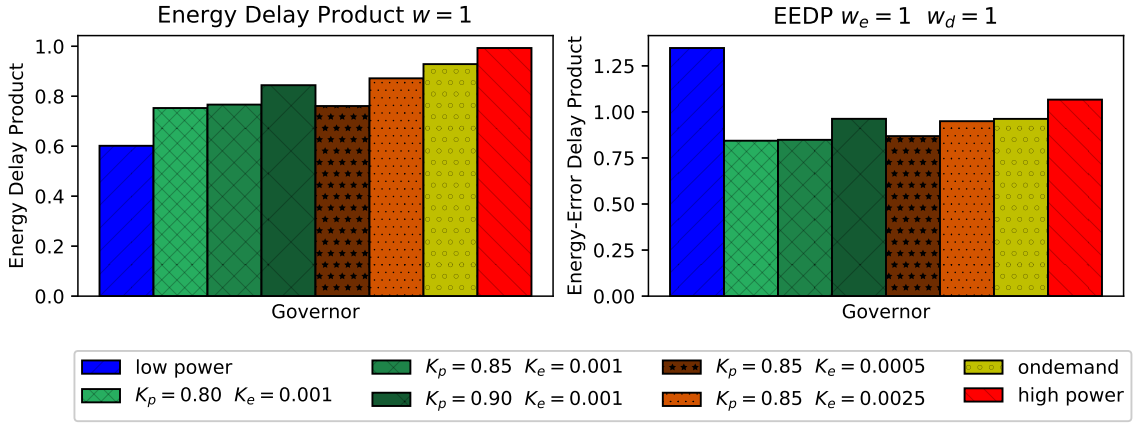


Figure 5.6: Using the measured performance, we derive two metrics: the energy delay product (EDP) and the Energy-Error Delay Product (EEDP).

In Figure 5.6, we look at the *energy delay product* as well as introduce a new metric, the *energy-error delay product*. Energy delay product is metric introduced by Horowitz *et al* that can be used to examine trade-offs between energy-saving techniques and performance [153] .

While initially used for circuit-level design, energy delay product has been expanded by the high-performance computing (HPC) community to allow more weighting on the delay [154, 155]. Thus, energy delay product can be defined as

$$\text{EDP} = E \cdot T^w \tag{5.5}$$

where E is the normalized energy, T is the normalized task time, and w is the weight placed on performance. In this metric, performance is defined by the task time. To be clear, in purely computing applications, the "task time" in equation 5.5 usually refers to execution time, which is the completion of the computational task. However, in our physical system, we refer to the "task time" as the time needed to traverse the entire course which takes into account the computational time, the quality of the path computed, and the time for the robot to complete the necessary motion.

In Figure 5.6, we see the energy-delay product for the experimental cases with $w = 1$. Here, we can clearly see that, because the time difference between governors is relatively small, any reduction in energy consumption results in a lower EDP, which is desirable in a purely computational system. If we wanted to place more emphasis on the time to complete the task, we would increase the value of w, weighing the performance of task completion more than the increased energy consumption.

However, the Energy Delay Product metric fails to capture all of the performance metric in this application. Control systems are often concerned with not only the time to complete the task T but also the physical system performance. Therefore, to encapsulate this added constraint we introduce an expanded metric, the energy-error delay product (EEDP)

$$\text{EEDP} = E \cdot \epsilon^{w_e} \cdot T^{w_d} \tag{5.6}$$

where E and T remain normalized energy and runtime as in Equation 5.5, ϵ is the normalized physical system error, w_ϵ is the weight placed on the physical system error, and w_d is the weight placed on delay.

When we include the average, normalized error creating the energy-error delay product (EEDP), we see the effects that slowing down the CPU has on the physical system performance. The more aggressive SA governors perform the best, reducing power consumption while not resulting in a large increase in error or delay. If we place a greater emphasis on minimizing the path error (i.e. by increasing K_e), we see a slight increase in the EEDP,

however the overall performance is still better than the static-high power setting.

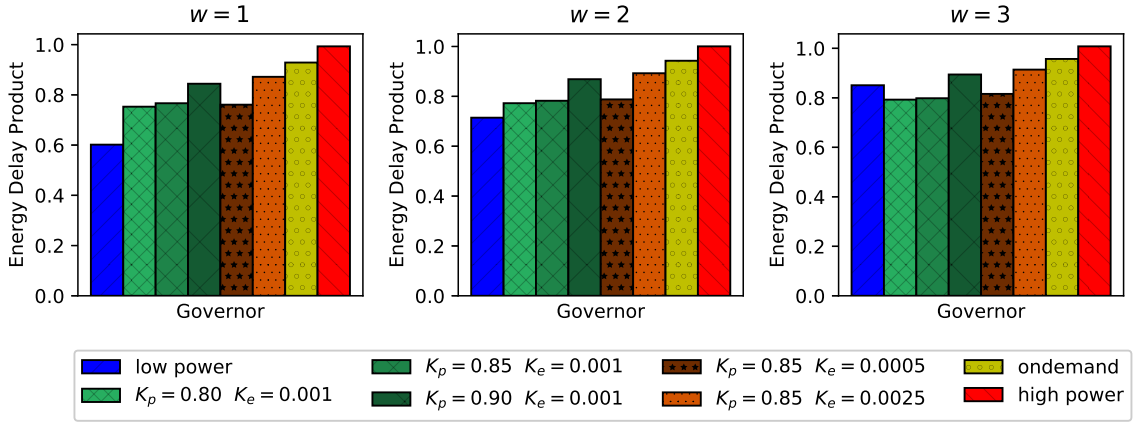


Figure 5.7: By varying the weight of w in the energy delay product (EDP), system designers can examine the impact of governor settings based upon the importance placed on the delay.

To better see the effect of w on the energy delay product, Figure 5.7 varies the weighting from 1 to 3. If a squared importance ($w = 2$) is placed on the delay, the situation aware governors are much closer to the low power state. If the weighting is increased to 3, the high K_p low K_e governors outperform the low power setting. This weighting depends entirely on the importance that the system designer places on traversing the course.

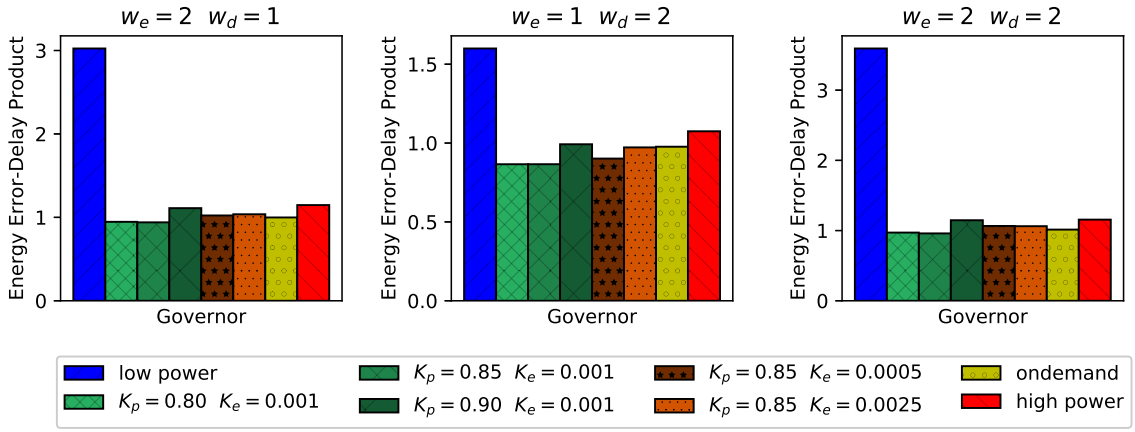


Figure 5.8: By varying the weight of w_e and w_d in the energy-error delay product (EEDP), system designers can examine the impact of governor settings based upon the importance placed on the delay and error.

Figure 5.8 shows the changes that increased w_e and w_d have on the energy-error delay product (EEDP). Because the error and delay are both higher in the low-power setting, the

EEDP in the heavier weighted metrics is much higher. However, what is interesting is that as we weight error higher (i.e. increased w_e), we see that the performance of the more error sensitive governors (i.e. higher K_e) improves. This is a validation of both correct behavior of the governors as well as the performance of the metric for evaluating the governors.

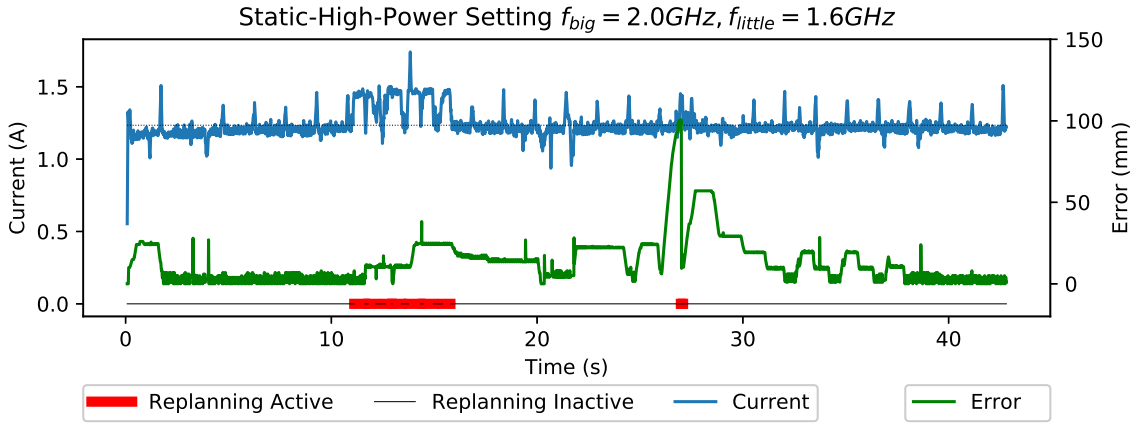
5.3.2 Time Series Evaluation

Next, we examine the computing system performance in order to see how it relates to the physical system performance. To make the time-series data easier to read, we simplified the course the robot had to traverse, placing only a single obstacle in its path.

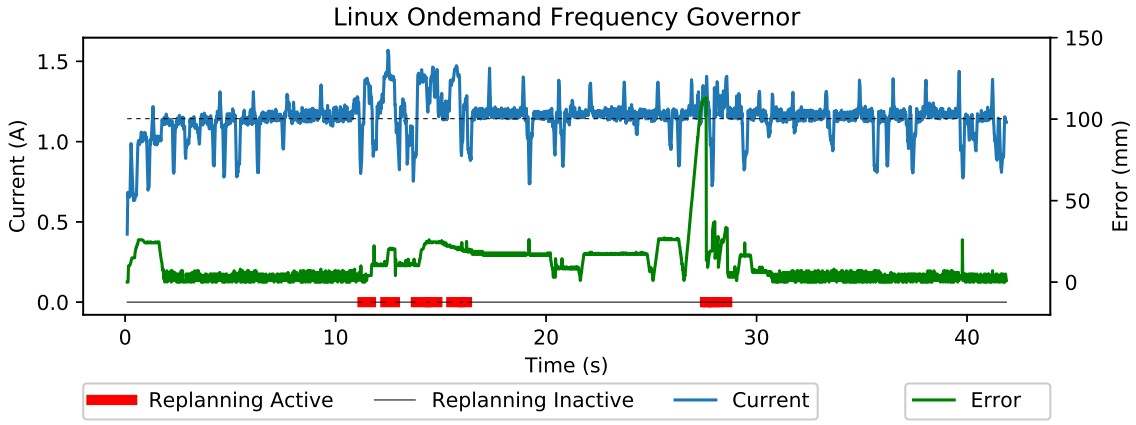
Figure 5.9a, 5.9b, and 5.9c show the experimentally measured power consumption, error, and activity of the static high-power (that is, the maximum frequency setting), on-demand, and situation-aware governors respectively.

In all the time-series figures, the green line and left y-axis show the actual measured current of the ODROID sampled every 10ms during the duration of the run. The thin dotted black line shows the average current for the duration of the run. The green line shows the deviation of the robot from the path in millimeters. On the bottom of each plot shows an indication of when the ADA* replanner is active (in red) and inactive (in black).

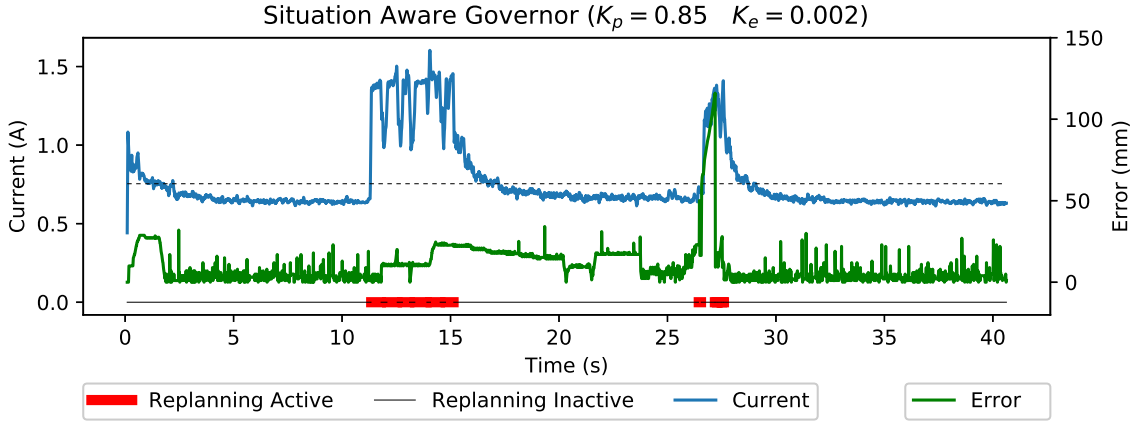
In the static-high-power case shown in Figure 5.9a, we see that the current stays nearly constant, averaging 1.234 A during the entire duration of the run. Here, in the two time periods in which the ADA* replanning is active, we see a corresponding increase in current draw, demonstrating the relationship between power consumption and activity, even when voltage and frequency remain the same (see Equation 5.1). The spike in error around 27s, seen in all the figures are due to a sharp turn the robot has to make around the obstacle. The ADA* lookahead parameter, while set to a small value in this run, still forces the robot to take a non-ideal path, causing a small spike in path deviation. Note that there are some periodic spikes, but it is difficult to determine their source because the ODROID is running an entire operating system, though we have disabled many unnecessary services.



(a) A single traversal of the experimental environment using the maximum power static governor.



(b) A single traversal of the experimental environment using the Linux dynamic on-demand governor.



(c) A single traversal of the experimental environment using the our situational-aware governor.

Figure 5.9: This figure shows the time-series data for three governors showing real-time current draw, path error, and activity of the replanning algorithm. [141]

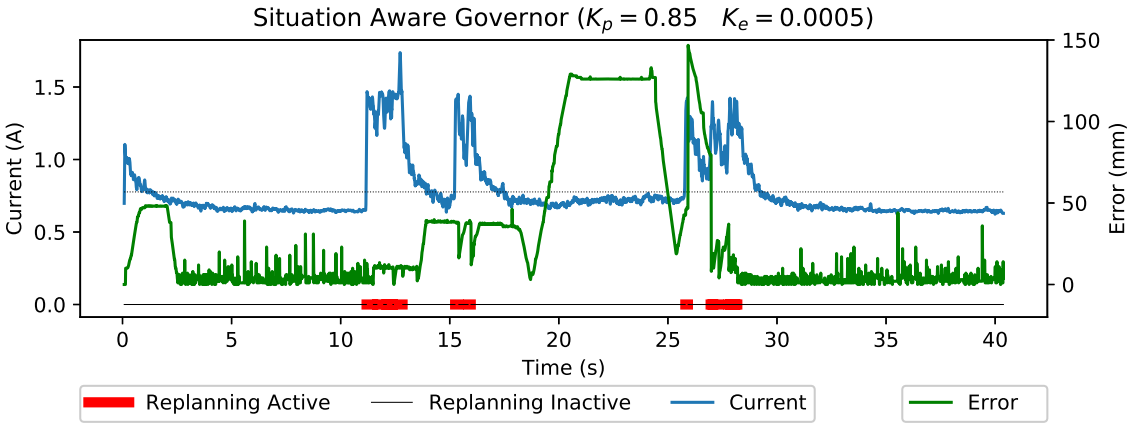
Figure 5.9b shows the performance of the on-demand governor. The governor is attempting to reduce the frequency, but the combination of system-wide idle statistics, hysteresis in frequency changes, and a slow sample-period, the on-demand governor fails to significantly reduce power, reducing the average current draw over the entire run by only 7.4% from that of the static high-power governor (1.234 A vs. 1.143 A).

Finally, Figure 5.9c shows the results from our situation-aware governor with $K_p = 0.85$ and $K_e = 0.002$. The QoS update algorithm in Equation 5.2 that implements the governor, is able to aggressively power-down the CPU, reducing the minimum current draw to half of either of the other governors and reducing the average current draw by 34% from that of the on-demand governor. Because it is aware of the importance and computational intensity of ADA*, the governor sets the CPU to its maximum performance during the time when a path replanning becomes necessary (between 12 s and 15 s, and around 27 s). The governor reverts to the algorithm in Equation 5.2 once the ADA* replanning is completed, showing a decrease in power usage similar in shape to a decay rate. This decay rate depends on the value of K_p, where higher values result in slower rates.

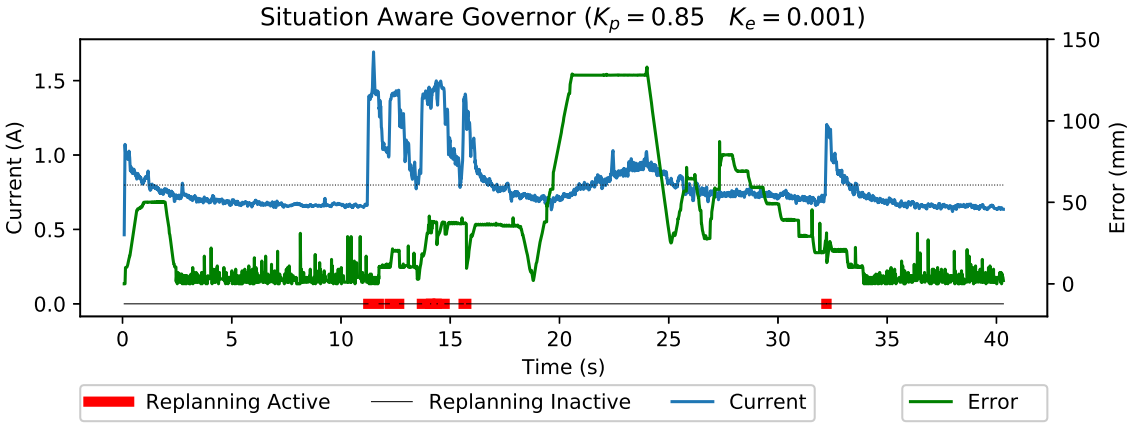
To further examine the behavior of the situation-aware governor's coefficients K_p and K_e, Figures 5.10a, 5.10b, and 5.10c fix the $K_p = 0.85$ and vary the K_e to place a greater sensitivity on the error values. The experiment represented by the data in Figure 5.10a sets $K_e = 0.0005$. During the large increase in error between 19 s and 25 s, there is very little upward push on the power mode. The downward pressure of the K_p term makes the governor very insensitive to the robot's error and simply tries to minimize power consumption, reaching an average current draw of 0.776 A.

Figure 5.10b doubles the K_e to 0.001 and the slight increase in power draw during the high-error portion of the run (19 s to 25 s) indicating the governor raising the power state in order to try to correct the error. The average current during this experiment is 0.798 A.

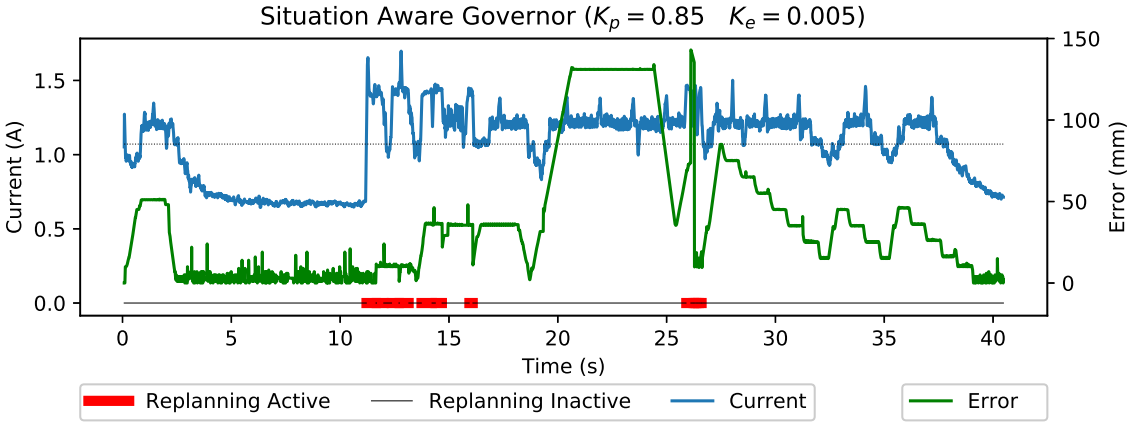
Finally, we increased the K_e value 5x to 0.005 and the results are shown in Figure 5.10c. In this case, when in the presence of very low error sch as between 2.5 s and 10 s, the K_p



(a) A single traversal of the simplified experimental environment using the situation-aware governor with ($K_p = 0.85, K_e = 0.0005$)



(b) A single traversal using the situation-aware governor ($K_p = 0.85, K_e = 0.001$)



(c) A single traversal using the situation-aware governor ($K_p = 0.85, K_e = 0.005$)

Figure 5.10: This figure shows the time-series data for the Situation-Aware Governor with $K_p = 0.85$ and varying K_e, showing real-time current draw, path error, and activity of the replanning algorithm.

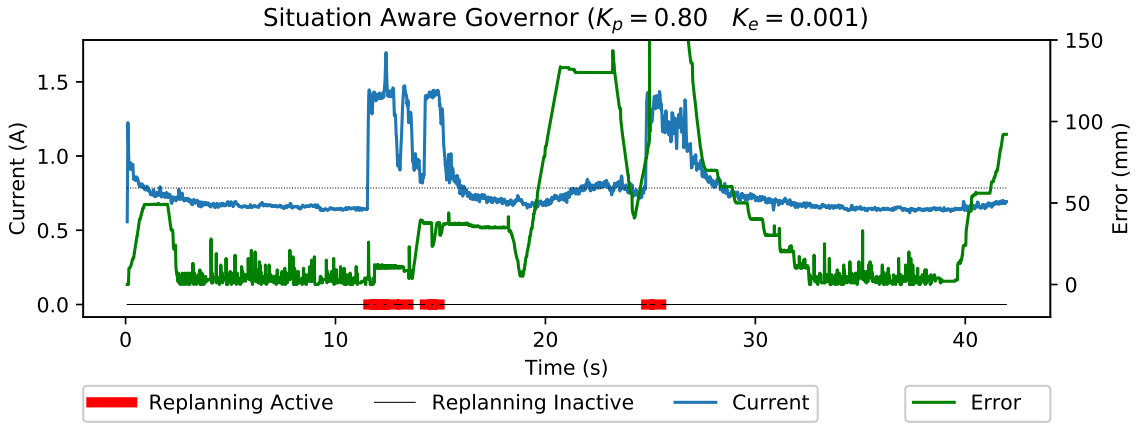
term reduces power in the same way as the other situation-aware governors. But in the presence of even a small error, the power increases significantly indicating the governor's attempt to correct error by increasing the power mode, giving an average current draw of 1.071 A.

In the experiments shown in Figures 5.11a, 5.11b, and 5.11c, the K_e was fixed at 0.001 and the K_p was varied to change the downward pressure on the power-state. For the first 10 seconds of each of these runs, the behavior is very similar with the lower values of K_p reducing the power more aggressively while the higher values of K_p more slowly reduce the power state. When $K_p = 0.80$ as in Figure 5.11a, even in the presence of large error, the governor aggressively attempts to push the power down. In Figure 5.11c, the higher error coupled with the less-aggressive (i.e. higher) K_p keeps the power mode high during periods of time with high error (e.g. 20 s to 32 s).

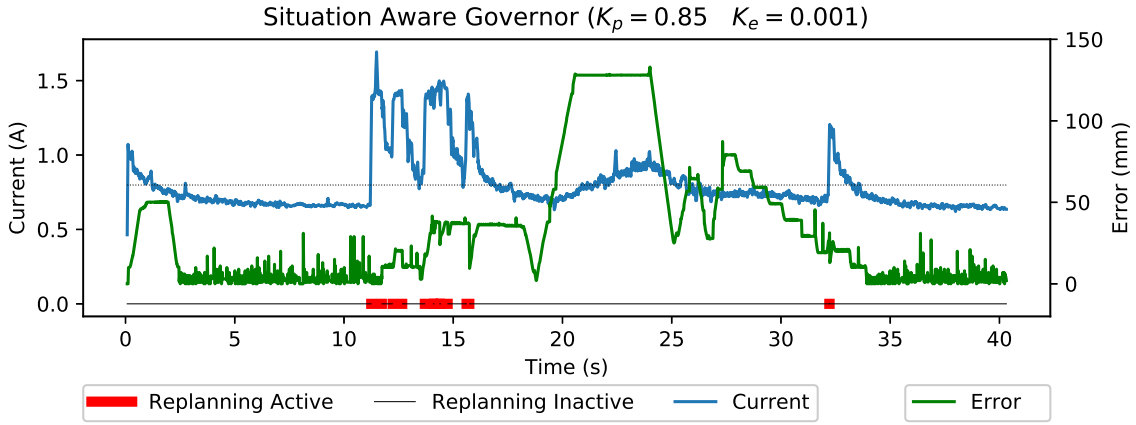
Figure	K_p	K_e	Avg. Current (A)	Energy (J)	RMS Err. (mm)	Time (s)
5.11a	0.80	0.001	0.772	186.7	76.14	48.37
5.10b	0.85	0.001	0.798	159.1	51.22	40.32
5.11c	0.90	0.001	0.885	178.4	55.82	40.41
5.10a	0.85	0.0005	0.776	156.7	50.99	40.38
5.10c	0.85	0.005	1.071	216.7	55.05	40.47

Table 5.1: The average current, total energy, RMS error, and time to traverse the course for the experiments in the simplified course discussed in Section 5.3.2 are summarized in the above table.

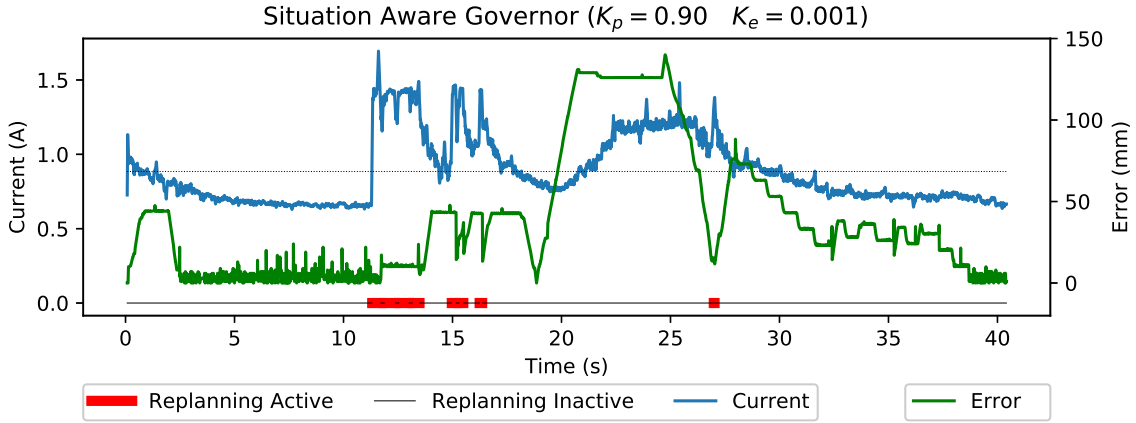
A summary of the important metrics for the simplified experiments of this section is shown in Table 5.1. It is important to note that these trials were selected as representative runs to show the behavior and are not the averages of multiple trials as in Figure 5.5. That being said, as in the time series plots, the behavior of the governor's parameters begin to emerge. For example, the most aggressive power-down coefficient ($K_p = 0.80$ shown in Figure 5.11a) has the lowest average current, however because it takes longer to traverse the course, uses a larger amount of energy than the less aggressive governors. Similarly, governors with larger error coefficients ($K_e = 0.005$ shown in Figure 5.10c) may have a



(a) A single traversal of the simplified experimental environment using the situation-aware governor ($K_p = 0.80, K_e = 0.001$)



(b) A single traversal using the situation-aware governor ($K_p = 0.85, K_e = 0.001$)



(c) A single traversal using the situation-aware governor ($K_p = 0.90, K_e = 0.001$)

Figure 5.11: This figure shows the time-series data for the Situation-Aware Governor with $K_e = 0.001$ and varying K_p, showing real-time current draw, path error, and activity of the replanning algorithm.

higher current draw without obtaining significant reduction in RMS error. These results illustrate the need to not just minimize the current or instantaneous power, but the total energy consumed. In addition, it demonstrates the need for a more flexible and powerful type of quality-of-service manager that is less reliant on optimizing coefficients but can instead learn to find optimal parameters.

5.4 Conclusions

A compute-aware software framework was developed that allows for control of an embedded processor's power management system in response to events caused by situation awareness in a physical system's controller. The goal is to have the ability to save power when the physical system's performance will not suffer from running the processor in a lower power mode. The software framework was designed to be modular and flexible enough to be used on a variety of platforms. To show the modularity and reusability of the framework, we developed a situation-aware recursive governor to serve as the QoS manager. This manager can be swapped out for more sophisticated algorithms as seen in Chapter 6. The feasibility of dynamically controlling the processor was tested experimentally on a mobile robot operating with obstacle avoidance.

It is clear that this governor is able to respond to the needs of the physical system controller, reducing power significantly. In the complex scenario, power was reduced by 26.9% over static high power and 20.6% of the existing power-management governor based upon CPU load rather than the situational awareness. In a simpler scenario, there was a power savings of 34% over the ondemand governor and 38.9% over high power mode. This significant power savings achieved by the situationally-aware method saw insignificant decrease in path-following performance (from 1.3-4.3% vs 3.5% for ondemand) and little increase in time to complete the task (1.3-1.8% longer).

The presence of phases with large error shows one of the challenges of tying power state directly to physical system error. Because the error of the path is not instantaneously

improved by the increase in power-mode and the error is often the accumulation of previous behavior, simply increasing the power mode when error is high does not guarantee a better overall run. Our experiments show that, in general, greater computational resources (i.e. higher performance CPU states) lead to lower overall error and faster traversal of the environment. However, this is where the importance of application guidance becomes clear. When the computing system prepares itself for a computationally intensive execution phase either through application guidance or prediction, it has a much better overall effect than simply reacting to computational or application performance metrics.

Another challenge is while all settings of the situation-aware governor reduce power compared to the default Linux governor, this power-manager requires tuning of K_p and K_e for the best performance. In the following chapter, we replace the situation-aware governor with a reinforcement learner in order to generalize the power-setting modes.

CHAPTER 6
COMPUTE-AWARE MANAGEMENT LAYER USING REINFORCEMENT LEARNING

The primary advantages of the software framework introduced in Chapter 1 and described in detail in Chapter 5 are modularity and abstraction which allow for flexibility. The flexibility is demonstrated in this chapter by replacing the situation-aware governor with a quality-of-service manager controlled via reinforcement learning while using, without modification, the same experimental platform. By eliminating the need for explicitly defining a power-management governor, we can simply take normalized metrics from application and hardware, and using a generalized but tunable reward function, obtain decision-making needed to balance power and performance. The Q-learning quality-of-service manager (2QoSM) outperforms both the default governors as well as the situation-aware governor from Chapter 5.

This chapter contains background on Q-Learning and its use in dynamic power management (Section 6.1), the framing of power management as a reinforcement learning problem (Sections 6.2 and 6.3), and experimental results on the robot test platform using a variety of reward functions (Section 6.4).

6.1 Introduction to Machine Learning

While machine learning is too broad of a topic to discuss in its entirety, some background information is required to introduce the terminology, explain the mechanisms and algorithms, and place this research into the context of recent work on computer power management. Machine learning is often divided into three paradigms: supervised learning, unsupervised learning, and reinforcement learning [156]. These techniques can occur in the discrete or continuous space. In the discrete case, supervised learning takes data and an associated label

and attempts to generalize so that future datapoints can be given a label. In the continuous case, supervised learning can be used for function approximation, as was done using a neural network in Section 3.2. Unsupervised learning attempts to cluster data and assign categories when labels aren't available. This work does not include unsupervised learning techniques.

The final type of machine learning problem, and the one encountered in this portion of the work, is *reinforcement learning*. It is discussed in detail in [157], but in essence it describes the situation in which an agent must learn the best action in its environment in order to maximize its reward. This type of problem can be formalized as a *Markov Decision Process*. A Markov Process is a memoryless stochastic process, that is, a process in which the following state is only dependent on the current state, not the history of states before. Therefore, a Markov Decision Process (MDP) is a discrete time stochastic control process represented by five variables: a set of states S, a set of actions A, the probability P that action a will transition to state s', the reward r received by transitioning from state s to s' by taking action a, and γ a discount rate that controls the value of immediate versus future rewards.

If we don't know *a priori* the probabilities P, one method of finding an optimal policy π^* is reinforcement learning. In this case, the agent observes itself in a certain state s_t, takes action a_t according to its policy π, finds itself in state s_{t+1} and receives a certain reward $r_t = r(s_t, a_t)$ [158]. This reward can have a finite-horizon or an infinite-horizon [157]. In the finite-horizon case, the goal is to maximize the expected reward for the next h steps whereas the infinite horizon case, the future rewards are discounted according to a discount factor γ

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \tag{6.1}$$

where r_t is the reward obtained at time-step t. These rewards are used to determine the optimal policy π^*.

6.1.1 Q-Learning

Q-Learning is a type of reinforcement learning technique that requires no knowledge of the underlying model [159, 160]. Instead Q-Learning estimates a real-valued function Q of states and actions where $Q(s, a)$ is the expected discount sum of future rewards for performing action a in state s [161]. The Q-values are stored in a matrix of dimensions $s \times a$. Q is defined [162] in slightly modified notation as the function

$$Q_{new}(s_t, a_t) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t)] \tag{6.2}$$

where $Q_{new}(s_t, a_t)$ is the updated value in the Q-matrix, α is a learning rate, r_t is the instantaneous reward, γ is the discount factor, and $\max_a Q(s_{t+1}, a_t)$ is the estimate of the optimal future value. The $\alpha \in [0, 1]$ learning rate is a parameter used to balance the incorporation of new information versus previously learned information. When the learning rate is high, the update of Q is more affected by the new reward and Q, whereas a smaller α favors existing Q values. The discount factor γ, as introduced in the previous section, determines the weighting of future rewards versus present rewards. In other words, if the discount factor is high, future rewards are weighted more, while a small discount factor favors instantaneous rewards. $\gamma = 0$ creates an algorithm that only maximizes instantaneous rewards, while $\gamma \geq 1$ will not converge and Q values will become infinite. The optimal future value is the best possible Q available in the next iteration after taking best action a.

The algorithm for updating the Q-matrix every step is shown as Algorithm 1.

In conventional Q-Learning, the algorithm ends when a goal state is reached. The system is restarted, saving the Q-matrix and learning continues. It is possible, however, to use a Q-Learner for non-episodic tasks, that is tasks which have no terminal state. As the learner improves its predicted future rewards through iterative updates to the Q-matrix, the difference between successive updates to a given $Q(s, a)$ begins to decrease, and the learner converges.

Algorithm 1 Q-Learning Update

1: **if** t = 0 **then**
2: Initialize Q-matrix $\forall a, s, Q(s, a) = 0$
3: **end if**
4: **while** $s_{t+1} \neq endstate$ **do**
5: Observe current state s_t
6: Find $\max_a Q(s_t, a)$
7: Take action a based upon selection policy
8: Calculate instantaneous reward r
9: Update Q based upon Equation 6.2
10: **end while**

There are a few details that can change the outcome of the learning which are not explored in detail in this research but are a topic for future experiments. It is good practice during training to, with a given probability ρ, take a random action instead of the best action in order to better explore the action-reward space. Also, it is possible to change the learning rate as the Q-learner converges, preventing large updates from outliers. While given infinite time the Q-Learner converges to an optimal policy, the parameters of γ, α and ρ can lead to different trained learners in finite time [163].

6.1.2 Reinforcement Learning and Power Management

A computer's power management system is a discrete-time stochastic control process [164]. Because the power consumption and performance of a system has both random and controlled outcomes, we can represent a power management system as a Markov decision process (MDP) [165]. Therefore, it is no surprise that a significant amount of work has been done using reinforcement learning techniques to optimize power and performance.

Reinforcement learning has been used extensively for power management in wireless networks [166, 167, 168, 169], building power systems [170, 171], and mobile/hybrid vehicles [172, 173]. Because the scope of machine learning in power management is so broad, this section will focus on machine learning used specifically for power management of computing systems. Tan *et al* [174], Liu *et al* [175], and Wang *et al* [176] used reinforcement

learning for dynamic power management of a hard disk attempting to balance power and latency. Tan *et al* were able to find a 24% reduction in power consumption and a 3% reduction in latency using Q-Learning compared to expert-based power management. Liu found a 30% to 60% decrease of power consumption in synthetic and real-world benchmarks using a hard drive simulator. Wang expanded the RL framework used on hard disk drives to wireless LAN cards as well.

Of particular interest to this work is the use of Q-learning to find optimal computer power/performance states (See Section 2.2.3) for a given workload. The majority of this research is focused on enterprise systems (cloud and virtualized instances) but many of the principles are similar. One important point to note is many of these techniques use machine learning to find an optimal configuration for a given workload, not to adjust the underlying hardware paramters in real-time based upon the performance of the application.

Rao *et al* use reinforcement learning to allocate memory and CPU cores to virtual machines based upon application performance compared to a pre-collected baseline performance [177]. This centralized technique was expanded to treat the allocation as a distributed learning task [178]. Tesauro *et al* use a Hybrid RL technique which uses offline batch RL to "seed" the learner to reduce the state space [179]. CoTuner also uses a hybrid RL approach for configuration of virtual machines and dynamic reconfiguration, however the interval for reconfiguration (1 s) is much longer than our system (10 ms) [180].

Zhang *et al* use temporal difference reinforcement learning to estimate the power consumption of virtual machines and allocate resources using a method they call cloud adaptive power management (CAPM) [181]. The researchers were able to estimate power consumption with 90% accuracy and better performance of the power manager compared to three other methods. CAPM is very closely related to this work, using both machine learning and a software framework, however it diverges in a few important ways. CAPM has the goal of allocating the optimal set of resources for a given job while our approach attempts to read the system state, take input from the application and hardware, and change the system

configuration in real time.

While these uses of Q-Learning for virtual machine provisioning share characteristics with our system, there has been research into dynamic power management controllers for CPUs that are very similar to the work described in this chapter.

Martinez and Ipek [182] examine machine learning techniques for low-level power management including DRAM scheduling and multiresource allocation. Ye and Xu use Q-Learning to optimize idle periods on multicore systems to reduce power consumption [183]. Our 2QoSM differs in that the authors attempt to reduce the amount of transitions, while 2QoSM is only concerned with the actual measured power consumption, not the relationship between state transitions and leakage current. If transitioning too often is increasing power consumption, 2QoSM should learn this. In addition, [183] is doing in simulation using synthetic benchmarks while the experimental platform in this work is a real application running on hardware. Shen *et al* [184, 185]use a Q-Learner to select DVFS states by constraining the performance and temperature while minimizing the energy. The metrics used are CPU intensiveness (similar to stall-cycle ratio examined in [32]), instructions-per-second, and temperature. Many of the design decisions are similar to the work described in this chapter, but our work has some notable differences. Most importantly, our reward and state is not based upon the CPU utilization (IPS and CPU intensiveness) but instead the measured performance of the application, which we believe is a much more complex and important indicator than CPU-load. In addition, the constraints are given by the user and thus can create scenarios in which the learner cannot find a policy that meets both constraints. Ge *et al* also rely on processor metrics, user-constraints, and CPU temperature for determining state and reward [186]

6.2 Defining the Problem in the Context of Reinforcement Learning

For power management to be defined as a Q-Learning problem, we need to be able to define both a current state, a set of actions, and an instantaneous reward. In our test platform, the

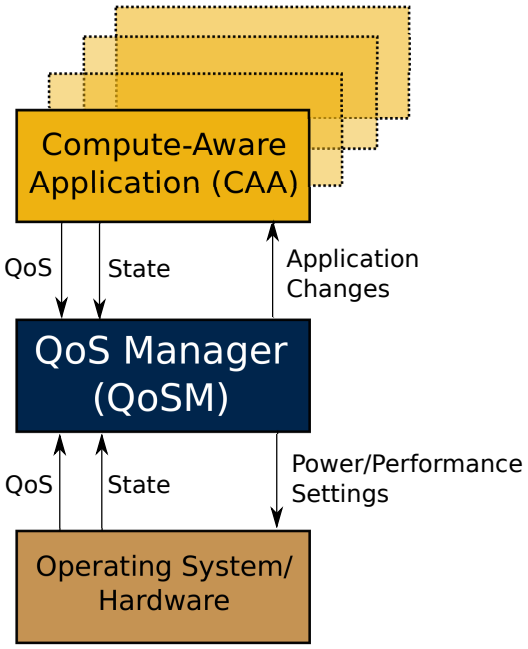


Figure 6.1: At a high level, the QoS Manager as described in Chapters 1 and 5 takes in a more generalized vector of data (state and QoS) and outputs a changes to hardware and application.

state of the system is a measurement of both the application and hardware. At a high level, the QoS Manager (QoSM) introduced in Chapter 1 and developed in Chapter 5 remains very similar to Figure 6.1. Instead of receiving specific metrics, the QoSM takes in a state vector and an application QoS.

To determine the state of the system, there are many metrics we could use. One challenge when using Q-Learning is that the state space increases multiplicatively with each additional state variable. Therefore, we must choose a useful set of state variables to minimize the search space while giving the most information possible.

The application can send any vector of data to the QoSM that the designer would find useful for describing an application state as well as a QoS measurement that describes the performance in the context of the application. The state vector could be as simple as a flag indicating whether a certain computationally intensive region of code is being run or it can be a set of metrics describing the activity of the application. This is left to the program designer. In our test platform, we selected two specific application states and a metric:

whether the ADA* replanning was active, whether the robot detected a new object, and the error from the path. This forms a vector of $[x_{replan}, x_{map}, \overline{x_{err}}]$. For application QoS, different metrics and combinations were evaluated, but the most basic QoS is a normalized error $E = [0, 1]$. Similarly the hardware uses discretized power as a state, and normalized power as a QoS.

Because our test platform has a heterogeneous CPU, our action space consisted of five DVFS operations: raise or lower the frequency on the high-performance cores, raise or lower the frequency on the lower-performance cores, or keep current DVFS settings (NOP). Tests were also conducted by adding an action that changed the heuristic parameter of ADA* (see Section 5.2.2 and [146] for more details), but it did not provide enough change to the system to warrant additional columns in the Q-matrix. In addition, the action space was expanded to turn on and off cores using Linux's CPU hotswap features but on our development board, this was very unstable and no tests completed.

A challenge to using Q-Learning with our test platform is the very nature of the physical system which operates on a slower time scale than the computing system. The physical error of the robot, that is the deviation from the ideal path, is correlated to the computational system power but there is not a direct relationship and doesn't respond instantaneously For example, even at maximum CPU performance, there is always a deviation from the ideal path due to the inertia of the mass, the lookahead of path planner, and precision of the odometry. In addition, even when physical system error is due to poor computational performance, increasing performance doesn't show an immediate improvement in the error because the robot is a dynamical system. This is in contrast to the power consumed by the robot which is directly proportional to the P-States of the processor (see Section 2.2). We use different techniques to try to balance these different states and is discussed in detail in Section 6.3.

6.3 Description and Implementation of System

The Q-Learner was written as a drop-in replacement for the QoS Manager described in Chapter 5. Instead of using an existing library for Q-Learning, we opted for a newly implemented Q-Learner in C using GNU Scientific Library [150]. While an existing library has advantages, there were a few motivating reasons to implement our learner from scratch. First of all, most of the machine learning libraries are written in Python which can not only too slow for systems programming, but also introduces a very large set of dependencies. Second, there exist major C++ implementations of the different libraries, but again, this requires changing the code from a purely C-based system to mixed languages. Third, the existing QoSM described in Chapter 5 was implemented in C and the existing API could be easily reused for a drop-in replacement. Finally, the actual Q-Learning algorithm is fairly simple and could be implemented cleanly with only the existing GSL library already used by our implementation of the ADA* algorithm.

6.4 Experimental Results

To test the effectiveness of the Q-Learning quality-of-service manager (2QoSM), we trained the learner on the mobile robot and then ran it through a course described in Chapter 5. We expanded the environment to 12m x 12m, again breaking it into 1 cm squares, this time creating a grid of 1,440,000 squares. For training, the environment consisted of 6 obstacles placed throughout the grid. For 600 s the robot traversed the grid moving to random points, replanning each time in order to simulate an infinite random environment. 600 s was chosen because most of the tested reward functions began saw convergence in this time period.

6.4.1 Reward Functions and Training

Figure 6.2 shows the updates to the Q-matrix for each tested reward function. As the learner converges, each update to the Q-matrix gets smaller because the expected value

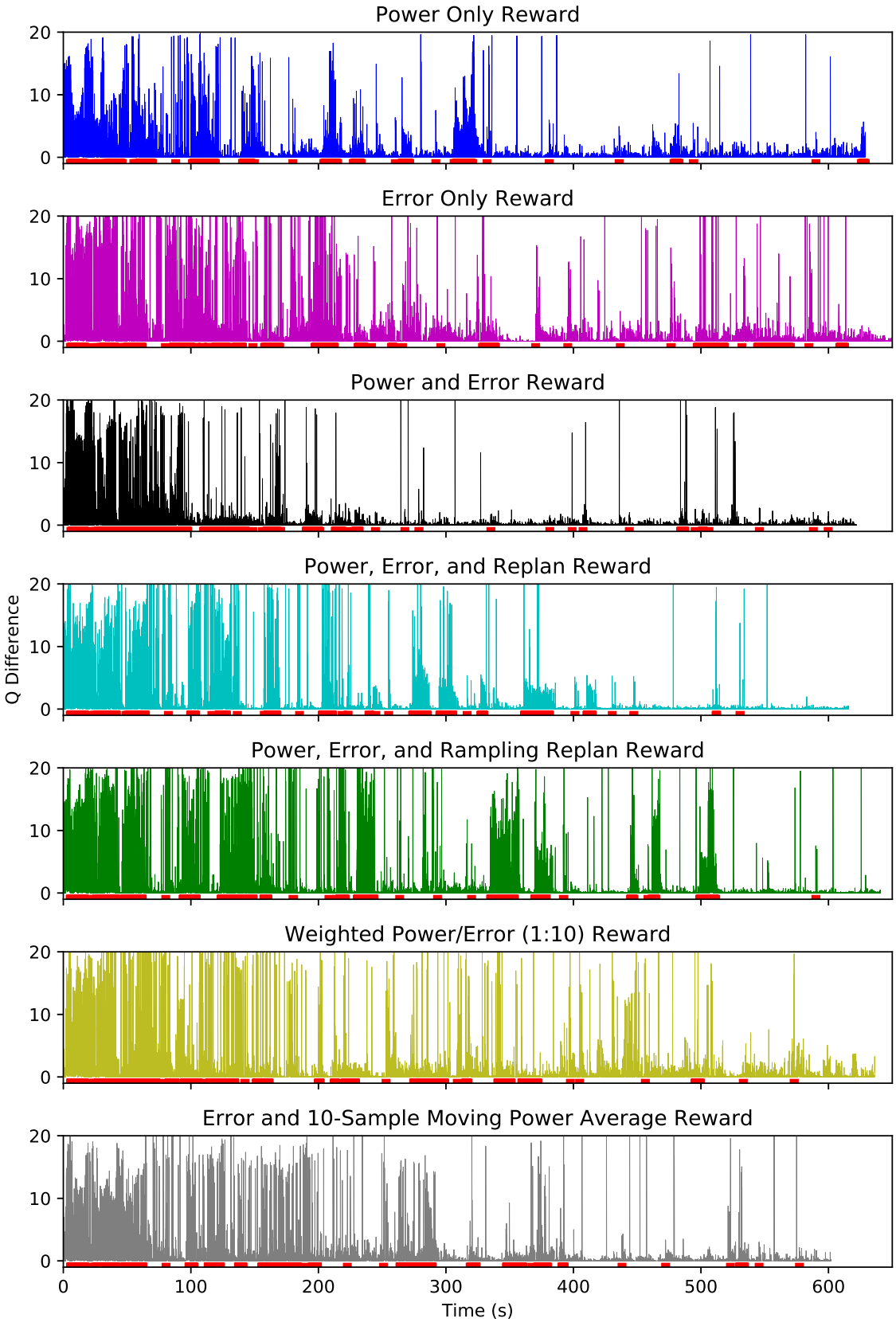


Figure 6.2: The differences in updates in the Q-matrix for different reward functions.

more closely matches the stored values. Because different reward functions have a different scale, the Q-differences were normalized to better compare the values between different reward functions. The red patches at the bottom of each plot show periods during which the path-planning algorithm is replanning. The reward r is made up of combinations of x_{pow}, x_{err}, x_{replan}.

The first time series in blue shows the convergence of the power-only reward function $r = 1 - x_{pow}$. It converges fairly quickly but continues to have bursts of larger updates throughout the training process. Because there is a near instantaneous change in power due to changes in P-state, the learner can quickly identify which actions increase the instantaneous reward (i.e. reduce power). But because we are also interested in the performance of the physical system, we again use error, that is, deviation from the ideal path as a measure of the physical system controller's quality of service.

First, we trained on the error by itself $r = 1 - x_{err}$. The second time series shown in the figure is the error-only reward function. While the learner does show some convergance, there are large updates throughout the training run. This demonstrates the difficulty that the Q-Learner has in predicting the best action when the outcome of previous actions are not immediately reflected in the instantaneous reward.

Since the goal is a balance between physical system performance (i.e. distance from path) and CPU power consumption, the next reward function tested was the sum of the power and error. Because it involves two normalized values $[0, 1]$, the reward function is calculated as $r = 2 - (x_{pow} + x_{err})$. The third graph in black shows the power and error combination reward function. It appears to converge the most quickly of the tested learners and with the exception of a few late spikes, it handles the replanning modes very well.

The fourth set of training data in Figure 6.2 shows the addition of the replanning mode in the reward function. Again, to keep reward positive, the reward function in this case is $r = 3 - (x_{pow} + x_{err} + x_{replan})$. The motivation for using replanning is to attempt to train Q-learner to avoid spending too much time in replanning mode by increasing performance

of the computational system in order to quickly find the path and then be able to reduce the P-states to save power. The final plot in black shows a reward function that contains the sum of power, error, and replanning mode. It takes a bit longer to converge than in the previous plot primarily in the sections in which the replanning change in the reward function creates a larger update in the Q-matrix. A likely explanation for this is that the replanning modes are not predicted by changes in error and power, and thus are unexpected events.

The fifth set of training data shows a ramping replanning instead of a simple negative flag. This change in the reward function is an attempt to teach the Q-learner to avoid spending time in the replanning mode by upping the performance of the CPU to complete it as quickly as possible. The reward function in this case is $r = 3 - (x_{err} + x_{pow} + \sum x_{replan})$. Like the previous data with replan added, there are bursts of large updates during replanning.

Because the changes in P-state are much more quickly reflected in the power than the path deviation, two techniques were attempted to balance the rewards: weighting the sum and using a moving average.

The sixth time series training data in gold shows a weighted reward function $r = 1 - (\alpha \cdot x_{pow} + (1 - \alpha) \cdot x_{err})$ where $\alpha = 0.1$. The value of $\alpha = 0.1$ was selected to try to put 10x the emphasis on error while still keeping the power component. This performs in a very similar manner to the error-only reward function, that is, poorly. Simply weighting the error more just increases the less predictive reward and reduces the easier to learn reward (i.e. power).

Finally, the seventh set of training data in grey shows a reward function $r = 2 - (x_{err} + \overline{x}_{pow})$ where $\overline{x}_{pow}$ is a 10-sample moving average. The thought behind this reward function is to change the time-scale of the observed reward, stretching the power to closer match the path deviation. This reward trained better than the weighted function above, even in larger regions of replanning.

There are clearly a large number of reward functions possible and this is only an examination of a few of them. In addition, the Q-Learner had a fixed $\gamma = 0.9$. Future work

could explore different parameters in combination with additional reward functions.

6.4.2 Aggregate Data

As in Chapter 5, it is helpful to look at the average metrics for the different reward functions. Figures 6.3, 6.4, and 6.5 show these aggregate values. Each bar is an average of 10 trials through the course.

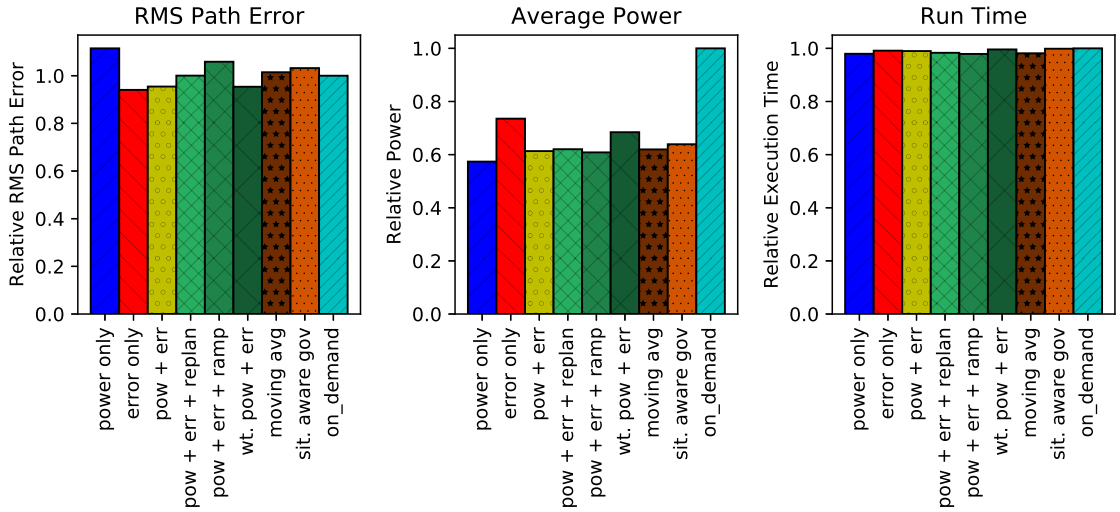


Figure 6.3: Aggregate metrics of the Q-Learner QoSM compared to the Linux ondemand governor.

Figure 6.3 show relative RMS path error, average power, and run time. The time needed to traverse the course is nearly the same for all governors and reward functions. These are relative based upon the Linux ondemand governor. They are also compared to the situation aware governor described in detail in Chapter 5. The power-only reward function behaves as expected, giving the lowest power but the worst error. Since the learner is not concerned with optimizing the error but only minimizing power, the error rises because the power mode is constantly reduced to its minimum. On the other hand, optimizing for error reduces the error to the lowest of all the reward functions but the power consumption is higher than all but the default Linux governor.

When the reward function is the simple sum of power and error, the error nearly matches

the error only case and the power consumption is better than the situation aware governor. The addition of replanning mode to the reward function doesn't give any improvement over the simple sum, though the ramping replanning performs better than the simple replanning flag.

Weighting the power and error with $\alpha = 0.9$ gives the second best RMS path error, but uses more power than all but the error-only reward function. Finally, the moving average is a fair balance between RMS error and power, but it does not perform as well the simple sum of power and error.

The lowest power consumption is shown by the power-only reward function, reducing power consumption by 2.98 W (42.6%) over the default Linux governor and 0.457 W (10.2%) from the situation-aware governor. The lowest RMS error is obtained by the error-only reward function, showing a 6.1% improvement over the on-demand governor and 8.9% improvement over the situation-aware governor. The simple sum of power and error reward function reduce power compared to the Linux on-demand governor by 2.703 W (38.7%) and by our situation-aware governor by 0.18 W (4.0%). The error is also improved over the on-demand and situation-aware governors by 4.6% and 7.49% respectively.

Figure 6.4 shows the energy-delay product (EDP: $w = 1$) and the energy-error-delay product (EEDP: $w_e = 1 w_d = 1$)introduced in [141] and explained in Section 5.3. Much like in the previous results, since the time to traverse the course is nearly the same but power is lowest in the most aggressive power-conscious mechanisms (in this case, the power-only reward function), the best EDP is found using these techniques. However, if we add physical system error to our metric as in EEDP, the reward function using the sum of power and error emerges as the strongest technique for QoSM. Nearly all of the Q-Learners show an improvement over the best performing situation-aware governor and all show a huge improvement over the Linux on-demand governor.

Finally, in Figure 6.5, we examine the performance of different weightings of the EEDP. By weighting the components of EEDP, system designers are better able to evaluate the

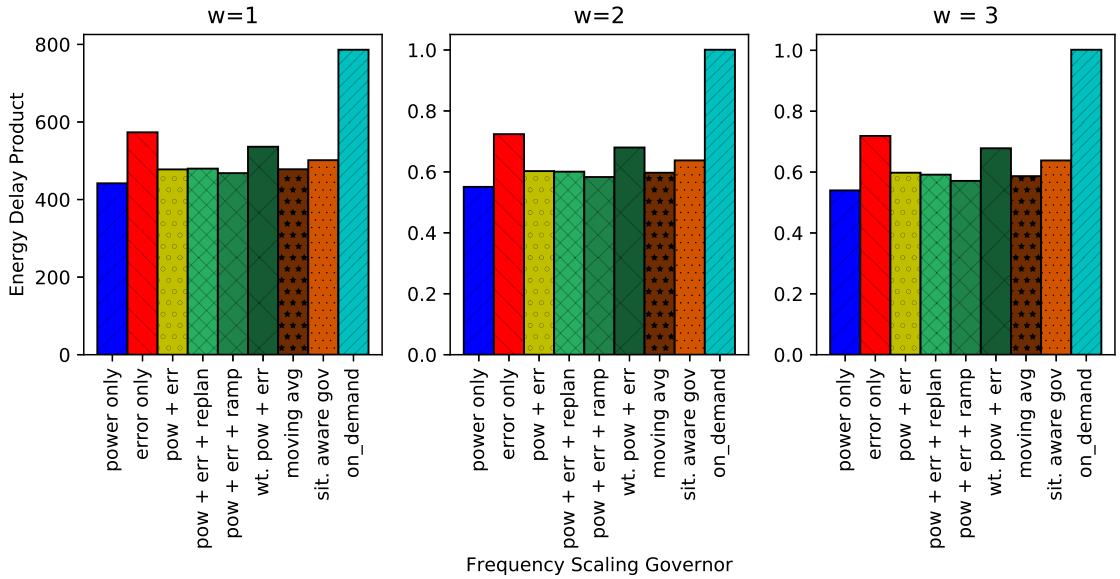


Figure 6.4: Energy-delay product (EDP) the Q-Learner QoSM compared to the Linux ondemand governor.

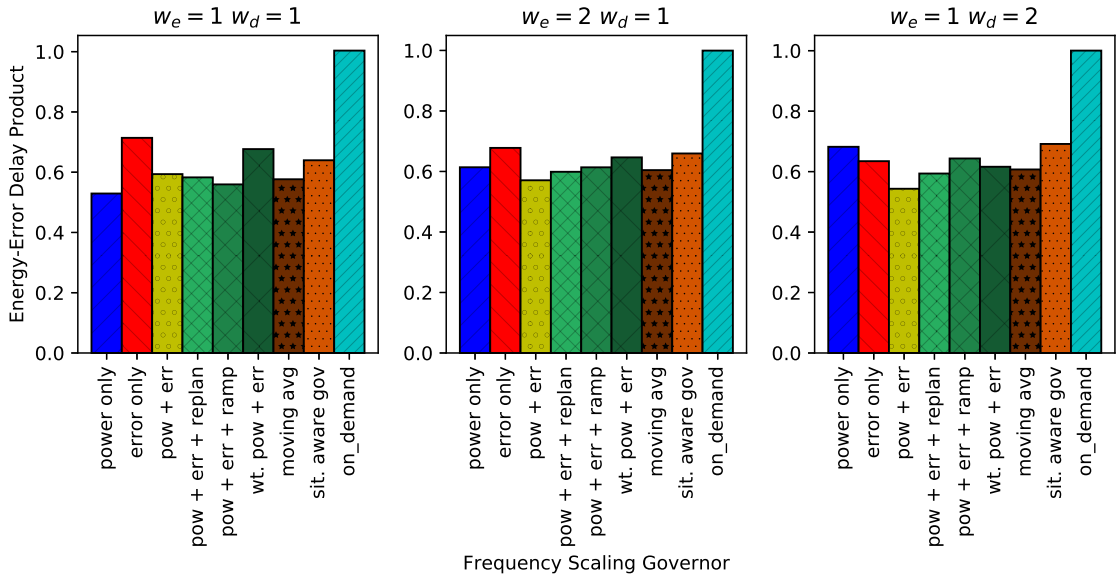


Figure 6.5: Aggregate metrics of the Q-Learner QoSM compared to the Linux ondemand governor.

performance of system changes. In all weightings, the simple sum of power and error shows the best overall performance but power-only and error-only change places depending on whether power or error receive more emphasis.

6.4.3 Time Series Analysis

After evaluating the average performance over a set of runs, this section examines the runtime behavior of representative examples of each reward function. To see examples of the situation-aware and on-demand governors, refer to Section 5.3.2. In these representative time series, in the top plot, the blue line shows real-time current draw of the ODROID, the green line shows the path error, and red shows sections where ADA* is replanning. The bottom plot in gold shows the instantaneous reward obtained after each action. It should be noted that the spikes in the error after replanning are due to the fact that while the ADA* algorithm is replanning, the robot is continuing on it's original path, often directly into the obstacle. When the new path is calculated, the robot sees a very large deviation of the path and thus makes a sharp turn to find the newly-calculated waypoint.

Figure 6.6 shows the performance of the power-only reward function. Because of the simple relationship between the action and reward, the learner rapidly reduces power consumption whenever replanning is not active. The reward clearly mirrors the instantaneous current draw. Current spikes outside of replanning are due to two primary circumstances: random actions taken by the Q-Learner to improve it's learning ability as well as background processes.

In Figure 6.7 the runtime behavior of the error-only reward function. Because the only metric used in the reward function is error and the error is not immediately or causally affected by the available actions of the learner, the behavior of this learner is much less easy to follow.

The best performing reward function, the simple sum of power and error, is shown in Figure 6.8. This learner shows the same general characteristics as the power-only reward

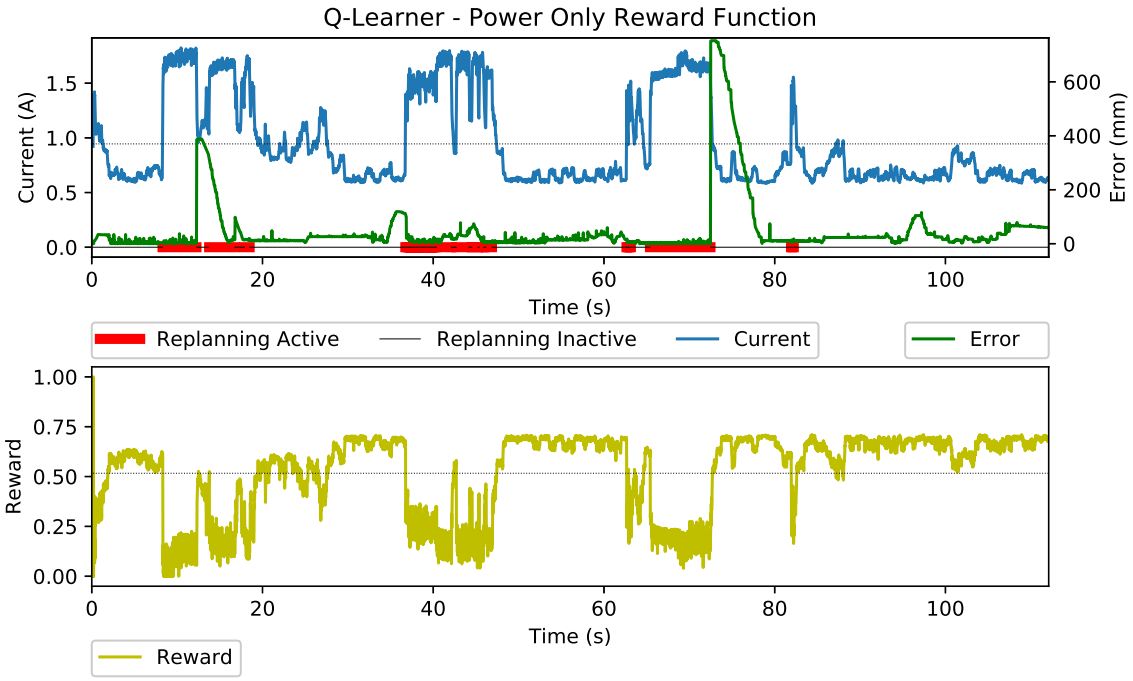


Figure 6.6: The QoSM using a Q-Learner with a power-only reward function.

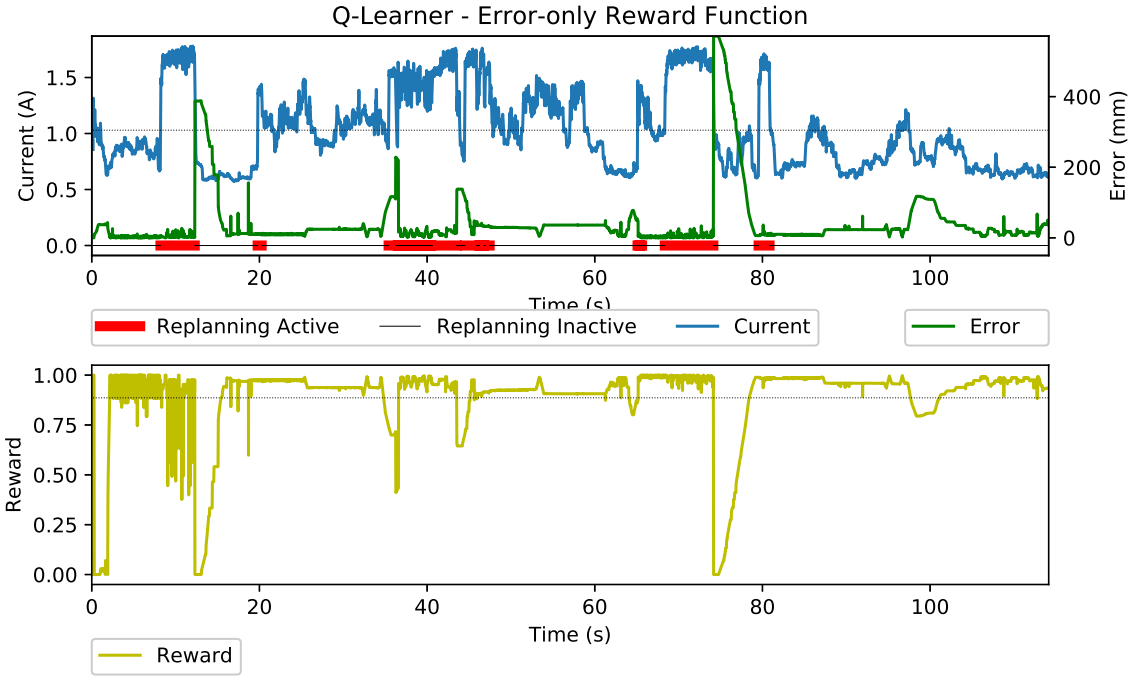


Figure 6.7: The QoSM using a Q-Learner with a error-only reward function.

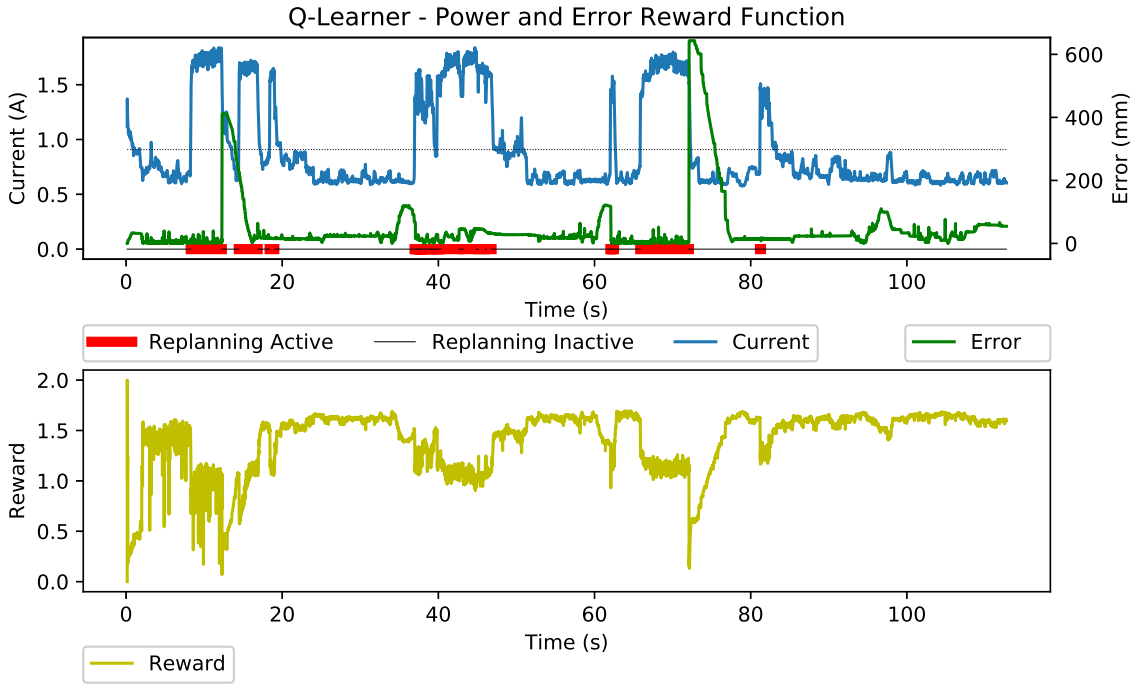


Figure 6.8: The QoSM using a Q-Learner with a simple sum of power and error for the reward function.

function, however as the aggregate statistics in Section 6.4.2 show, the overall performance is better.

Figures 6.9 and 6.10 show the behavior of the QoSM after the addition of replanning to the reward function. Because of the presence of the power in the reward function, the power is reduced overall, but the reaction to error seems overpowered by replanning without showing improvement.

The weighted reward function ($\alpha = 0.9$) shown in Figure 6.11 shows interesting behavior. Because the power is still a portion of the reward function, it seems to reduce power for the duration of the run. However, due to the stronger addition of the error, there are spikes in power when the error gets large which seems to indicate that the learner finds some relationship between its actions changing the P-states and the error.

While the 10-sample moving average shown in Figure 6.12 performs well, it does seem to reduce power a bit slower than the other learners which is expected since the current value is the moving average that lags behind the actual value. This seems to demonstrate the

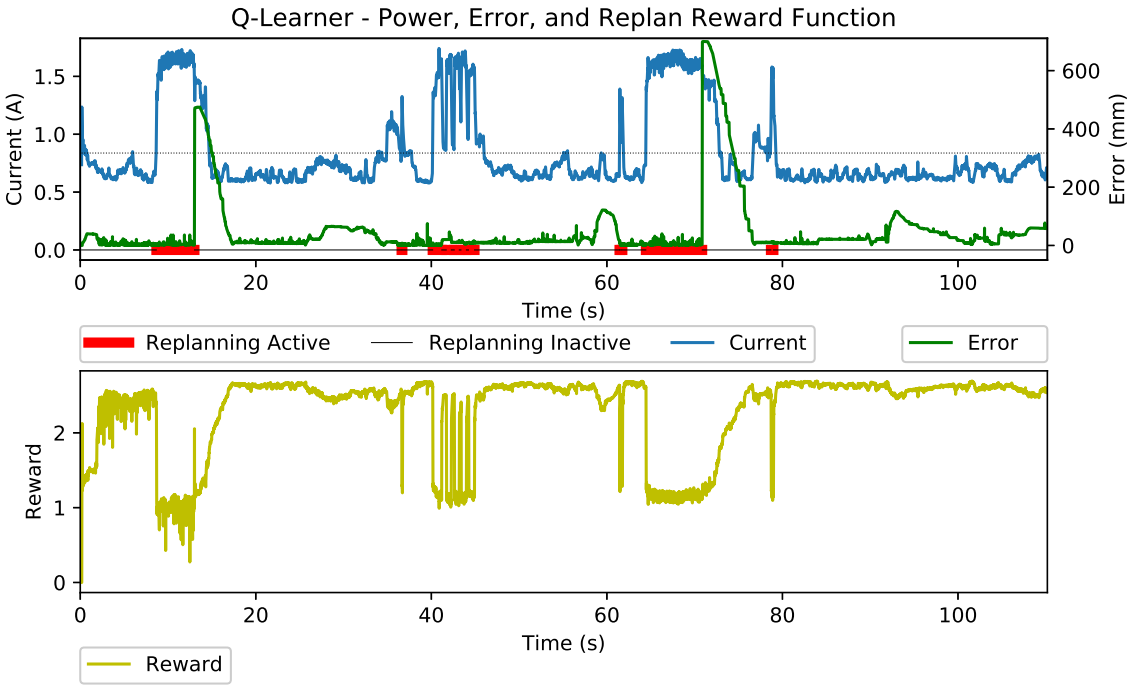


Figure 6.9: The QoSM using a Q-Learner with a power, error, and replanning as the reward function.

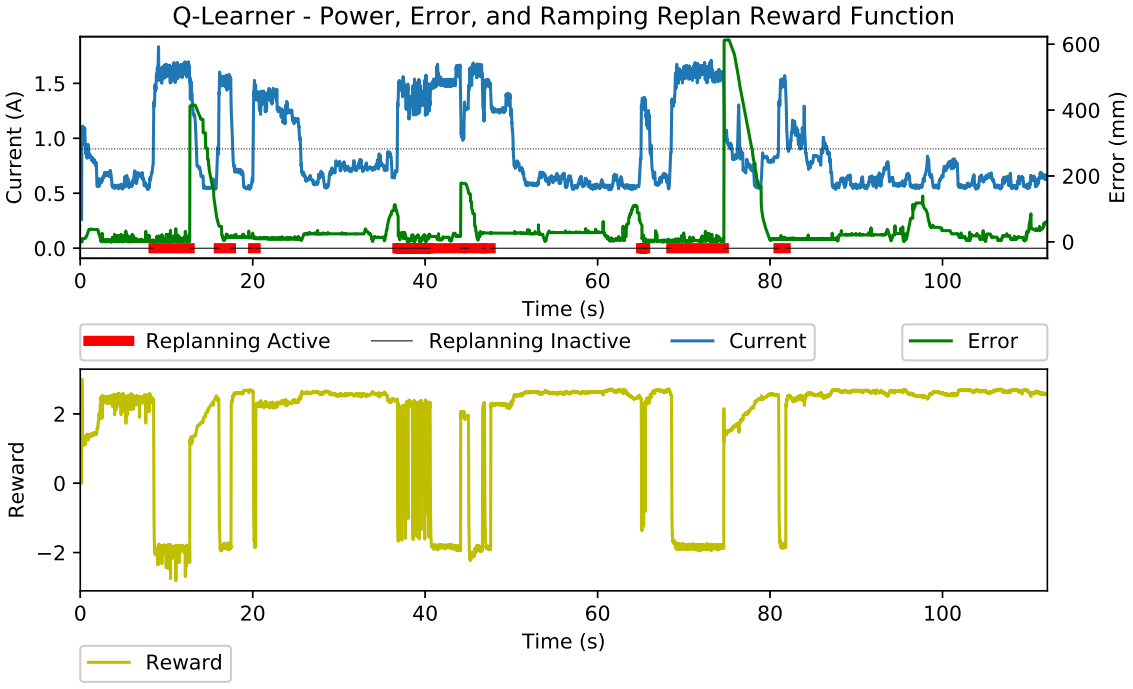


Figure 6.10: The QoSM using a Q-Learner with a power, error, and a ramping replanning reward function.

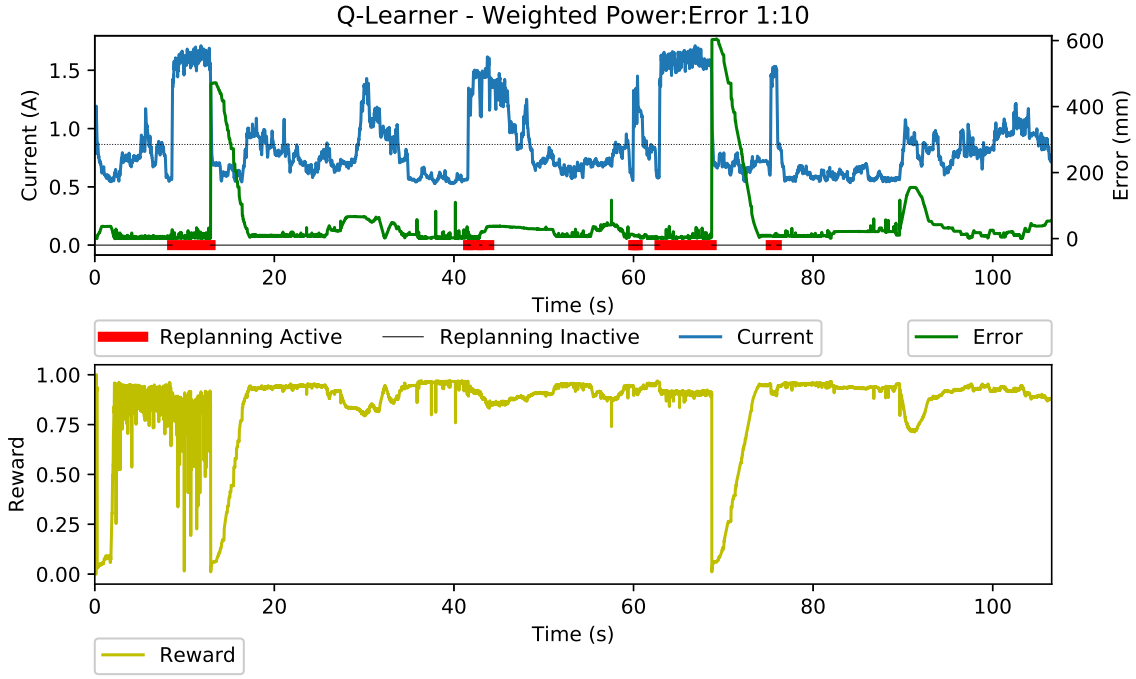


Figure 6.11: The QoSM using a Q-Learner with a weighted sum of power and error (1:10) reward function.

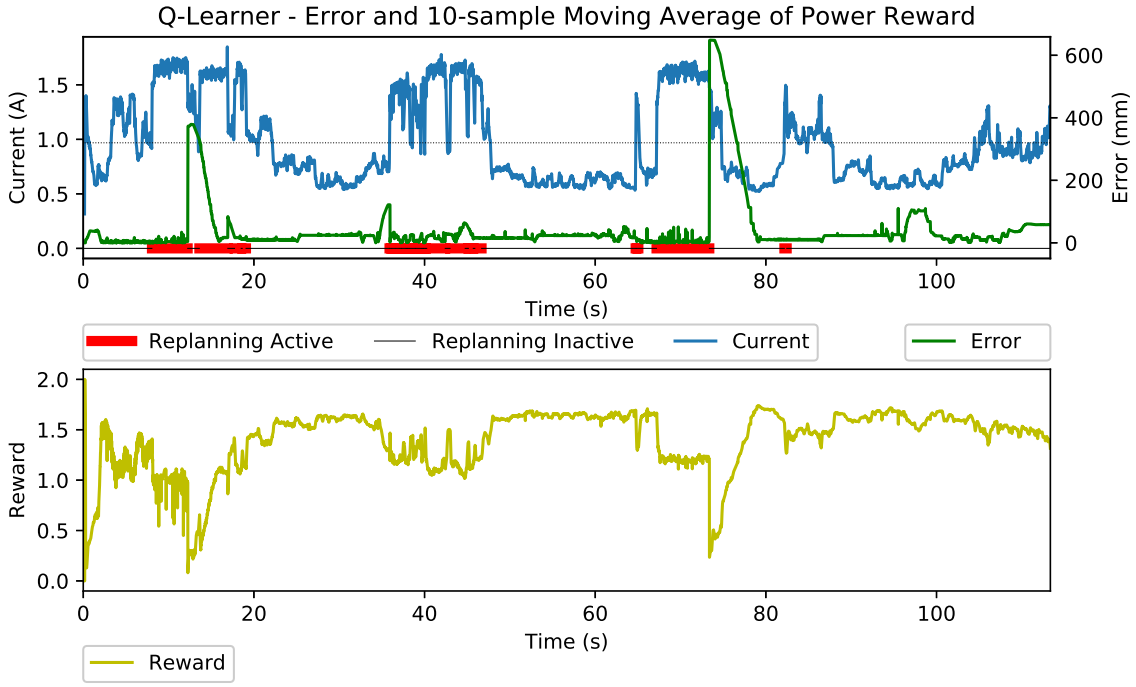


Figure 6.12: The QoSM using a Q-Learner with the reward function made up of the sum of error and 10-sample moving average of the power.

correct behavior of the moving average but it doesn't show improvement of the simple sum.

6.5 Conclusion

Using the Q-Learner to as a drop-in replacement for the previously developed QoSM demonstrates a few advantages of this system. First of all, the ease of replacing a high-value component shows the versatility and reusability of this paradigm of power management. Future work will look at different machine learning techniques as well as varied applications both in a single-purpose environment as well as for general application management. Second, using a modern machine learning technique (Q-Learning) to create a generalized quality-of-service manager makes a strong case for non-heuristic power managers.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This dissertation makes the case for a proactive method of computer power management. To find a balance between software isolation and hardware/software codesign, a framework is described and implemented that takes into account the performance and needs of software and hardware and reconfigures both in order to save power with minimum performance penalty. The bidirectional communication between hardware and software expands upon the concepts of dynamic power management and application guidance to create a unified quality-of-service manager (QoSM) to make power- and performance-aware decisions on behalf of both application and hardware.

To enable power- and performance-aware system management, enabling work in state estimation, prediction, and control was completed. This preliminary work introduces methods which allow for more complex and powerful quality-of-service management techniques. Because switching between algorithms can introduce instabilities, a novel technique for transient management was demonstrated using speculative threads.

On the autonomous robot experimental platform, two different quality-of-service managers were tested. The results show 20-40% reductions in power consumption with less than 5% decrease across two metrics of physical system performance. While these results are strong, there is a lot of space to explore using different parameters, test platforms, and machine learning techniques.

7.2 Future Work

The primary direction for future work lies in the expansion of the machine-learning techniques for power and performance management. There are two very promising avenues for expanding this work: examining different applications and using different machine learning techniques.

This dissertation focused primarily on embedded systems, specifically those in a mobile robot. As discussed in the background section, enterprise and high-performance systems share some characteristics with embedded systems. For one, there is often only a single primary application running on a given system. This means that there is a greater opportunity to use application guidance and performance metrics to lead the hardware towards the right performance states. Second, power budgets in data centers are very important because of the sheer scale of these systems. Another interesting area to explore is the optimization of multiple different applications with the Q-learner, expanding the number of states based upon multiple application performance metrics. Some preliminary work has been done in treating applications and hardware as "sources" and "sinks" of resources into a single learner. This expansion could allow dynamic optimization of heterogeneous applications and hardware platforms.

The second major area of improvement is in the type of machine learning used for the quality-of-service manager (QoSM). For one, the Q-Learner has a number of parameters that were not explored in detail. Once a set of parameters was found to be satisfactory, the remaining experiments were conducted with these settings. There could be much to learn for a greater parametric exploration of the Q-Learner. Second, while the Q-Learner showed better performance than our situation aware governor and the default Linux governor, Q-Learning seems to work best when there is an immediate causal relationship between the action, state, and observed reward. Because the change in the physical system performance is not only delayed from the changes to the computational system but also is affected by

situations outside the control of the system, the Q-Learner can struggle to find the optimal settings for hardware. There are other techniques such as $Q(\lambda)$ that may be a better match for this type of system.

REFERENCES

[1] A. S. G. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015.

[2] A. Andrae, "Total consumer power consumption forecast," *Nordic Digital Business Summit*, 2017.

[3] S. I. Association, "2015 international technology roadmap for semiconductors (itrs)," 2015.

[4] D. J. Goodman and C. E. Sundberg, "Quality of service and bandwidth efficiency of cellular mobile radio with variable-bit-rate speech transmission," in *33rd IEEE Vehicular Technology Conference*, vol. 33, May 1983, pp. 316–321.

[5] M. Andrews, K. Kumaran, K. Ramanan, A. Stolyar, P. Whiting, and R. Vijayakumar, "Providing quality of service over a shared wireless link," *IEEE Communications Magazine*, vol. 39, no. 2, pp. 150–154, Feb. 2001.

[6] C. G. Cassandras and W. Shi, "Perturbation analysis of multiclass multiobjective queueing systems with 'quality-of-service' guarantees," in *Proceedings of 35th IEEE Conference on Decision and Control*, vol. 3, Dec. 1996, 3322–3327 vol.3.

[7] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic, "Controlware: A middleware architecture for feedback control of software performance," in *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ser. ICDCS '02, Washington, DC, USA: IEEE Computer Society, 2002, pp. 301–, ISBN: 0-7695-1585-1.

[8] T. T. Dhivyaprabha, "QoS agent based framework and algorithm for task scheduling in grid," in *2014 IEEE 8th International Conference on Intelligent Systems and Control (ISCO)*, Jan. 2014, pp. 218–223.

[9] D. Perez-Palacin, R. Mirandola, F. Monterisi, and A. Montoli, "QoS-driven Probabilistic Runtime Evaluations of Virtual Machine Placement on Hosts," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, Dec. 2015, pp. 90–94.

[10] M. Shafiuzzaman, N. Nahar, and M. R. Rahman, "A proactive approach for context-aware self-adaptive mobile applications to ensure quality of service," in *2015 18th International Conference on Computer and Information Technology (ICCIT)*, Dec. 2015, pp. 544–549.

[11] V. Anagnostopoulou, M. Dimitrov, and K. A. Doshi, "SLA-guided energy savings for enterprise servers," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, Apr. 2012, pp. 120–121.

[12] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94, Monterey, California: USENIX Association, 1994.

[13] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithm for dynamic speed-setting of a low-power cpu," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '95, Berkeley, California, USA: ACM, 1995, pp. 13–25, ISBN: 0-89791-814-2.

[14] *Advanced configuration and power interface specification v6.1*, 2016.

[15] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, Apr. 1992.

[16] J. Wakerly, *Digital design: Principles and practices (4th edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005, ISBN: 0131863894.

[17] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.

[18] H. J. M. Veendrick, "Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits," *IEEE Journal of Solid-State Circuits*, vol. 19, no. 4, pp. 468–473, Aug. 1984.

[19] Intel Corporation and Microsoft Corporation, *Advanced Power Management (APM) BIOS Interface Specification v1.2*, Feb. 1996.

[20] R. Schöne, D. Molka, and M. Werner, "Wake-up latencies for processor idle states on current x86 processors," *Comput. Sci.*, vol. 30, no. 2, pp. 219–227, May 2015.

[21] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable voltage core-based systems," in *Proceedings of the 35th Annual Design Automation Conference*, ser. DAC '98, San Francisco, California, USA: ACM, 1998, pp. 176–181, ISBN: 0-89791-964-5.

[22] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proceedings of the 1998 International Symposium on*

Low Power Electronics and Design, ser. ISLPED '98, Monterey, California, USA: ACM, 1998, pp. 76–81, ISBN: 1-58113-059-7.

[23] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[24] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, ser. FOCS '95, Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–, ISBN: 0-8186-7183-1.

[25] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1571–1580, Nov. 2000.

[26] AMD, *AMD K6-2E+ Datasheet*, Sep. 2000.

[27] K. Nikov, J. L. Nunez-Yanez, and M. Horsnell, "Evaluation of hybrid run-time power models for the arm big.little architecture," in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, Oct. 2015, pp. 205–210.

[28] Intel Corporation, *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*, Sep. 2008.

[29] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel core i7 turbo boost feature," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 188–197.

[30] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz, "Beyond dvfs: A first look at performance under a hardware-enforced power bound," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 947–953.

[31] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug. 2010, pp. 189–194.

[32] M. Giardino and B. Ferri, "Correlating hardware performance events to cpu and dram power consumption," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Aug. 2016, pp. 1–2.

[33] L. Benini, A. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 299–316, Jun. 2000.

[34] D. Brodowsk, N. Golde, V. Kumar, and R. J. Wysocki, *Cpu frequency and voltage scaling code in the linux(tm) kernel*, ch. CPUFreq Governors.

[35] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, sn, vol. 2, 2006, pp. 215–230.

[36] N. B. Rizvandi, J. Taheri, and A. Y. Zomaya, "Some observations on optimal frequency selection in dvfs-based energy consumption minimization," *J. Parallel Distrib. Comput.*, vol. 71, no. 8, pp. 1154–1164, Aug. 2011.

[37] S. Ahmed and B. H. Ferri, "Prediction-based asynchronous cpu-budget allocation for soft-real-time applications," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2343–2355, Sep. 2014.

[38] R. Efraim, R. Ginosar, C. Weiser, and A. Mendelson, "Energy aware race to halt: A down to earth approach for platform energy management," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 25–28, Jan. 2014.

[39] E. Rotem, U. C. Weiser, A. Mendelson, R. Ginosar, E. Weissmann, and Y. Aizik, "H-earth: Heterogeneous multicore platform energy management," *Computer*, vol. 49, no. 10, pp. 47–55, Oct. 2016.

[40] R. Ge, X. Feng, W. c. Feng, and K. W. Cameron, "Cpu miser: A performance-directed, run-time system for power-aware clusters," in *2007 International Conference on Parallel Processing (ICPP 2007)*, Sep. 2007, pp. 18–18.

[41] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 143–154.

[42] S. Albers and A. Antoniadis, "Race to idle: New algorithms for speed scaling with a sleep state," *ACM Trans. Algorithms*, vol. 10, no. 2, 9:1–9:31, Feb. 2014.

[43] A. Das, G. V. Merrett, and B. M. Al-Hashimi, "The slowdown or race-to-idle question: Workload-aware energy optimization of smt multicore platforms under process variation," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 535–538.

[44] M. A. Awan and S. M. Petters, "Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems," in *2011 23rd Euromicro Conference on Real-Time Systems*, Jul. 2011, pp. 92–101.

[45] K. Kumar, K. Doshi, M. Dimitrov, and Y. H. Lu, "Memory energy management for an enterprise decision support system," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, Aug. 2011, pp. 277–282.

[46] M. E. Tolentino, J. Turner, and K. W. Cameron, "Memory miser: Improving main memory energy efficiency in servers," *IEEE Transactions on Computers*, vol. 58, no. 3, pp. 336–350, Mar. 2009.

[47] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, Washington, DC, USA: IEEE Computer Society, 2007, pp. 134–145, ISBN: 0-7695-3047-8.

[48] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, New York, NY, USA: ACM, 2009, pp. 24–33, ISBN: 978-1-60558-526-0.

[49] M. Giardino, K. Doshi, and B. Ferri, "Soft2LM: Application Guided Heterogeneous Memory Management," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Aug. 2016, pp. 1–10.

[50] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, ser. HotPower'10, Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8.

[51] L. Tan and Z. Chen, "Slow down or halt: Saving the optimal energy for scalable hpc systems," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15, Austin, Texas, USA: ACM, 2015, pp. 241–244, ISBN: 978-1-4503-3248-4.

[52] P. Cao, E. W. Felten, and K. Li, "Implementation and performance of application-controlled file caching," in *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '94, Monterey, California: USENIX Association, 1994.

[53] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '91, Pacific Grove, California, USA: ACM, 1991, pp. 95–109, ISBN: 0-89791-447-3.

[54] A. W. Appel and K. Li, "Virtual memory primitives for user programs," *SIGPLAN Not.*, vol. 26, no. 4, pp. 96–107, Apr. 1991.

[55] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412–418, Jul. 1981.

[56] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, Copper Mountain, Colorado, USA: ACM, 1995, pp. 251–266, ISBN: 0-89791-715-4.

[57] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA: USENIX, 2012, pp. 335–348, ISBN: 978-1-931971-96-6.

[58] *Madvise(2) linux programmers's manual*, Apr. 2014.

[59] M. R. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. A. Doshi, "A Framework for Application Guidance in Virtual Memory Systems," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13, New York, NY, USA: ACM, 2013, pp. 155–166, ISBN: 978-1-4503-1266-0.

[60] C. Cantalupo, V. Venkatesan, J. R. Hammond, and S. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," 2015.

[61] H. Kopetz, *Real-time systems: Design principles for distributed embedded applications*, 2nd. Springer Publishing Company, Incorporated, 2011, ISBN: 9781441982360.

[62] P. Caspi and O. Maler, "From control loops to real-time programs," in *Handbook of Networked and Embedded Control Systems*, D. Hristu-Varsakelis and W. S. Levine, Eds. Boston, MA: Birkhäuser Boston, 2005, pp. 395–418, ISBN: 978-0-8176-4404-8.

[63] Q. Li, "Fundamentals of rtos-based digital controller implementation," in *Handbook of Networked and Embedded Control Systems*, D. Hristu-Varsakelis and W. S. Levine, Eds. Boston, MA: Birkhäuser Boston, 2005, pp. 353–375, ISBN: 978-0-8176-4404-8.

[64] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem: Overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 36:1–36:53, May 2008.

[65] B. H. Ferri, A. A. Ferri, and G. B. Runge, "Frequency-weighted variable-length controllers using anytime control strategies," *Journal of Dynamic Systems, Measurement, and Control*, vol. 137, no. 8, Aug. 1, 2015.

[66] J. Muller, A. Rottmann, L. M. Reindl, and W. Burgard, "A probabilistic sonar sensor model for robust localization of a small-size blimp in indoor environments using a

particle filter," in *2009 IEEE International Conference on Robotics and Automation*, May 2009, pp. 3589–3594.

[67] E. Hygounenc, I.-K. Jung, P. Soueres, and S. Lacroix, "The autonomous blimp project of laas-cnrs: Achievements in flight control and terrain mapping," *The International Journal of Robotics Research*, vol. 23, no. 4-5, pp. 473–511, 2004.

[68] K. H. Jones and J. N. Gross, "Reducing size, weight, and power (swap) of perception systems in small autonomous aerial systems," in *14th AIAA Aviation Technology, Integration, and Operations Conference*, 2014, p. 2705.

[69] B. Li and K. Nahrstedt, "A control-based middleware framework for quality-of-service adaptations," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1632–1650, Sep. 1999.

[70] H. Hoffmann, "Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems," in *2014 26th Euromicro Conference on Real-Time Systems*, Jul. 2014, pp. 223–232.

[71] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "Poet: A portable approach to minimizing energy under soft real-time constraints," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2015, pp. 75–86.

[72] C. Imes and H. Hoffmann, "Bard: A unified framework for managing soft timing and power constraints," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, Jul. 2016, pp. 31–38.

[73] M. R. Stan and K. Skadron, "Power-aware computing," *Computer*, vol. 36, no. 12, pp. 35–38, Dec. 2003.

[74] R. Ge, X. Feng, and K. W. Cameron, "Performance-constrained distributed dvs scheduling for scientific applications on power-aware clusters," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov. 2005, pp. 34–34.

[75] O. S. Unsal and I. Koren, "System-level power-aware design techniques in real-time systems," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1055–1069, Jul. 2003.

[76] L. Lu, P. Varman, and K. Doshi, "Graduated qos by decomposing bursts: Don't let the tail wag your server," in *2009 29th IEEE International Conference on Distributed Computing Systems*, Jun. 2009, pp. 12–21.

[77] L. Ponciano, A. Brito, L. Sampaio, and F. Brasileiro, "Energy efficient computing through productivity-aware frequency scaling," in *2012 Second International Conference on Cloud and Green Computing*, Nov. 2012, pp. 191–198.

[78] A. Gregory and S. Majumdar, "Energy aware resource management for mapreduce jobs with service level agreements in cloud data centers," in *2016 IEEE International Conference on Computer and Information Technology (CIT)*, Dec. 2016, pp. 568–577.

[79] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, Jan. 1994.

[80] B. H. Ferri and A. A. Ferri, "Reconfiguration of iir filters in response to computer resource availability," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, 2:1–2:25, Oct. 2009.

[81] S. Zilberstein, "Using anytime algorithms in intelligent systems," *AI magazine*, vol. 17, no. 3, p. 73, 1996.

[82] M. Boddy and T. L. Dean, *Solving time-dependent planning problems*. Brown University, Department of Computer Science, 1989.

[83] M. Likhachev, G. J. Gordon, and S. Thrun, "Ara*: Anytime a* with provable bounds on sub-optimality.," in *NIPS*, 2003, pp. 767–774.

[84] T. Kovacshzy, G. Peceli, and G. Simon, "Transients in reconfigurable signal processing channels," *IEEE Transactions on Instrumentation and Measurement*, vol. 50, no. 4, pp. 936–940, Aug. 2001.

[85] V. Valimaki and T. I. Laakso, "Suppression of transients in variable recursive digital filters with a novel and efficient cancellation method," *IEEE Transactions on Signal Processing*, vol. 46, no. 12, pp. 3408–3414, Dec. 1998.

[86] J. Piskorowski, "Dynamic reduction of transients duration in delay-equalized chebyshev filters," in *2007 IEEE Instrumentation Measurement Technology Conference IMTC 2007*, May 2007, pp. 1–5.

[87] Y. Peng, D. Vrancic, and R. Hanus, "Anti-windup, bumpless, and conditioned transfer techniques for pid controllers," *IEEE Control Systems*, vol. 16, no. 4, pp. 48–57, Aug. 1996.

[88] G. Qin, Z. Duan, and G. Chen, "Lq bumpless transfer between two tracking controllers," *International Journal of Control*, vol. 85, no. 10, pp. 1546–1556, 2012.

[89] I. Malloci, L. Hetel, J. Daafouz, C. Iung, and P. Szczepanski, "Bumpless transfer for switched linear systems," *Automatica*, vol. 48, no. 7, pp. 1440–1446, Jul. 2012.

[90] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[91] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013.

[92] Y. Choi, A. Knies, G. Vedaraman, and J. Williamson, "Design and experience: Using the intel itanium2 processor performance monitoring unit to implement feedback optimizations," in *EPIC2 Workshop*, 2002.

[93] E. Duesterwald, C. Cascaval, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *2003 12th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2003, pp. 220–231.

[94] G. Contreras and M. Martonosi, "Power prediction for intel xscale/spl reg/processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, IEEE, 2005, pp. 221–226.

[95] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 325462-061US. Dec. 2016.

[96] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[97] P. J. Denning, "The locality principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, Jul. 2005.

[98] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33, Monterey, California, USA: ACM, 2000, pp. 32–41, ISBN: 1-58113-196-8.

[99] S. Imamura, H. Sasaki, K. Inoue, and D. S. Nikolopoulos, "Power-capped dvfs and thread allocation with ann models on modern numa systems," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct. 2014, pp. 324–331.

[100] S. A. Narayana, "An artificial neural networks based temperature prediction framework for network-on-chip based multicore platform," *arXiv preprint arXiv:1612.04197*, 2016.

[101] R. Hesse and N. E. Jerger, "Improving dvfs in nocs with coherence prediction," in *Proceedings of the 9th International Symposium on Networks-on-Chip*, ACM, 2015, p. 24.

[102] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M.

Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015.

[103] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–14.

[104] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010, pp. 363–374.

[105] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 175–186, ISBN: 978-1-4503-0053-7.

[106] Micron Technologies Inc., *16Gb: x4, x8 TwinDie DDR4 SDRAM Datasheet*, 2015.

[107] Samsung, *8Gb B-die DDR4 SDRAM Datasheet v1.6*, Jan. 2016.

[108] J. Stuecheli, D. Kaseridis, H. C. Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2010, pp. 375–384.

[109] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: ACM, 2010, pp. 83–93, ISBN: 978-1-4503-0053-7.

[110] T. Hamamoto, S. Sugiura, and S. Sawada, "On the retention time distribution of dynamic random access memory (dram)," *IEEE Transactions on Electron Devices*, vol. 45, no. 6, pp. 1300–1309, Jun. 1998.

[111] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in dram (rapid): Software methods for quasi-non-volatile dram," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, Feb. 2006, pp. 155–165.

[112] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the 46th Annual Design Automation*

Conference, ser. DAC '09, New York, NY, USA: ACM, 2009, pp. 664–469, ISBN: 978-1-60558-497-3.

[113] O. Zilberberg, S. Weiss, and S. Toledo, "Phase-change Memory: An Architectural Perspective," *ACM Comput. Surv.*, vol. 45, no. 3, 29:1–29:33, Jul. 2013.

[114] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, 40:40–40:51, Jul. 2013.

[115] D.-J. Shin, S. K. Park, S. M. Kim, and K. H. Park, "Adaptive Page Grouping for Energy Efficiency in Hybrid PRAM-DRAM Main Memory," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, ser. RACS '12, New York, NY, USA: ACM, 2012, pp. 395–402, ISBN: 978-1-4503-1492-3.

[116] M. Lee, V. Gupta, and K. Schwan, "Software-controlled Transparent Management of Heterogeneous Memory Resources in Virtualized Systems," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '13, New York, NY, USA: ACM, 2013, 5:1–5:6, ISBN: 978-1-4503-2103-7.

[117] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, Washington, DC, USA: IEEE Computer Society, 2014, pp. 13–24, ISBN: 978-1-4799-6998-2.

[118] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–12, ISBN: 978-1-4799-6998-2.

[119] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, New York, NY, USA: ACM, 2015, pp. 607–618, ISBN: 978-1-4503-2835-7.

[120] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 231–242.

[121] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11, New York, NY, USA: ACM, 2011, pp. 85–95, ISBN: 978-1-4503-0102-2.

[122] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," *Proceedings of the Workshop on Energy-Efficient Design (WEED)*, Jun. 2013.

[123] H. Seok, Y. Park, K.-W. Park, and K. H. Park, "Efficient Page Caching Algorithm with Prediction and Migration for a Hybrid Main Memory," *SIGAPP Appl. Comput. Rev.*, vol. 11, no. 4, pp. 38–48, Dec. 2011.

[124] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Sep. 2012, pp. 337–344.

[125] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45, Washington, DC, USA: IEEE Computer Society, 2012, pp. 235–246, ISBN: 978-0-7695-4924-8.

[126] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Comput. Archit. Lett.*, vol. 11, no. 2, pp. 61–64, Jul. 2012.

[127] H. Park, S. Yoo, and S. Lee, "Power management of hybrid dram/pram-based main memory," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2011, pp. 59–64.

[128] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, New York, NY, USA: ACM, 2008, pp. 72–81, ISBN: 978-1-60558-282-5.

[129] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, New York, NY, USA: ACM, 2005, pp. 190–200, ISBN: 978-1-59593-056-9.

[130] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, Jul. 2015.

[131] M. Moreau, "Estimating the energy consumption of emerging random access memory technologies," PhD thesis, Institutt for elektronikk og telekommunikasjon, 2013.

[132] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[133] J. Leppäjärvi, "A pragmatic, historically oriented survey on the universality of synchronization primitives," 2008.

[134] M. Giardino, W. Maxwell, B. Ferri, and A. Ferri, "Speculative thread framework for transient management and bumpless transfer in reconfigurable digital filters," in *2018 Annual American Control Conference (ACC)*, Jun. 2018, pp. 3786–3791.

[135] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *Proceedings of the 12th International Conference on Supercomputing*, ser. ICS '98, Melbourne, Australia: ACM, 1998, pp. 77–84, ISBN: 0-89791-998-X.

[136] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31, Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 226–236, ISBN: 1-58113-016-3.

[137] J. Oplinger and M. S. Lam, "Enhancing software reliability with speculative threads," *SIGPLAN Not.*, vol. 37, no. 10, pp. 184–196, Oct. 2002.

[138] J. F. Martnez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 18–29, Oct. 2002.

[139] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified modeling language reference manual, the (2nd edition)*. Pearson Higher Education, 2004, ISBN: 0321245628.

[140] ARM, "big.LITTLE Technology: The Future of Mobile," 2013.

[141] .

[142] S. Natarajan, *Imprecise and approximate computation*. Springer Science & Business Media, 1995, vol. 318.

[143] G. Lencse and S. Répás, "Method for benchmarking single board computers for building a mini supercomputer for simulation of telecommunication systems," in *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*, Jul. 2015, pp. 246–251.

[144] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous multi-processing solution of exynos 5 octa with arm® big. little™ technology,"

[145] A. Butko, F. Bruguier, A. Gamatie, G. Sassatelli, D. Novo, L. Torres, and M. Robert, "Full-system simulation of big.little multicore architecture for performance and energy exploration," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sep. 2016, pp. 201–208.

[146] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic a*: An anytime, replanning algorithm.," in *ICAPS*, 2005, pp. 262–271.

[147] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, Mar. 2016.

[148] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia, "Survey of robot 3d path planning algorithms," *Journal of Control Science and Engineering*, vol. 2016, pp. 1–22, 2016.

[149] A. Gasparetto, P. Boscariol, A. Lanzutti, and R. Vidoni, "Path planning and trajectory planning algorithms: A general overview," in *Motion and Operation Planning of Robotic Systems: Background and Practical Approaches*, G. Carbone and F. Gomez-Bravo, Eds. Cham: Springer International Publishing, 2015, pp. 3–27, ISBN: 978-3-319-14705-5.

[150] B. Gough, *Gnu scientific library reference manual - third edition*, 3rd. Network Theory Ltd., 2009, ISBN: 9780954612078.

[151] J. Borenstein and Y. Koren, "Motion control analysis of a mobile robot," *Journal of dynamic systems, measurement, and control*, vol. 109, no. 2, pp. 73–78, 1987.

[152] D. Gookin, *Programmer's guide to ncurses*. New York, NY, USA: John Wiley and Sons, Inc., 2007, ISBN: 0470107596.

[153] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-power digital design," in *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, Oct. 1994, pp. 8–11.

[154] K. W. Cameron, R. Ge, and X. Feng, "High-performance, power-aware distributed computing for scientific applications," *Computer*, vol. 38, no. 11, pp. 40–47, Nov. 2005.

[155] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. W. Cook, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE Micro*, vol. 20, no. 6, pp. 26–44, Nov. 2000.

[156] C. Bishop, *Pattern recognition and machine learning*. Springer, 2006, ISBN: 978-0-387-31073-2.

[157] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[158] T. M. Mitchell, *Machine learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997, ISBN: 9780070428072.

[159] C. J. C. H. Watkins, "Learning from delayed rewards," PhD thesis, King's College, Cambridge, UK, May 1989.

[160] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1, 1992.

[161] R. S. Sutton, A. G. Barto, and R. J. Williams, "Reinforcement learning is direct adaptive optimal control," *IEEE Control Systems Magazine*, vol. 12, no. 2, pp. 19–22, Apr. 1992.

[162] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*, 1st. Cambridge, MA, USA: MIT Press, 1998, ISBN: 0262193981.

[163] E. Even-Dar and Y. Mansour, "Learning rates for q-learning," *J. Mach. Learn. Res.*, vol. 5, pp. 1–25, Dec. 2004.

[164] G. A. Paleologo, L. Benini, A. Bogliolo, and G. D. Micheli, "Policy optimization for dynamic power management," in *IN DESIGN AUTOMATION CONFERENCE*, ACM Press, 1998, pp. 182–187.

[165] L. Benini, A. Bogliolo, G. A. Paleologo, and G. D. Micheli, "Policy optimization for dynamic power management," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, vol. 18, pp. 813–833, 1998.

[166] A. Galindo-Serrano and L. Giupponi, "Distributed q-learning for interference control in ofdma-based femtocell networks," in *2010 IEEE 71st Vehicular Technology Conference*, May 2010, pp. 1–5.

[167] M. Bennis and D. Niyato, "A q-learning based approach to interference avoidance in self-organized femtocell networks," in *2010 IEEE Globecom Workshops*, Dec. 2010, pp. 706–710.

[168] N. Mastronarde and M. van der Schaar, "Fast reinforcement learning for energy-efficient wireless communication," *IEEE Transactions on Signal Processing*, vol. 59, no. 12, pp. 6262–6266, Dec. 2011.

[169] R. C. Hsu, C. Liu, K. Wang, and W. Lee, "Qos-aware power management for energy harvesting wireless sensor network utilizing reinforcement learning," in *2009 International Conference on Computational Science and Engineering*, vol. 2, Aug. 2009, pp. 537–542.

[170] Q. Wei, D. Liu, and G. Shi, "A novel dual iterative$q$-learning method for optimal battery management in smart residential environments," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 4, pp. 2509–2518, Apr. 2015.

[171] C. Harris and V. Cahill, "Exploiting user behaviour for context-aware power management," in *WiMob'2005), IEEE International Conference on Wireless And Mobile Computing, Networking And Communications, 2005.*, vol. 4, Aug. 2005, 122–130 Vol. 4.

[172] R. Xiong, J. Cao, and Q. Yu, "Reinforcement learning-based real-time power management for hybrid energy storage system in the plug-in hybrid electric vehicle," *Applied Energy*, vol. 211, pp. 538–548, 2018.

[173] S. Yue, D. Zhu, Y. Wang, and M. Pedram, "Reinforcement learning based dynamic power management with a hybrid power supply," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Sep. 2012, pp. 81–86.

[174] Y. Tan, W. Liu, and Q. Qiu, "Adaptive power management using reinforcement learning," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD '09, San Jose, California: ACM, 2009, pp. 461–467, ISBN: 978-1-60558-800-1.

[175] W. Liu, Y. Tan, and Q. Qiu, "Enhanced q-learning algorithm for dynamic power management with performance constraint," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, Mar. 2010, pp. 602–605.

[176] Y. Wang, Q. Xie, A. Ammari, and M. Pedram, "Deriving a near-optimal power management policy using model-free reinforcement learning and bayesian classification," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11, San Diego, California: ACM, 2011, pp. 41–46, ISBN: 978-1-4503-0636-2.

[177] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: A reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th International Conference on Autonomic Computing*, ser. ICAC '09, Barcelona, Spain: ACM, 2009, pp. 137–146, ISBN: 978-1-60558-564-2.

[178] J. Rao, X. Bu, C.-Z. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," Jul. 2011, pp. 45–54.

[179] G. Tesauro, R. Das, H. Chan, J. Kephart, D. Levine, F. Rawson, and C. Lefurgy, "Managing power consumption and performance of computing systems using reinforcement learning," in *Advances in Neural Information Processing Systems*, 2008, pp. 1497–1504.

[180] X. Bu, J. Rao, and C. Xu, "A model-free learning approach for coordinated configuration of virtual machines and appliances," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, Jul. 2011, pp. 12–21.

[181] Z. Zhang, Q. Guan, and S. Fu, "An adaptive power management framework for autonomic resource configuration in cloud computing infrastructures," in *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, Dec. 2012, pp. 51–60.

[182] J. F. Martinez and E. Ipek, "Dynamic multicore resource management: A machine learning approach," *IEEE Micro*, vol. 29, no. 5, pp. 8–17, Sep. 2009.

[183] R. Ye and Q. Xu, "Learning-based power management for multicore processors via idle period manipulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 7, pp. 1043–1055, Jul. 2014.

[184] H. Shen, J. Lu, and Q. Qiu, "Learning based dvfs for simultaneous temperature, performance and energy management," in *Thirteenth International Symposium on Quality Electronic Design (ISQED)*, Mar. 2012, pp. 747–754.

[185] H. Shen, Y. Tan, J. Lu, Q. Wu, and Q. Qiu, "Achieving autonomous power management using reinforcement learning," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 2, 24:1–24:32, Apr. 2013.

[186] Y. Ge and Q. Qiu, "Dynamic thermal management for multimedia applications using machine learning," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2011, pp. 95–100.

VITA

Michael Giardino was born in New Orleans, LA. He attended Lusher Elementary and Ben Franklin High School in New Orleans, LA before graduating from Saint Stanislaus College Prep in Bay Saint Louis, MS.

Michael attended the University of New Orleans as an undergraduate in electrical engineering. While at UNO, Michael completed two research experiences for undergraduates (REUs) under the supervision of Dr. Dimitrios Charalampidis. In addition, he was a member of the first place team of the Region 5 IEEE Robot Competition in 2011, received the award for Outstanding Sophomore in the College of Engineering, and the Iverson scholarship.

After graduating from the University of New Orleans with the bachelor's degree in Electrical Engineering, Michael enrolled at the Georgia Institute of Technology in Fall 2011. At Georgia Tech, Michael was awarded the Institute's Presidential Fellowship (2011-2016) as well as ECE's Outstanding Graduate Teaching Assistant in Spring 2014. He interned at Intel in Chandler, AZ with the Enterprise and Big Data group under the supervision of Dr. Kshitij Doshi and Arakere Ramesh. Michael graduated with the master's degree in Electrical and Computer Engineering in 2013 and the PhD in 2019.

Michael currently lives in northern Italy with his wife Rachel and their dog Rufus. Michael and Rachel are expecting a son in February 2019.