UNIVERSITY OF CALIFORNIA,
IRVINE

Adaptive Resource Management for Mobile Multiprocessors through Computational Self-Awareness

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Bryan Donyanavard

Dissertation Committee:
Nikil Dutt, Chair
Tony Givargis
Fadi Kurdahi
Alexandru Nicolau

2019

DEDICATION

for Reggie

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Page

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank:

Everyone at UCI, friends and colleagues, that had an impact on me during this significant period of my life. Hossein, Majid, Tiago, and Zhi for figuring it all out ahead of me. Sajjad, Roger, and Maral for sharing the pain. Hamid for the sanity. Nik for being patient. Amir for the advice. Armin and Flo for making sure I don't slack in my old age. All of the DRG members past and present I shared this experience with, as well as my collaborators abroad.

Nik, again, for being a mentor and role model in all respects.

My family for their varying ranges of tolerance to support throughout my (admittedly long) educational journey. Mom, Dad, Claire, and Jessi. Particularly Grandma, Grandpa, Ana, and Aga for the good genes. And Auntie Debbie, the only one who might understand.

My friends from before I was such a huge success for (literally) putting clothes on my back and drinks in my cup. Saugus boys, SB boys, 272 boys, and #anarchY.

Finally, my work would not have been possible without funding from the Donald Bren School of Information and Computer Sciences as well as the NSF (grant CCF-1704859), or permission from the ACM and IEEE to include content from my previously published work in [72], [21], and [20].

CURRICULUM VITAE

Bryan Donyanavard

EDUCATION

Doctor of Philosophy in Computer Science **2019**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Engineering **2010**
Univeristy of California, Santa Barbara *Santa Barbara, CA*

Bachelor of Science in Computer Engineering **2008**
Univeristy of California, Santa Barbara *Santa Barbara, CA*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2019**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Instructor **2018**
University of California, Irvine *Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

Design Methodology for Responsive and Robust MIMO Control of Heterogeneous Multicores **2018**
IEEE Transactions on Multi-Scale Computing Systems

ShaVe-ICE: Sharing Distributed Virtualized SPMs in Many-Core Embedded Systems **2018**
ACM Transactions on Embedded Computing Systems

Automatic Management of Software Programmable Memories in Many-core Architectures **2016**
IET Computers & Digital Techniques

SPMPool: Runtime SPM Management for Memory-Intensive Applications in Embedded Many-Cores **2016**
ACM Transactions on Embedded Computing Systems

REFEREED CONFERENCE PUBLICATIONS

SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management **Oct 2019**
International Symposium on Microarchitecture

Workload Characterization for Memory Management in Emerging Embedded Platforms **Sep 2019**
International Embedded Systems Symposium

Exploring Hybrid Memory Caches in Chip Multiprocessors **Jul 2018**
International Symposium on Reconfigurable Communication-centric Systems-on-Chip

SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management **Mar 2018**
International Conference on Architectural Support for Programming Languages and Operating Systems

Gain Scheduled Control for Nonlinear Power Management in CMPs **Mar 2018**
Design, Automation & Test in Europe Conference & Exhibition

PoliCym: Rapid Prototyping of Resource Management Policies for HMPs **Oct 2017**
International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype

SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores **Oct 2016**
International Conference on Hardware/Software Codesign and System Synthesis

On Detecting and Using Memory Phases in Multimedia Systems **Oct 2016**
ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia

Energy-Aware Real-Time Face Recognition System on Mobile CPU-GPU Platform **Sep 2010**
European Conference on Computer Vision

ABSTRACT OF THE DISSERTATION

Adaptive Resource Management for Mobile Multiprocessors through Computational Self-Awareness

By

Bryan Donyanavard

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Nikil Dutt, Chair

Runtime resource management for many-core systems is increasingly complex. The complexity can be due to diverse workload characteristics with conflicting demands or limited shared resources such as memory bandwidth and power. Resource management strategies for many-core systems must distribute shared resource(s) appropriately across workloads, while coordinating the high-level system goals at runtime in a scalable and robust manner.

To address the complexity of dynamic resource management in many-core systems, state-of-the-art techniques that use heuristics have been proposed. These methods lack the formalism in providing robustness against unexpected runtime behavior. One of the common solutions for this problem is to deploy classical control approaches with bounds and formal guarantees. Traditional control theoretic methods lack the ability to adapt to (1) changing goals at runtime (i.e., *self-adaptivity*), and (2) changing dynamics of the modeled system (i.e., *self-optimization*).

In this thesis, I explore adaptive resource management techniques that provide self-optimization and self-adaptivity by employing principles of computational self-awareness, specifically *reflection*. By supporting these self-awareness properties, the system will reason about the actions it takes by considering the significance of competing objectives, user requirements, and operating conditions while executing unpredictable workloads.

Chapter 1

Introduction

Battery powered-devices are the most ubiquitous computers in the world. Users expect the devices to support high performance applications running on same device, sometimes at the same time. The devices support a wide range of applications, from interactive maps and navigation, to web browsers and email clients. In order to meet the performance demands of the complex workloads, increasingly powerful hardware platforms are being deployed in battery-powered devices. These platforms include a number of configurable knobs that allow for a tradeoff between power and performance, e.g., dynamic voltage and frequency scaling (DVFS), core gating, idle cycle injection, etc. These knobs can be set and modified at runtime based on workload demands and system constraints. Heterogeneous manycore processors (HMPs) have extended this principle of dynamic power-performance tradeoffs by incorporating single-ISA, architecturally differentiated cores on a single processor, with each of the cores containing a number of independent tradeoff knobs. All of these configurable knobs allow for a large range of potential tradeoffs. However, with such a large number of possible configurations, HMPs require intelligent runtime management in order to achieve application goals for complex workloads while considering system constraints. Additionally, the knobs may be interdependent, so configuration decisions must be coordinated. In

this thesis, we explore the use of computational self-awareness to address challenges of adaptive resource management in mobile multiprocessors.

1.1 Background and Related Work

Self-aware computing is a new paradigm that does not strictly introduce new research concepts, but unifies overlapping research efforts in disparate disciplines [49]. The concept of self-awareness from psychology has inspired research in autonomous systems and neuroscience, and existing research in fields such as adaptive control theory support properties of self-awareness. This thesis employs principles of self-awareness in order to make the design, maintenance and operation of complex, heterogeneous systems adaptive, autonomous, and highly efficient.

1.1.1 Computational Self-Awareness

Computational self-awareness is the ability of a computing system to recognize its own state, possible actions and the result of these actions on itself, its operational goals, and its environment, thereby empowering the system to become autonomous [40]. An infrastructure for system introspection and reflective behavior forms the foundation of self-aware systems.

Reflection

Reflection can be defined as *the capability of a system to reason about itself and act upon this information* [83]. A reflective system can achieve this by maintaining a representation of itself (i.e., a self-model) within the underlying system, which is used for reasoning. Reflection is a key property of self-awareness. Reflection enables decisions to be made based on both *past* observations, as well as predictions made from past observations. Reflection and prediction involve two types of

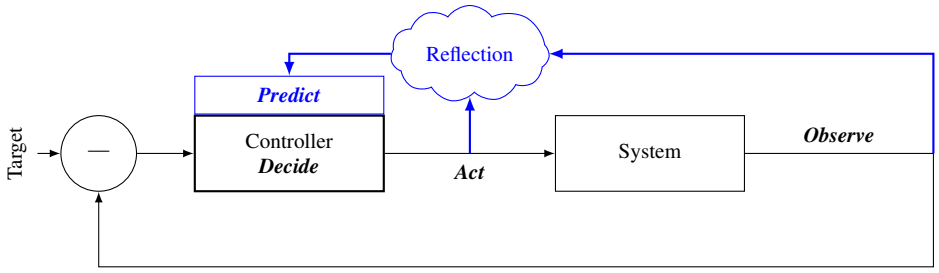


Figure 1.1: Feedback loop overview. The bottom part of the figure represents a simple observe-decide-act loop. The top part (in blue) adds the reflection mechanism to this loop, enabling predictions for smart decision making.

models: (1) a self-model of the subsystem(s) under control, and (2) models of other policies that may impact the decision-making process. Predictions consider *future* actions, or events that may occur before the next decision, enabling "what-if" exploration of alternatives. Such actions may be triggered by other resource managers running with a shorter period than the decision loop. The top half of Figure 1.1 (in blue) shows prediction enabled through reflection that can be utilized in the decision making process of a feedback loop. The main goal of the prediction model is to estimate system behavior based on potential actuation decisions. This type of prediction is most often performed using linear regression-based models [61, 70, 1, 81] due to their simplicity, while others employ a binning-based approach in which metrics sensed at runtime are used to classify workloads into categories [50, 19].

1.1.2 Closed-loop Resource Management in Mobile Systems

Runtime resource management for many-core systems is increasingly challenging due to the complex interaction of: i) integrating hundreds of (heterogeneous) cores and uncore components on a single chip, ii) limited amount of system resources (e.g., power, cores, interconnects), iii) diverse workload characteristics with conflicting constraints and demands, and iv) increasing pressure on shared system resources from data-intensive workloads. As system size and capability scale, designers face a large space of configuration parameters controlled by actuation knobs, which in turn generate a very large number of cross-layer actuation combinations [105]. Making runtime

decisions to configure knobs in order to achieve a simple goal (e.g., maximize performance) can be challenging. That challenge is exacerbated when considering a goal that may change throughout runtime, and consist of conflicting objectives (e.g., maximize performance while minimizing power consumption). Additionally, ubiquitous mobile devices are expected to be general-purpose, supporting any combination of applications (i.e., workloads) desired by users, often without any prior knowledge of the workload.

Designers face a large space of configuration parameters that often are controlled by a multiple actuation knobs, which in turn generate a very large number of cross-layer actuation configurations. For instance, Zhang and Hoffman [105] show that for an 8-core Intel Xeon processor, combining only a handful of actuation knobs (such as clock frequency and Hyperthreading levels) generates over 1000 different actuation configurations; they use binary search to efficiently explore the configuration space for achieving a *single* goal: cap the Thermal Design Power (TDP) while maximizing performance.

Closed-loop systems have been used extensively to improve the state of a system by configuring knobs in order to achieve a goal. Closed-loop systems traditionally deploy an *Observe*, *Decide* and *Act* (ODA) feedback loop (lower half (in black) of Figure 1.1) to determine the system configuration. In an ODA loop, the observed behavior of the system is compared to the target behavior, and the discrepancy is fed to the controller for decision making. The controller invokes actions based on the result of the *Decide* stage.

Resource management approaches in the literature can be classified into three main classes: (1) hueristic-based-approaches [67, 31, 56, 28, 90, 88, 29, 39, 18, 27, 7, 22, 23, 17, 42, 105, 3, 94, 96, 103, 53, 85, 95, 62, 19, 24, 16, 25, 14, 13, 8, 86, 87, 65, 64, 66], (2) control-theory-based approaches [55, 38, 73, 74, 58, 102, 100, 26, 54, 63, 43, 36, 80, 44, 30, 68], and (3) stochastic-/machine-learning-based approaches [10, 9, 59, 12, 46, 4]. There exist some proposed solutions that incorporate aspects of multiple categories, e.g., there are a number of works that use learning to build predictive models, and use heuristics to make runtime decisions based on the predictive

models [7, 22, 23, 19]. Recent work has combined aspects of machine learning and feedback control [59]. In addition, there have been efforts to enable coordinated management in computer systems in various ways [7, 71, 10, 92, 41, 99, 89, 93, 11, 23, 25, 24, 16, 14, 84, 47, 15, 69]. These works coordinate and control multiple goals and actuators in a non-conflicting manner by adding an ad-hoc component or hierarchy to a controller.

In this thesis, I demonstrate the effectiveness of computational self-awareness in adaptive resource management for mobile processors. The self-aware resource managers discussed are implemented using classical and hierarchical control.

1.2 Challenges and Contributions

In this thesis, I focus on the following challenges in the context of mobile resource management:

1. **Efficiency:** How can we design lightweight, yet responsive controllers?
2. **Coordination:** How do we control and coordinate (possibly conflicting) actuations while tracking multiple objectives simultaneously (e.g., frame rate and chip power)?
3. **Scalability:** How can we properly design control hierarchies to manage large and complex systems?
4. **Autonomy:** How can controllers automatically respond to abrupt runtime changes in objectives (e.g., changing the priority of objectives)?

(1) Efficiency is a base requirement of any runtime resource management policy: runtime decisions must be made with justifiable overhead. (2) Coordination and (3) scalability apply to both the number of control knobs and system configurations, as well as the amount of system resources. (4) Autonomy is crucial: system goals need to be adaptively managed and objectives holistically

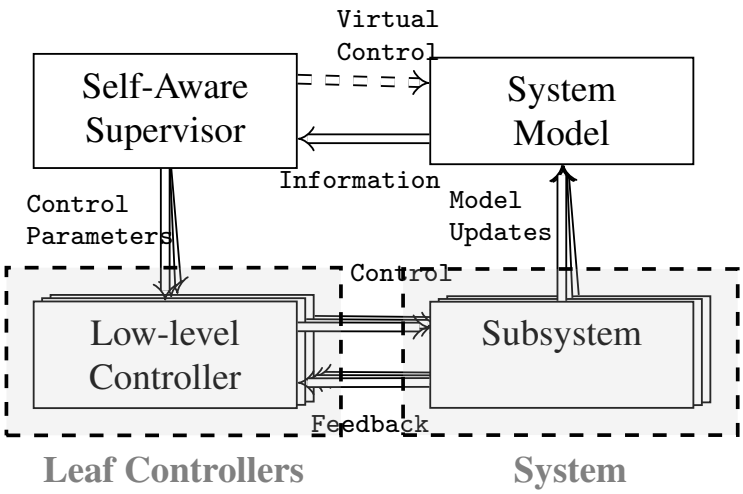


Figure 1.2: Hierarchical resource manager.

coordinated in the face of varying conditions over time. Present solutions [69, 59] suggest that hierarchy is essential when considering the challenges in runtime resource management of modern mobile systems.

Figure 1.2 shows the hierarchical approach to resource management I propose. Deploying multiple low-level controllers allows for system decomposition for scalability. A high-level supervisor provides coordination and self-awareness by guiding the low-level controllers. Using this resource management hierarchy, I will provide self-awareness properties to address the identified challenges.

Chapter 2

Self-Adaptive Supervision

2.1 Self-Adaptivity

Self-adaptivity is the ability of a system to adjust to changes in goals due to external stimuli, i.e., an effect that is not within the scope of control. For example, if a system experiences a thermal event due to increased ambient temperature during a computational sprint and enters an unsafe state, a self-adaptive manager will have the ability to modify the goal from maximizing performance to minimizing temperature.

2.2 Motivation

Controllers may behave non-optimally, or even detrimentally, in meeting a shared goal without knowledge of the presence or behavior of seemingly orthogonal controllers [75, 7, 93, 26, 24, 14]. Consider the MIMO controller in Figure 2.1 that controls a single-core system with two control inputs ($u(t)$) and interdependent measured outputs ($y(t)$) [68]. The controller tracks two objectives (frames per second, or FPS, and power consumption) by controlling two actuators (operating

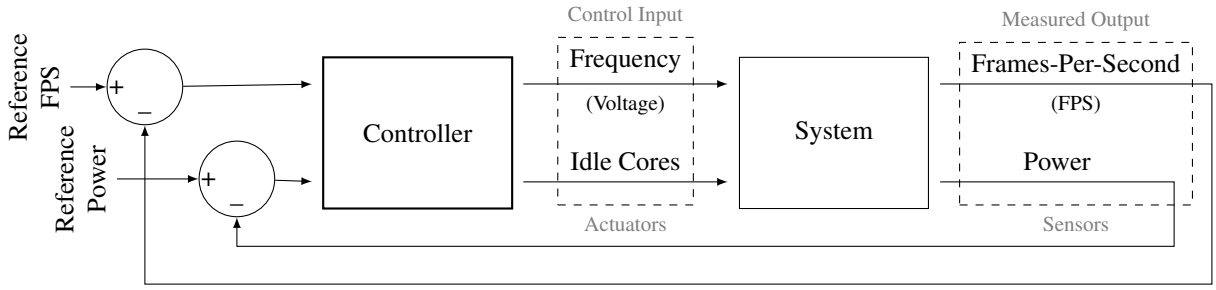


Figure 2.1: Basic 2×2 MIMO for single-core system. Clock frequency and cache size are used as control inputs. FPS and power are measured outputs that are compared with reference (i.e., target) values.

frequency and cache size). We implement the MIMO using a Linear Quadratic Gaussian (LQG) controller [82] similarly to [68]:

$$x(t+1) = A \times x(t) + B \times u(t) \tag{2.1}$$

$$y(t) = C \times x(t) + D \times u(t) \tag{2.2}$$

where x is the system state, y is the measured output vector, and u is the control input vector.[1]

LQG control allows us to specify 1) the relative sensitivity of a system to control inputs, and 2) the relative priority of measured outputs. This is done using 1) a weighted Tracking Error Cost matrix (Q) and 2) a Control Effort Cost matrix (R). The weights are specified during the design of the controller. While this is convenient for achieving a fixed goal, it can be problematic for goals that change over time (e.g., minimizing power consumption before a predicted thermal emergency).

The controller must choose an appropriate trade-off when we cannot achieve both desirable performance and power concurrently. Unfortunately, classical MIMOs fix control weights at design time, and thus *cannot* perform *runtime* tradeoffs that require changing output priorities. Even with

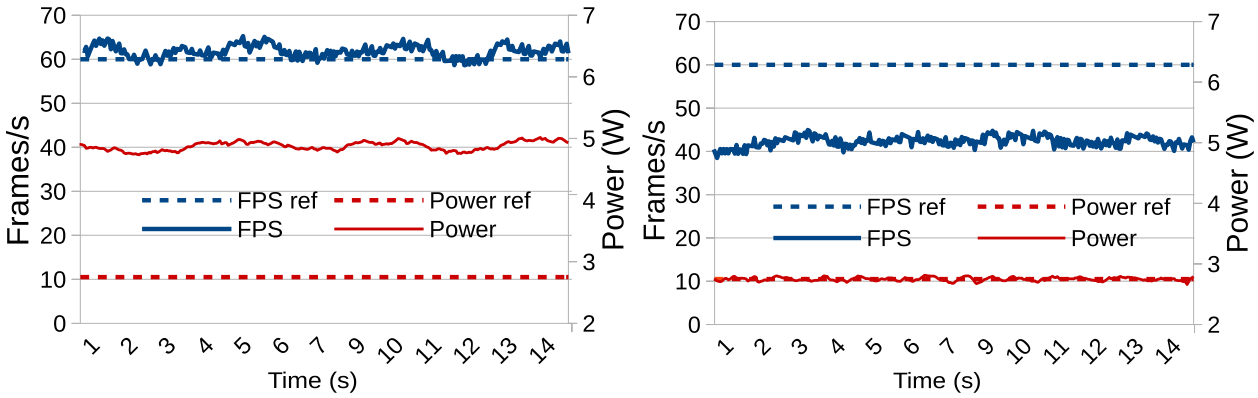
[1] We interchangeably use the terms (*measured output* and *sensor*), as well as the terms (*control input* and *actuator*), as shown in Figure 2.1.

constant *reference values*, i.e., desired output values, unpredictable disturbances (e.g., changing workload and operating conditions) may cause the reference values to become unachievable. It is also plausible for the reference values themselves to change dynamically at runtime with system state and operating conditions (e.g., a thermal event).

Let us now consider a more complex scenario: a multi-threaded application running on Linux, executing on a mobile processor, where the system needs to track both the performance (FPS) and power simultaneously. Figure 2.1 shows the 2×2 MIMO model for this system with operating frequency and the number of active cores as control inputs, and FPS and power as measured outputs.

Both the FPS and power reference values are trackable individually, but not jointly. We implement and compare two different MIMO controllers in Linux to show the effect of competing objectives. One controller prioritizes FPS, and the other prioritizes power. Figure 2.2 shows the power and performance (in FPS) achieved by each MIMO controller using typical reference values for a mobile device: 60 FPS and 5 Watts. The application is `x264`, and the mobile processor consists of an ARM Cortex-A15 quad-core cluster. Each MIMO controller is designed with a different Q matrix to prioritize either FPS or power: Figure 2.2a's controller favors FPS over power by a ratio of 30:1 (i.e., only 1% deviation from the FPS reference is acceptable for a 30% deviation from the power reference), while Figure 2.2b uses a ratio of 1:30. We observe that neither controller is able to manage changing system goals. Thus, there is a need for a supervisor to autonomously orchestrate the system while considering the significance of competing objectives, user requirements, and operating conditions.

The use of supervisory control presents at least three additional advantages over conventional controllers. First, fully-distributed MIMO or SISO controllers *cannot* address system-wide goals such as power capping. Second, conventional controllers *cannot* model actuation effects that require system-wide perspective, such as task migration. Third, classical control theory *cannot* address problems requiring optimization (e.g., minimizing an objective function) alone [45, 68].



(a) FPS-oriented controller.

(b) Power-oriented controller.

Figure 2.2: x264 running on a quad-core cluster controlled by 2×2 MIMOs with different output priorities.

2.3 Supervisory Control Theory

Supervisory control utilizes modular decomposition to mitigate the complexity of control problems, enabling automatic control of many individual controllers or control loops. Supervisory control theory (SCT) [76] benefits from formal synthesis methods to define principal control properties for *controllability* and *observability*. The emphasis on formal methods in addition to *modularity* leads to *hierarchical consistency* and *non-conflicting* properties.

SCT solves complex synthesis problems by breaking them into small-scale sub-problems, known as modular synthesis. The results of modular synthesis characterize the conditions under which decomposition is effective. In particular, results identify whether a valid decomposition exists. A decomposition is valid if the solutions to sub-problems combine to solve the original problem, and the resulting composite supervisors are non-blocking and minimally restrictive. Decomposition also adds robustness to the design because nonlinearities in the supervisor do *not* directly affect the system dynamics. This horizontal decomposition in sub-problems allows us to divide the overall system into several smaller subsystems which are controlled by individual controllers.

Figure 1.2 illustrates how a supervisory control structure can hierarchically manage feedback control loops. As shown in the figure, supervision is vertically decomposed into tasks performed at

different levels of abstraction [91]. The supervisory controller is designed to control the high-level *system model*, which represents an abstraction of the system. The *subsystems* compose the pre-existing *system* that does *not* meet the given specifications without the aid of a controller or a supervisor. The *information* channel provides information about the updates in the high-level model to the supervisory controller. Due to the fact that the system model is an abstract model, the controlling channel is an indirect *virtual control* channel. In other words, the control decisions of the supervisory controller will be implemented by controlling the *low-level controller(s)* through *control parameters*. Consequently, the low-level controller(s) can control one or multiple subsystems using the *control* channel and gather information via *feedback*. The changes in the subsystems can trigger *model updates* in the state of the high-level system model. These updates reflect the results of low-level controllers' controlling actions.

The scheme of Figure 1.2 describes the division of supervision into high-level management and low-level operational supervision. Virtual control exercised via the high-level control channel can be implemented by modifying control parameters to adaptively coordinate the low-level controllers, e.g., by adjusting their operating modes according to the system goal. The combination of horizontal and vertical decomposition enables us to not only physical divide the system into subsystems, but also to logically divide the sub-problems in any appropriate way, e.g., due to varying epochs (control invocation period) or scope. The important requirement of this hierarchical control scheme is control consistency and hierarchical consistency between the high-level model and the low-level system, as defined in the standard Ramadge-Wonham control mechanism [91]. For a detailed description of SCT, we refer the reader to [76, 77, 5, 91].

2.3.1 Self-Adaptivity via Supervision

Supervisory controllers are preferable to *adaptive (self-tuning) controllers* for complex system control due to their ability to integrate **logic** with **continuous dynamics**. Specifically, supervisory

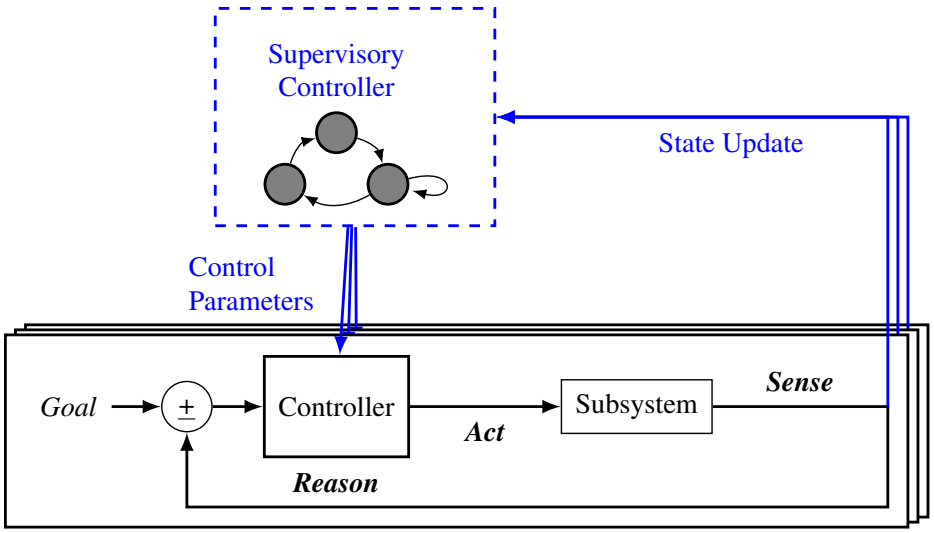


Figure 2.3: Self-Adaptivity via Supervisory Control Theory.

control has two key properties: i) rapid adaptation in response to abrupt changes in management policy [35], and ii) low computational complexity by computing control parameters for different policies offline.

Figure 2.3 depicts the two mechanisms that enable SCT-based management via low-level controllers: **gain scheduling** and **dynamic references**. Gain scheduling is a nonlinear control technique that uses a set of linear controllers predesigned for different operating regions. Gain scheduling enables the appropriate linear controller based on runtime observations [48]. Scheduling is implemented by switching between sets of control parameters, i.e., $A_1 \rightarrow A_2$, $B_1 \rightarrow B_2$, $C_1 \rightarrow C_2$, and $D_1 \rightarrow D_2$ in Equations 2.1 and 2.2. In this case, the *controller gains* are the values of the control parameters A, B, C, and D. Gains are useful to change objectives at runtime in response to abrupt and sudden changes in management policy. In LQG controllers, this is done by changing priorities of outputs using the Q and R matrices (Section 2.2). This is what we call the Hierarchical Control structure, in which local controllers solve specified tasks while the higher-level supervisory controller coordinates the global objective function. In this structure, the supervisory controller receives information from the plant (e.g., the presence of a thermal emergency) or the user/application (e.g., new QoS reference value), and steers the system towards the desired policy

using its design logic and high-level model. Thanks to its top-level perspective, the supervisor can update reference values for each low-level controller to either optimize for a certain goal (e.g., getting to the optimum energy-efficient point) or manage resource allocation (e.g., allocating power budget to different cores).

2.4 Case Study: On-chip Resource Management

In this section, supervisors are designed and evaluated for two different hierarchical resource management use-cases. The first use-case requires management of QoS under a power budget on a HMP. The resource manager (SPECTR) consists of a supervisor that guides low-level classical controllers to configure core operating frequency and number of active cores for each core cluster. The second use-case requires management of QoS under a power budget for a CMP. The resource manager (SOSA) consists of a supervisor that guides low-level reinforcement learners to configure per-core operating frequency, as well as make requests for task migration.

2.4.1 Hierarchical System Architecture

Figure 2.4 depicts a high-level view of SPECTR for many-core system resource management. Either the user or the system software may specify *Variable Goals and Policies*. The *Supervisory Controller* aims to meet system goals by managing the low-level controllers. High-level decisions are made based on the feedback given by the *High-level Plant Model*, which provides an abstraction of the entire system. Various types of *Classic Controllers*, such as PID or state-space controllers, can be used to implement each low-level controller based on the target of each subsystem. The flexibility to incorporate any pre-verified off-the-shelf controllers without the need for system-wide verification is essential for the modularity of this approach. The supervisor provides parameters such as output references or gain values to each low-level controller during runtime according to

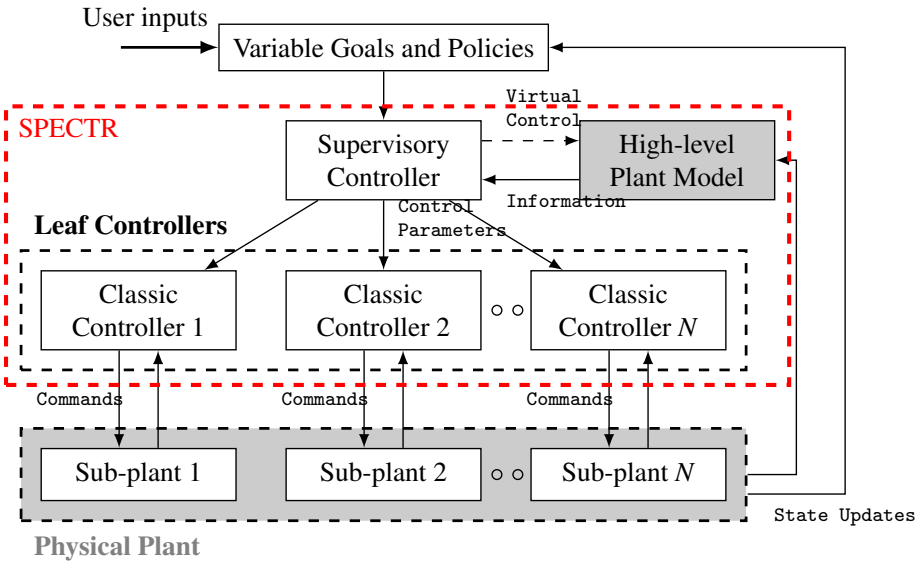


Figure 2.4: SPECTR overview.

the system policy. Low-level controller subsystems update the high-level model to maintain global system state, and potentially trigger the supervisory controller to take action. The high-level model can be designed in various fashions (e.g., rule-based or estimator-based [77][35][60]) to track the system state and provide the supervisor with guidelines. We illustrate the steps for designing a supervisory controller using the following experimental case study in which SCT is deployed on a real HMP platform, and we then outline the entire design flow from modeling of the high-level plant to generating the supervisory controller.

2.4.2 SPECTR Resource Manager

Figure 2.5 shows an overview of our experimental setup. We target the Exynos platform [32], which contains an HMP with two quad-core clusters: the **Big** core cluster provides high-performance out-of-order cores, while the **Little** core cluster provides low-power in-order cores. Memory is shared across all cores, so application threads can transparently execute on any core in any cluster. We consider a typical mobile scenario in which a single foreground application (the *QoS appli-*

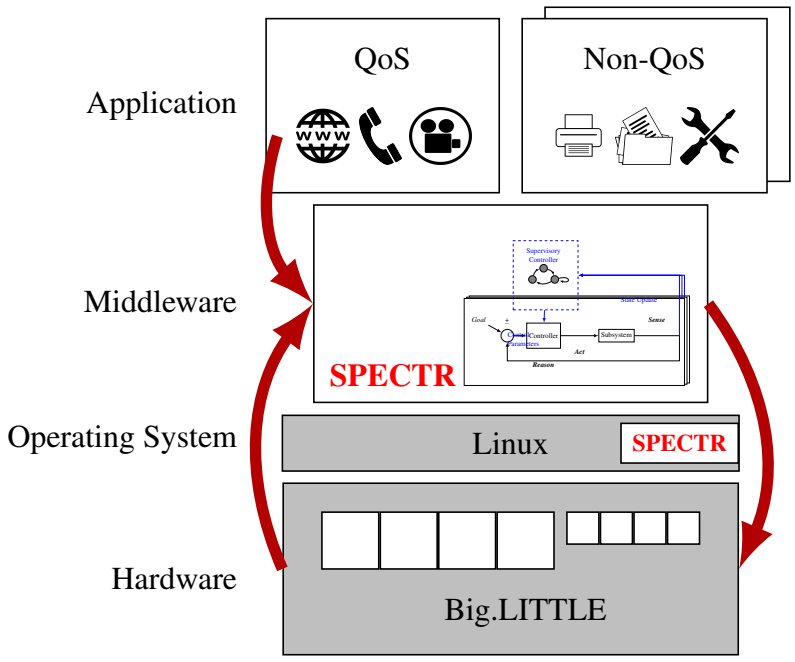


Figure 2.5: SPECTR implementation on the Exynos HMP with two heterogeneous quad-core clusters. Representing a typical mobile scenario with a single foreground application running concurrently with many background applications.

cation) is running concurrently with many background applications (the *Non-QoS applications*). This mimics a typical mobile use-case in which gaming or media processing is performed in the foreground in conjunction with background email or social media syncs.

The system goals are twofold: i) meet the QoS requirement of the foreground application while minimizing its energy consumption; and ii) ensure the total system power always remains below the Thermal Design Power (TDP).

The **subsystems** are the two heterogeneous quad-core (*Big* and *Little*) clusters. Each cluster has two actuators: one actuator to set the operating frequency (F_{next}) and associated voltage of the cluster; and one to set the number of active cores (AC_{next}) on the cluster. We measure the power consumption (P_{curr}) of each cluster, and simultaneously monitor the QoS performance (QoS_{curr}) of the designated application to compare it to the required QoS (QoS_{ref}).[2]

[2] The Exynos platform provides only per-cluster power sensors and DVFS; hence our use of cluster-level sensors and actuators.

Supervisory control commands guide the **low-level MIMO controllers** in Figure 2.5 to determine the number of active cores and the core operating frequency within each cluster.

Supervisory control minimizes the system-wide power consumption while maintaining QoS. In our scenario, the QoS application runs only on the Big cluster, and the supervisor determines whether and how to adjust the cluster's power budget based on QoS measurements.

Gain scheduling is used to switch the priority objective of the low-level controllers. We define two sets of gains for this case-study: 1) *QoS-based* gains are tuned to ensure that the QoS application can meet the performance reference value, and 2) *Power-based* gains are tuned to limit the power consumption while possibly sacrificing some performance if the system is exceeding the power budget threshold.

2.4.3 SPECTR Experimental Evaluation

We compare SPECTR with three alternative resource managers. The first two managers use two uncoordinated 2×2 MIMOs, one for each cluster: *MM-Pow* uses power-oriented gains, and *MM-Perf* uses performance-oriented gains. These fixed MIMO controllers act as representatives of a state-of-the-art solution, as presented in [68], one prioritizing power and the other prioritizing performance. The third manager consists of a single full-system controller (*FS*): a system-wide 4×2 MIMO with individual control inputs for each cluster. *FS* uses power-oriented gains and its measured outputs are chip power and QoS. This single system-wide MIMO acts as a representative for [105], maximizing performance under a power cap.

We analyze an execution scenario that consists of three different phases of execution:

1. *Safe Phase*: In this phase, only the QoS application executes (with an achievable QoS reference within the TDP). The goal is to meet QoS and minimize power consumption.

2. *Emergency Phase*: In this phase, the QoS reference remains the same as that in the Safe Phase while the power envelope is reduced (emulating a thermal emergency). The goal is to adapt to the change in reference power while maintaining QoS (if possible).

3. *Workload Disturbance Phase*: In this phase, the power envelope returns to TDP and background tasks are added (to induce interference from other tasks). The goal is to meet the QoS reference value without exceeding the power envelope.

This execution scenario with three different phases allows us to evaluate how SPECTR compares with state-of-the-art resource managers when facing workload variation and system-wide changes in state (e.g., thermal emergency) and goals.

Evaluated resource manager configurations

We generate stable low-level controllers for each resource manager using the Matlab System Identification Toolbox [57].[3] We use the Control Effort Cost matrix (R) to prioritize changing clock frequency over number of cores at a ratio of 2:1, as frequency is a finer-grained and lower-overhead actuator than core count. We generate training data by executing an in-house microbenchmark and varying control inputs in the format of a staircase test (i.e., a sine wave), both with single-input variation and all-input variation. The micro-benchmark consists of a sequence of independent multiply-accumulate operations performed over both sequentially and randomly accessed memory locations, thus yielding various levels of instruction-level and memory-level parallelism. The range of exercised behavior resembles or exceeds the variation we expect to see in typical mobile workloads, which is the target application domain of our case studies.

[3] We generate the models with a stability focus. All systems are stable according to Robust Stability Analysis. We use Uncertainty Guardbands of 50% for QoS and 30% for power, as in [68].

Experimental setup

We perform our evaluations on the ARM big.LITTLE [2] based Exynos SoC (ODROID-XU3 board [32]) as described in our case study (Figure 2.5). We implement a Linux userspace daemon process that invokes the low-level controllers every 50*ms*. When evaluating SPECTR, the daemon invokes the supervisor every 100*ms*. We use ARM's Performance Monitor Unit (PMU) and per-cluster power sensors for the performance and power measurements required by the resource managers. The userspace daemon also implements the Heartbeats API [37] monitor to measure QoS. By periodically issuing *heartbeats*, the application informs the system about its current performance. The user provides a performance reference value using the Heartbeats API.

To evaluate the resource managers, we use the following benchmarks from the PARSEC benchmark suite [6] as QoS applications (i.e., the applications that issue heartbeats to the controller): `x264`, `bodytrack`, `canneal`, and `streamcluster`. The selected applications consist of the most CPU-bound along with the most cache-bound PARSEC benchmarks, providing varied responses to change in resource allocation. Speedups from 3.2*X* (`streamcluster`) to 4.5*X* (`x264`) are observed with the maximum resource allocation values compared to the minimum. We also use one of four machine-learning workloads as our QoS application: `k-means`, `KNN`, `least squares`, and `linear regression`. These four workloads provide a wide range of data-intensive use cases. For all experiments, each QoS application uses four threads. The background (non-QoS) tasks used in the third execution phase are single-threaded microbenchmarks, and have no runtime restrictions, i.e., the Linux scheduler can freely migrate them between and within clusters.

2.4.4 Effectiveness of Self-Adaptivity through Supervision

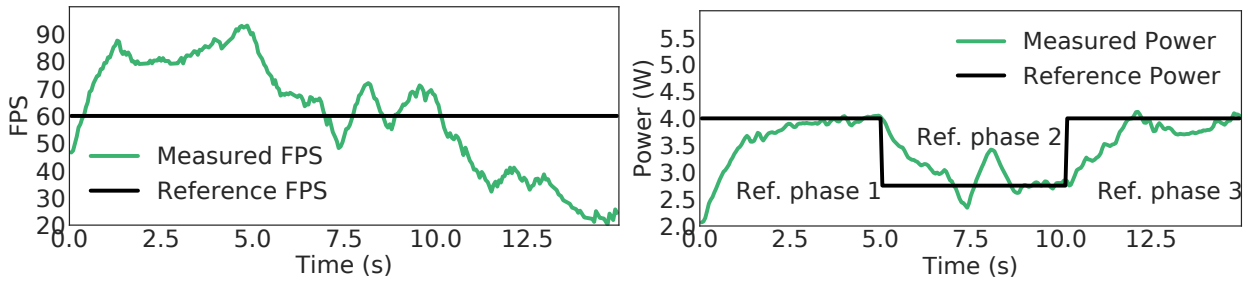
We focus our discussion on the `x264` benchmark results. Other results are summarized at the end of this section. We use heartbeats to measure the frames per second (FPS) as our QoS metric.

Figure 2.6 shows the measured FPS and power for `x264` with respect to their reference values over the course of execution for all of the resource management controllers.

`x264` Benchmark

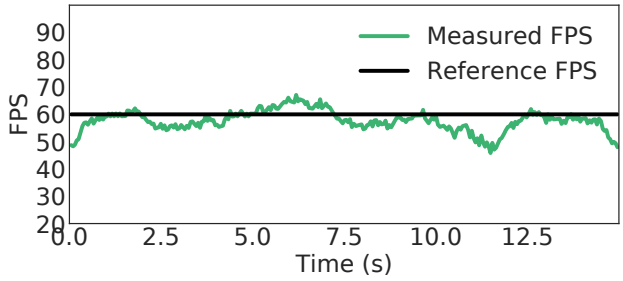
To show the energy efficiency of SPECTR, we study the Safe Phase. The Safe Phase consists of the first 5 s of execution during which only the QoS application executes on the Big cluster. In this phase, all controllers are able to achieve the FPS reference value within the power envelope. Figures 2.7a and 2.7b show the average steady-state error (%) of QoS and power respectively for each resource manager in Phase 1. Steady-state error is used to define *accuracy* in feedback control systems [33]. Steady-state error values are calculated as $reference - measured\ output$. Negative values indicate that the power/QoS **exceeds** the reference value, positive values indicate power savings or failure to meet QoS. We make two key observations. First, both MM-Perf and SPECTR reduce power consumption by 25 % (Fig. 2.7b) while maintaining FPS within 10 % (Fig. 2.7a) of the reference value. The MM-Perf controller operates efficiently because the reference FPS value is achievable within the TDP threshold. The SPECTR controller similarly operates efficiently: it is able to recognize that the FPS is achievable within TDP and, as a result, lower the reference power. Second, the FS and MM-Pow controllers unnecessarily exceed the reference FPS value and, as a result, consume excessive power. This is because these controllers prioritize meeting the power reference value, consuming the entire available power budget to maximize performance.

To show SPECTR's ability to adapt to a sudden change in operating constraints, we study the Emergency Phase. The Emergency Phase of execution emulates a thermal emergency, during which, the TDP is lowered to ensure that the system operates in a safe state. This occurs during the second 5 s period of execution in Figure 2.6. We observe that all controllers are able to react to the change in power reference value and maintain QoS. However, compared to the other controllers, FS has a sluggish reaction (Figure 2.6f) to the change in power reference, despite the fact that it is designed to prioritize tracking the power output. *Settling time* is a property used to quantify responsiveness

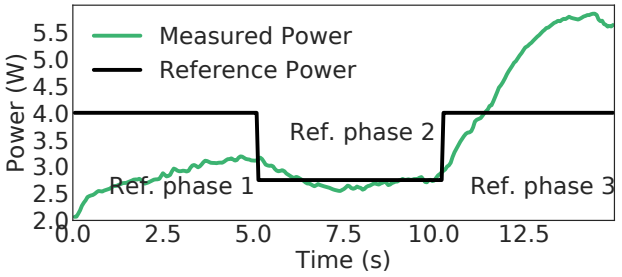


(a) MM-Pow FPS

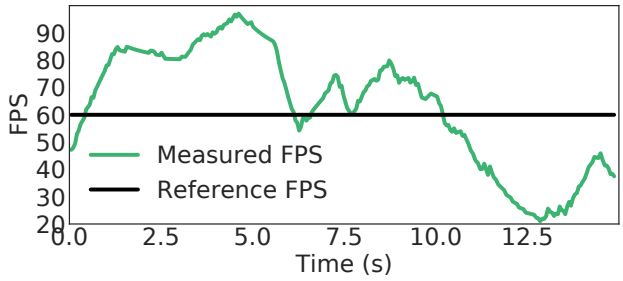
(b) MM-Pow Power



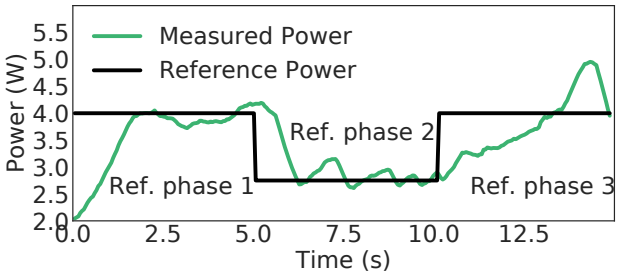
(c) MM-Perf FPS



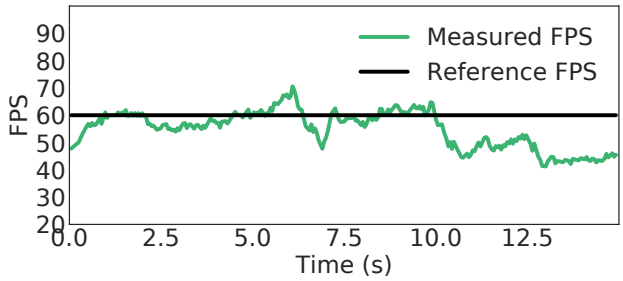
(d) MM-Perf Power



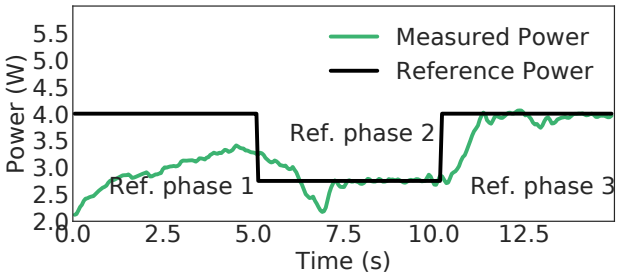
(e) FS FPS



(f) FS Power

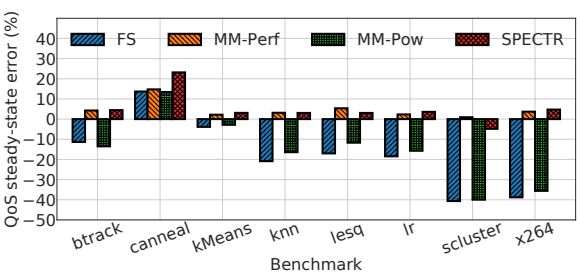


(g) SPECTR FPS

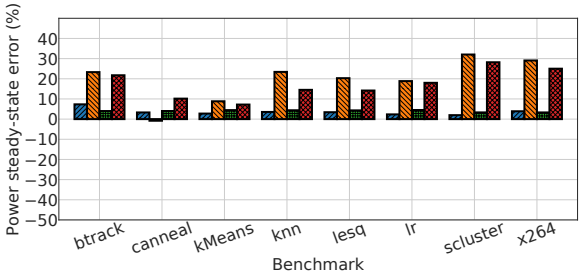


(h) SPECTR Power

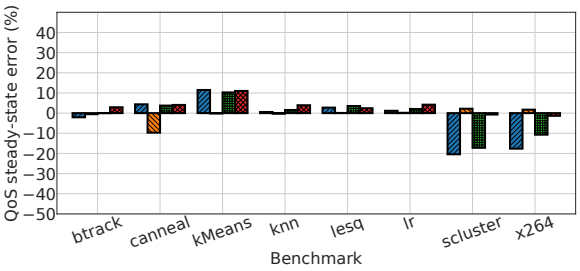
Figure 2.6: Measured FPS and Power of all four resource managers for three Phases of 5 s each, for the x264 benchmark.



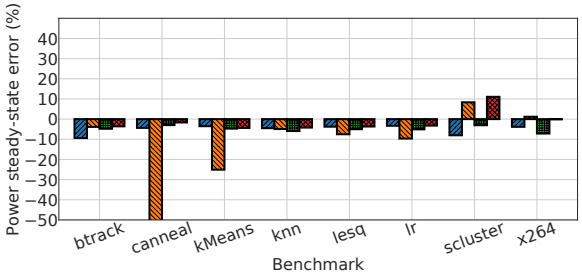
(a) QoS steady-state error in Phase 1.



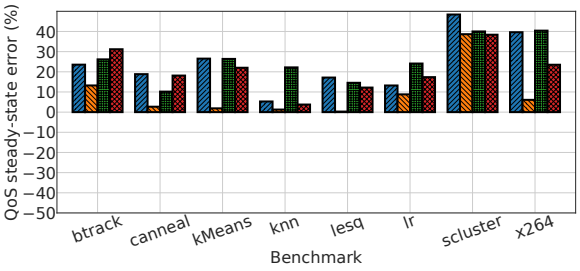
(b) Power steady-state error in Phase 1.



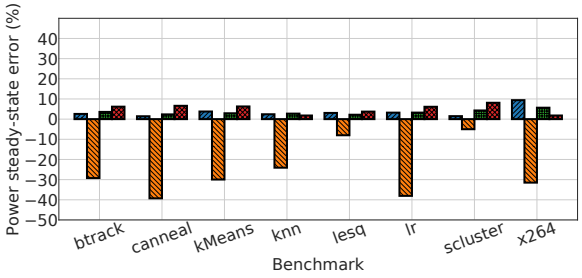
(c) QoS steady-state error in Phase 2.



(d) Power steady-state error in Phase 2.



(e) QoS steady-state error in Phase 3.



(f) Power steady-state error in Phase 3.

Figure 2.7: Steady-state error for all benchmarks, grouped by phase. A negative value indicates the amount of power/QoS **exceeding** the reference value (bad), a positive value indicates the amount of power saved (good) or QoS degradation (bad).

of feedback control systems [33]. *Settling time* is the time it takes to reach sufficiently close to the steady-state value after the reference values are set. The average settling time for the power output of FS is 2.07 s, while SPECTR has an average settling time of 1.28 s. The larger size of the state-space (*x(t)* matrix in Equation 2.1 and 2.2) and the higher number of control inputs in the 4×2 FS compared to those of 2×2 controllers in SPECTR is the reason for the slow settling time of FS. This is also the reason why SISO controllers are generally faster that MIMOs [33].

To show SPECTR's ability to adapt to workload disturbance and changing system goals, we study the Workload Disturbance Phase. The Workload Disturbance Phase occurs in seconds 10-15 of execution in Figure 2.6. In this phase, 1) the QoS reference value and the power envelope return to the same values as in Phase 1, and 2) we introduce disturbance in the form of background tasks. As a result of the workload disturbance, the QoS reference is *not* achievable within the TDP. We make two observations regarding the steady-state error in Figures 2.7e and 2.7f. First, SPECTR behaves similarly to MM-Pow, even though in Phase 1 it behaved similarly to MM-Perf. The SPECTR supervisor is able to recognize the change in execution scenario and constraints, and adapt its priorities appropriately. In this case, SPECTR achieves much higher FPS than all controllers except MM-Perf (Fig. 2.7e), while obeying the TDP limit (Fig. 2.7f). Second, both FS and MM-Pow operate at the TDP limit, but achieve a significantly lower FPS than the reference value. MM-Perf comes within ˜5 % of the reference FPS (Fig. 2.7e) while exceeding the TDP by more than 30 % (Fig. 2.7f), which is undesirable.

Other Benchmarks

We perform the same experiments for PARSEC benchmarks `bodytrack`, `canneal`, `streamcluster`, as well as machine-learning benchmarks `k-means`, `KNN`, `least squares`, and `linear regression`. For these workloads, we use the generic *heartbeat rate* (HB) directly as the QoS metric, as FPS is not an appropriate metric. Figures 2.7a, 2.7c, and 2.7e show the average steady-state error (%) of QoS for Phases 1, 2, and 3 respectively. Figures 2.7b, 2.7d, and 2.7f show the average steady-state

error (%) of power for Phases 1, 2, and 3 respectively. We summarize the observations for the additional experiments with respect to `x264` for the three phases. In the Safe Phase, the behavior of `bodytrack`, `streamcluster`, `k-means`, `KNN`, `least squares`, and `linear regression` is similar to that of `x264` (Figures 2.7a and 2.7b). `canneal` follows the same pattern with respect to power as all other benchmarks (Fig. 2.7b). `canneal`'s QoS steady-state error is the only difference in behavior we observe in Phase 1. None of the managers are able to meet the QoS reference value for `canneal` in Phase 1 (Fig. 2.7a). This is due to the fact that the phase of `canneal` captured in the experiment primarily consists of serialized input processing, so the number of idle cores has reduced affect on QoS. In the Emergency Phase, our observations from `x264` hold for nearly all benchmarks regarding response to change in power reference value, achieving less than 10 % power steady-state error (Fig. 2.7d). The only exceptions are `canneal` and `k-means`: the MM-Perf manager is unable to react to change in TDP for `canneal` and `k-means`. The MM-Perf manager lacks a supervisory coordinator and prioritizes performance, and was unable to find a configuration for `canneal` and `k-means` that satisfied the QoS reference value within TDP. In the Workload Disturbance Phase, SPECTR, FS, and MM-Pow all achieve near-reference power (Fig. 2.7f). As expected, MM-Perf violates the TDP in all cases, but always achieves the highest QoS (Fig. 2.7e).

We conclude that SPECTR is effective at (1) efficiently meeting multiple system objectives when it is possible to do so, (2) appropriately balancing multiple conflicting objectives, and (3) quickly responding to sudden and unpredictable changes in constraints due to workload or system state.

2.4.5 Overhead Evaluation

To show the overhead of the low-level MIMO controllers, we study their execution time. We measure the MIMO controller execution time to be 2.5 ms, on average, over 30 s. The MIMO controller is invoked every 50 ms resulting in a 5 % overhead, which is experienced by all evaluated controllers. We measure the runtime of the supervisor to be 30 µs, which is negligible even with

respect to the MIMO controller execution time. The supervisor is invoked less frequently than the MIMO controllers (2× the period in our case), executes in parallel to the workload and MIMO controllers, and simply evaluates the system state in order to determine if the MIMO controller gains need changing. State changes that result in interventions on the low-level controllers occur only due to system-wide changes in the state (e.g., thermal emergency) or goals (e.g., change in performance reference value or execution mode), which are infrequent. When the supervisor needs to change the MIMO gains, it simply points the coefficient matrices to a different set of stored values. In our case study, we have two sets of gains (QoS and power oriented) that are generated when the controllers are designed and stored during system initialization. Changing the coefficient arrays at runtime takes effect immediately, and has no additional overhead.

To show the overhead of SPECTR's supervisory controller, we compare the total execution time of identical workloads with and without SPECTR. With respect to the preemption overhead due to globally managing resources, Linux's HMP scheduler typically maps SCT threads to a core on the low-power Little cluster. Therefore, the SCT threads are executed without preempting the QoS application, which always executes on the Big cluster. We verify the overall impact of the control system overhead by running the benchmarks on two different systems: i) a vanilla Linux setup[4] and ii) vanilla Linux with SPECTR running in the background. For (ii), SPECTR controllers perform all the required computations but do *not* change the system knobs (thus only the SPECTR overhead affects the system). When comparing the QoS of the applications across multiple runs, we verify a negligible average difference of 0.1% between the two systems.

We conclude that the benefits of SPECTR come at a negligible performance overhead.

[4] Ubuntu 16.04.2 LTS and Linux kernel 3.10.105 (https://dn.odroid.com/5422/ODROID-XU3/Ubuntu/).

2.4.6 SOSA Resource Manager

The SPECTR resource manager specified previously is responsible for configuring knobs with uniform subsystem scope. The SPECTR supervisor includes an optimizer to find the ideal set-point for the low-level classical controllers. However, one benefit of the supervisory method we propose is the flexibility to incorporate any off-the-shelf low-level controllers.

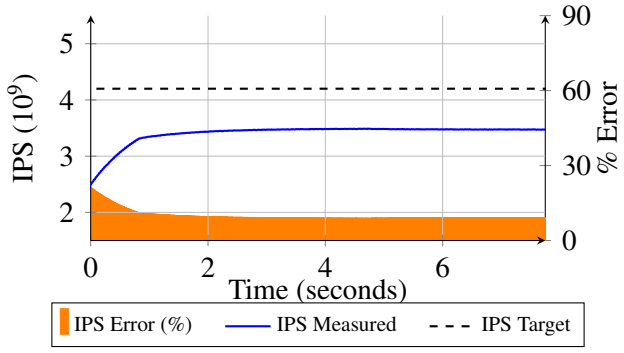
One way we can take advantage of this property to improve the SPECTR hierarchy is to deploy reinforcement learners as low-level controllers. Reinforcement learners are self-optimizing, which removes the need for optimization in the supervisor. Our improved resource manager (SOSA [21]) is now both **s**elf-**o**ptimizing via low-level controllers and **s**elf-**a**daptive via supervisor. We further explore self-optimizing low-level controllers in Chapter 3. Here, we design an improved SOSA resource manager that no longer requires an optimizer, and includes configuration knobs at different scopes: per-core operating frequency, and global task migration. The per-core frequency only affects each subsystem, while task migration between cores affects multiple subsystems. The SOSA supervisor includes a task migration policy to coordinate requests from the distributed low-level controllers.

2.4.7 SOSA Experimental Evaluation

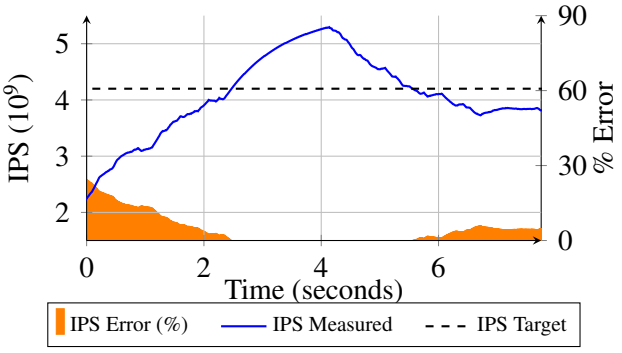
We further validate the self-adaptivity provided by our supervisory approach in the SOSA use-case.

To show SOSA's ability to adapt to changing operating conditions, we study its ability to track a changing goal for a fixed workload. Our execution scenario consists of two different phases of execution:

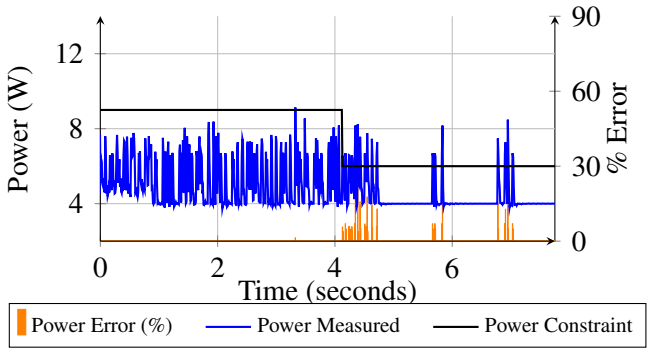
1. *Normal Phase*: In this phase, only the focus application executes. The goal is to meet target instructions-per-second (IPS) and minimize power consumption.



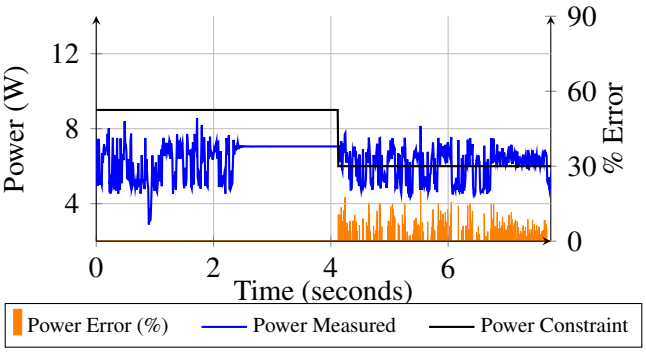
(a) SPECTR instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.



(b) SOSA instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.



(c) SPECTR power. Error measured as % of the max power possible *above* budget.



(d) SOSA power. Error measured as % of the max power possible *above* budget.

Figure 2.8: Instructions-per-second (IPS) and power for `k-means` with a power budget change after 4 s. IPS values are smoothed using averaging.

2. *Low-power Phase*: In this phase, the IPS target remains the same as that in the Normal Phase while the power budget is reduced. The goal is to prioritize honoring the power budget while maintaining target IPS (if possible).

We compare to an implementation of SPECTR that uses per-core SISOs (Frequency→IPS), with a self-adaptive software supervisor to provide target coordination and a simple task migration heuristic. SPECTR consists of a self-adaptive supervisor and classical low-level controllers [72]. Our goal is to demonstrate that SOSA's self-adaptivity is on-par with SPECTR.

Figure 2.8 shows SOSA's and SPECTR's ability to track the IPS target while honoring the power budget in the Normal Phase and Low-power Phase. The first 4 s of execution are in the Normal Phase. Observe the IPS of both managers (Figure 2.8a and 2.8b). SPECTR reaches its peak value after one second of execution, maxing out at 9% below the IPS target. SOSA's learning takes longer – it spends the first 2.5 s exploring the configuration space, and reaches peak value at 4 s. However, SOSA is able to match SPECTR's peak IPS value after only 1.5 s, and reaches a peak value above the IPS target and more than **1.5**× that of SPECTR, all within the power budget. We make two specific observations in SOSA's Normal Phase regarding the power (Figure 2.8d). First, the power spends the first 2.5 s with substantial noise, indicating exploration for learning. Second, the settling of the power around the 2.5 s mark indicates that SOSA has learned a meaningful model. This is reinforced by our observations about the IPS. SPECTR's lower IPS peak and continuous power noise is due to excessive task migration. The task migrations negatively affect the utilization of each core by the focus tasks, limiting the maximum achieved IPS and causing dips in the power (i.e., the low ends of the spikes).

After 4 s, the execution enters the Low-power Phase, and the power budget is reduced by half. SPECTR continues attempting configurations to increase IPS, as it is still not reaching the target. After 0.5 s, the controller responds to the change in priority, and it finds a configuration that honors the new budget without IPS degradation. In Figure 2.8c we see that the SPECTR controller peri-

odically experiences large power spikes that violate the budget. In the Low-power phase, SOSA's configuration violates the power budget, causing the learners to switch to low-power models, which have yet to be populated. This explains the degradation in IPS (Figure 2.8b), as well as the noise in the power (Figure 2.8d). Observe that at 7 s, both the IPS and power begin to stabilize around their targets, with ˜10 % error. Looking closely, just before 8 s, the power for SOSA dips significantly, indicating that the learners are continuously updating and acting in an attempt to honor the power constraint. Based on the learning observed in the Normal Phase, we believe that SOSA will have a populated model that will lead to efficient decisions. We also believe that this presents an opportunity for applying transfer learning, as there are clearly some rules and configurations that are desirable for both Normal and Low-power Phases.

In conclusion, both resource managers are able to consider dynamic goals, balancing IPS targets while accounting for a power budget. The SOSA LCTs take time to learn new objectives compared to pre-populated models, but they are able to learn from undesirable configurations, and as a result SOSA considerably outperforms SPECTR, which struggles to coordinate migration and DVFS.

2.5 Summary

Modern mobile systems require intelligent management to balance user demands and system constraints. At any given time, the relative priority of demands and constraints may change based on uncontrollable context, such as dynamic workload or operating condition. A resource manager must be able to autonomously detect such context changes and adapt appropriately. This property is known as self-adaptivity. We demonstrate one way to design a self-adaptive resource manager: using supervisory control theory. Supervisory control theory lends itself well to this challenge due to its high level of abstraction and lightweight implementation. The proposed supervisor successfully adapts to changes when managing quality of service under a power budget for chip multiprocessors.

The hierarchy using supervisory control theory represents early exploration of self-adaptivity in the resource management domain, and a slight degree of self-awareness. This approach can be enhanced in one way through the definition and generation of goals. Initial work based on goal-driven autonomy has been done toward this end [79].

Chapter 3

Self-Optimizing Controllers

3.1 Self-optimization

Self-optimization is the ability of a system to adapt and act efficiently by itself in the face of *internal stimuli*. We consider internal stimuli as changes related to dynamics in the system's self-model, i.e., model inaccuracy. Internal stimuli do not necessarily include workload itself, but if the self-model is application-dependent, workload changes may be the source of internal stimuli. For example, if the system's self-model is application-dependent, and the executing application changes, a self-optimizing manager will have the ability to reason and act towards achieving the system goal(s) efficiently for the new application. However, if the system's self-model is rigid and the system dynamics used to reason and act are oversimplified, model inaccuracies may lead to undesirable or inefficient decisions when the application changes.

3.2 Motivation

Dynamic voltage/frequency scaling (DVFS) has been established as an effective technique to improve the power-efficiency of chip-multiprocessors (CMPs) [34]. In this context, numerous closed-loop control-theoretic solutions for chip power management [73, 38, 58, 63, 54, 94] have been proposed. These solutions employ *linear control* techniques to limit the power consumption by controlling the CMP operating frequency. However, the relationship between operating frequency and power is often *nonlinear*. Figure 3.1 illustrates this by showing total power consumed by a 4-core ARM A15 cluster executing a CPU-intensive workload through its entire frequency range (200MHz–2GHz), along with the total power consumed by a 4-core ARM A7 cluster through its frequency range (200MHz–1400MHz). While the A7 cluster frequency-power relationship is almost linear, the A15 cluster's larger frequency range (and more voltage levels) results in a nonlinear relationship. Using a linear model to estimate the behavior of such a system leads to inaccuracies. Inaccurate models result in inefficient controllers, which defeats the very purpose of using control theoretic techniques for power management.

Ideally, control-theoretic solutions should provide formal guarantees, be simple enough for runtime implementation, and handle nonlinear system behavior. Static linear feedback controllers can provide robustness and stability guarantees with simple implementations, while adaptive controllers modify the controller at runtime to adapt to the discrepancies between the expected and the actual system behavior. However, modifying the controller at runtime is a costly operation that also invalidates the formal guarantees provided at design time.

Instead, consider integrating multiple linear models within a single controller implementation in order to estimate nonlinear behavior of DVFS for CMPs. This is a well-established and lightweight adaptive control theoretic technique called *gain scheduling*.

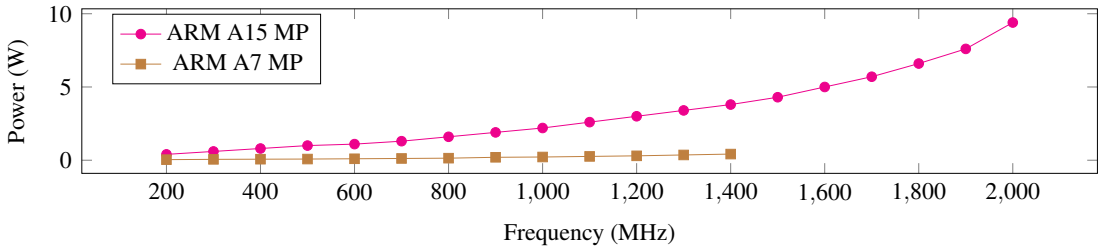


Figure 3.1: Cluster power vs. operating frequency.

Background on Classical Control

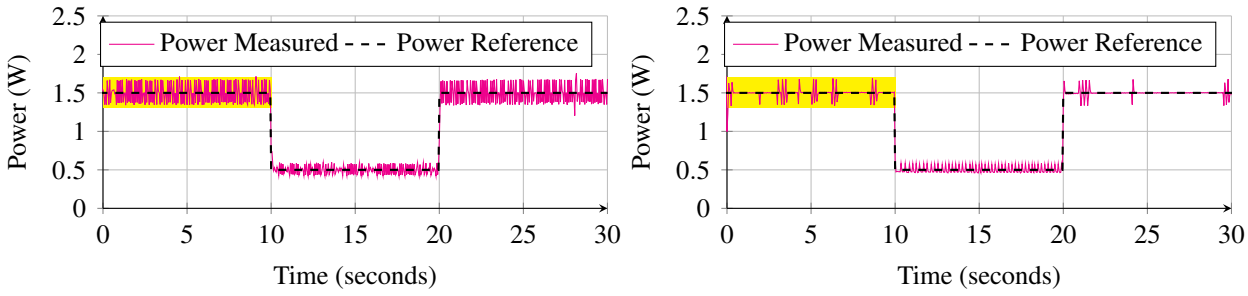
Discrete-time control techniques are the most appropriate to implement control of computer systems. The proportional-integral-derivative (PID) controller is a simple and flexible classical feedback controller that computes control input $u(t)$ based on the error $e(t)$ between the measured output and reference output:

$$u(k) = K_p e(k) + K_i \sum_{0}^{k} e(k)\Delta t + K_d \frac{\Delta e(k)}{\Delta t} \tag{3.1}$$

K_p, K_i, and K_d are control parameters for the proportional, integral, and derivative gains respectively.

PI controllers[1] have been successfully used to manage DVFS of CMPs [58, 101, 63, 54, 94]. Mishra et al. [58] propose the use of PID controllers for VF islands. The authors model power consumption based on the assumption that the difference relationship between power consumption in successive intervals can be approximated linearly as a function of frequency, which only holds for limited range. Similarly, Hoffman et al. [38] propose a feedback control technique for power management that includes DVFS, and their transfer function assumes a linear relationship between power and frequency. However, Figure 3.1 shows that $f \rightarrow P$ becomes nonlinear at higher frequencies. Inaccuracies in linear estimation of nonlinear systems can negatively impact the steady-state error and transient response of the controller. Take for example a system operating under a power

[1] Due to the significant stochastic component of computer systems, PI controllers are preferred over PID controllers [33].



(a) Full range SISO controller (Controller 1).

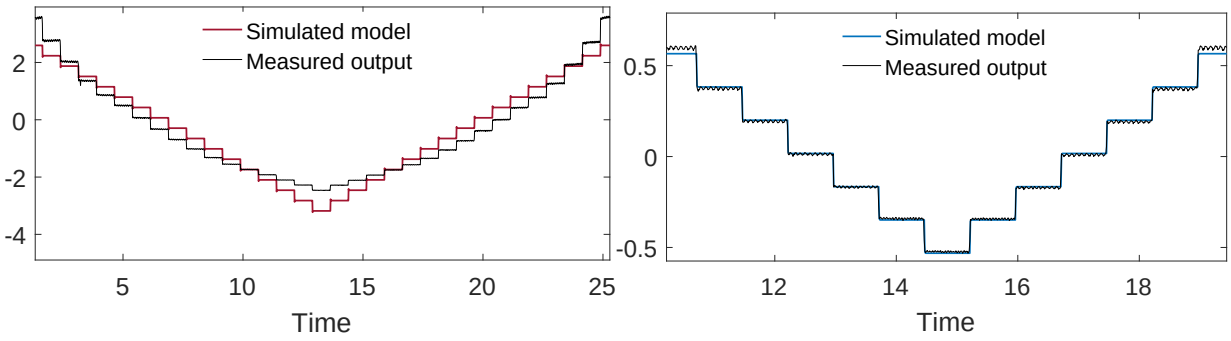
(b) Sub-range SISO controller (Controller 2).

Figure 3.2: Time plots of two DVFS controllers tracking a dynamic power reference.

budget, or experiencing a thermal emergency – a DVFS controller designed from an inaccurate model could lead to wasted power or even unnecessary operation at an unsafe frequency.

Consider a DVFS controller for a 4-core CMP with a single frequency domain. The first steps in designing a controller are defining the system and identifying the model. The power consumption of our CMP is not linear across the entire range of supported operating frequencies (200MHz–2GHz), which makes it challenging to model the entire range with a single linear estimation. However, we can divide the measured output (power) for the entire range of frequencies into multiple *operating regions* that exhibit linear behavior. In this example, we identify a model for two different systems: (1) the CMP's behavior through all operating frequencies; (2) the CMP's behavior through a sub-range of the operating frequencies. This specific operating region spans the frequency sub-range of 200MHz–1200MHz. Using these models, we can generate two different $f \rightarrow P$ Single-Input-Single-Output (SISO) PI controllers, and compare them using measured SASO analysis [33], focusing on *Accuracy* and *Settling time*. We refer to the full-range controller as Controller 1, and the sub-range controller as Controller 2. Figure 3.2 displays Controller 1 (Fig. 3.2a) and Controller 2's (Fig. 3.2b) ability to track a dynamic power reference over time for our CMP.

Accuracy is defined by the steady-state error between the measured output and reference input, e.g., the yellow highlighted region in Figure 3.2b from 0-10 seconds. We calculate the steady-state error as the mean squared error (MSE) between the measured power and reference power. Both



(a) Power for full frequency range.

(b) Power for 200–800 MHz.

Figure 3.3: Modeled and observed behavior of nonlinear full-range system (a) vs. linear operating region (b).

controllers are able to track within 1% of the target power. However, the MSE of Controller 2 is 0.003, while that of Controller 1 is 0.013 – an order of magnitude larger. This byproduct of model inaccuracy translates into wasted power and undesirable operating frequency, as well as unnecessary changes in the frequency control input (i.e., increased control effort cost).

Settling time is the time it takes to reach sufficiently close to the steady-state value after the reference values are specified, e.g., when the reference changes in Figure 3.2b at 10 seconds. The settling time of Controller 2 is 40 ms on average, while Controller 1 is more than double on average at 100 ms. Because our actuation periods are 50 ms, this means that our sub-range controller often reaches steady state on its first actuation while the full range controller requires multiple actuation periods to respond to a change in reference.

Identifying operating regions at design time allows us to switch system models at runtime, improving the effectiveness of static controllers.

3.3 Case Study: Gain Scheduled Controller (GSC) for Power Management

In this section, we outline our process for designing gain scheduled nonlinear controllers for a CMP.[2] As a demonstrative case study, we target the ODROID-XU3 platform [32] which contains an ARM big.LITTLE based Exynos 5422 Octa-core SoC that has heterogeneous multi-processing (HMP) cores. The Exynos platform contains an HMP with two 4-core clusters: the *big* cluster provides high-performance out-of-order cores, while the *little* cluster provides low-power in-order cores. For the purpose of our study, we disable the little cluster (due to its linear behavior) and use only the big cores to emulate a uniform nonlinear CMP[3].

3.3.1 Defining and Modeling Linear Subsystems

Selecting the control input and measured output of a DVFS controller is straightforward. Frequency is the knob available to the user in software, and power is the metric of interest. On our Exynos CMP, the operating frequency of cores is set at the cluster level, and power sensors measure power at the cluster level. A SISO controller is a natural solution, with the entire CMP composing the system under control.

For system identification we generate test waveforms from applications and use statistical black-box methods based on System Identification Theory [51, 52] for isolating the deterministic and stochastic components of the system to build the model.

Figure 3.3a shows a comparison of a simulated model output vs. the measured output over the entire frequency range of our CMP. It is evident that there are ranges for which the estimated behavior differs from that of the actual system behavior. We know that voltage has a nonlinear

[2] Details to confirm and formalize popular notions regarding gain scheduled design can be found in [78, 48].
[3] We refer to this as the Exynos CMP or CMP throughout.

Region	Frequency Range (MHz)	Voltage (V)
1	1600 – 2000	1.25
2	1300 – 1500	1.10
3	900 – 1200	1.00
4	200 – 800	0.90

Table 3.1: VF Pairs for ARM A15 in Exynos 5422.

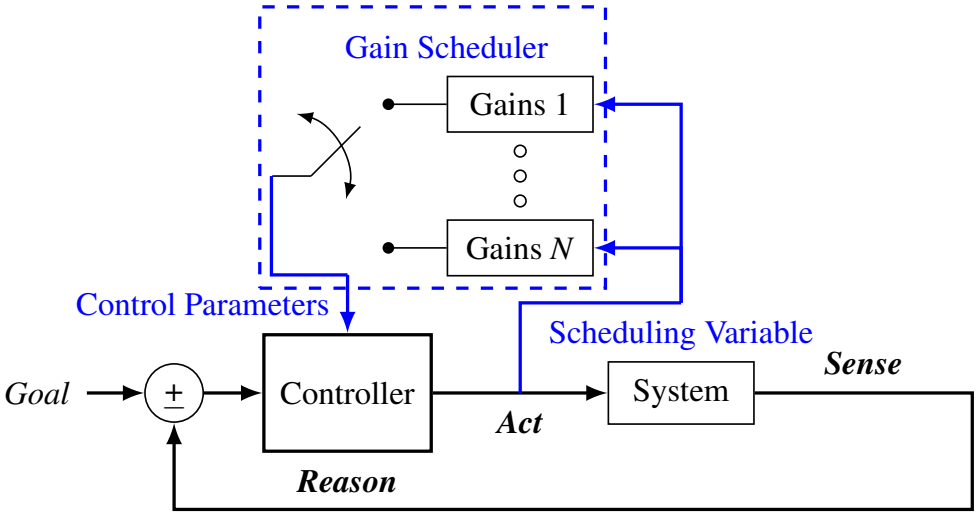


Figure 3.4: Block diagram of GSC.

effect on dynamic power ($P = CV^2 f$). The nonlinear relationship between frequency and voltage pairs through the range of operating frequencies amplifies this effect (Table 3.1). Table 3.1 lists all valid VF pairs for the CMP, in which there are only four different voltage levels. Figure 3.3b shows the measured vs. modeled output when the system is defined by a single operating region grouped by frequencies that operate at the same voltage level.

3.3.2 Generating Linear Controllers

We generate a PI controller separately for each operating region using the system models and MATLAB's Control System toolbox. This is a straightforward process for a simple off-the-shelf PI controller.

In the next step, the designed controller is evaluated against disturbance and uncertainties in order to ensure it remains stable at a defined confidence level. Unaccounted elements, modeling limitations, and environmental effects are estimated as model uncertainty in order to check the disturbance rejection of the controller. In our case, we can confirm our controller is robust enough to reject the disturbance from workload variation.

Each controller we design for an operating region is defined by its control parameters K_P and K_I which are stored (in memory) in the gain scheduler (Figure 3.4). In the gain scheduler, we incorporate logic to determine which gains to provide the controller when invoked.

3.3.3 Implementing Gain Scheduling

The gain scheduler enables us to adapt to nonlinear behavior (Figure 3.4) by combining multiple linear controllers. It stores predefined controller gains and is responsible for providing the most appropriate gains based on the operating region in which the system currently resides each time the controller is invoked.

The scheduling variable is the variable used to define operating regions. For our controller, the scheduling variable is frequency as it is simpler to implement in software and has a direct VF mapping (Table 3.1). Our gain scheduler implements lightweight logic that determines the set of gains based on the system's operating frequency (scheduling variable). Algorithm 1 shows the logic implemented in our gain scheduler with N operating regions where f is the scheduling variable and K_P and K_I are the controller parameters. In addition to the K_P and K_I controller parameters, there is also an $offset$. The $offset$ is the mean actuation value for the operating region, and is necessary for providing the control input for the next control period. Algorithm 1 accounts for the transitions between operating regions (lines 1-6) by applying a full-range linear controller. This method is utilized as the sets of gains for a particular operating region perform poorly outside of that region.

Algorithm 1 Gain Scheduler Implementation

Input: f: frequency, scheduling variable
Outputs: K_{P_n}, K_{I_n}, $offset_n$: updated controller parameters;
Variables: ref_{prev}, ref_{next}: power reference values for previous and next control periods;
Constants: $Region[N]$: operating regions, defined by mutually exclusive range of frequencies; $K_P[N]$, $K_I[N]$, $offset[N]$: stored controller parameters for each operating region; K_{P_G}, K_{I_G}, $offset_G$: controller parameters for full-range linear controller;

```
1:  if ref_next != ref_prev then
2:      K_Pn = K_PG
3:      K_In = K_IG
4:      offset_n = offset_G
5:      return
6:  else
7:      for i = 1 to N do
8:          if Region[i].contains(f) then
9:              K_Pn = K_P[i]
10:             K_In = K_I[i]
11:             offset_n = offset[i]
12:             return
13:         end if
14:     end for
15: end if
```

3.3.4 Experiments

Our goal is to evaluate our nonlinear GSC with respect to the state-of-the-art linear controller in terms of both theoretical and observed ability to track power goals on a CMP. Our evaluation is done using the Exynos CMP running Ubuntu Linux.[4] We consider a typical mobile scenario in which one or more multi-threaded applications execute concurrently across the CMP.

Controllers

We designed two DVFS controllers for power management of the CMP: 1) **a linear controller** that estimates the transfer function similarly to [38, 58]; and our proposed 2) **GSC**. The GSC contains three operating regions (Table 3.2). We combine the two smallest adjacent Regions, 1 and 2 (Table 3.1), to create Controller 2.1. Controllers are provided a single power reference for the

[4] Ubuntu 16.04.2 LTS and Linux kernel 3.10.105

whole system. The control input is frequency, and the measured output is power, applied to the entire CMP.

The controller is implemented as a Linux userspace process that executes in parallel with the applications. Power is calculated using the on-board current and voltage sensors present on the ODROID board. Power measurements and controller invocation are performed periodically every 50 ms.

Workloads

We developed a custom micro-benchmark used for system identification. The micro-benchmark consists of a sequence of independent multiply-accumulate operations yielding varied instruction-level parallelism. This allows us to model a wide range of behavior in system outputs given changes in the controllable inputs. We test our controllers using three PARSEC benchmarks: `bodytrack`, `streamcluster`, and `x264`. For each case, we execute one multithreaded application instance of the benchmark with four threads, resulting in a fully-loaded CMP. We empirically select three references that we alternate between during execution. ref_1 is 3.5 W, the highest reference and a reasonable power envelope for a mobile SoC. This represents a high-performance mode that maximizes performance under a power budget. ref_2 is 0.5 W, the lowest reference and represents a reduced budget in response to a thermal event. ref_3 is 1.5 W, a middling reference that could represent the result of an optimizer that maximizes energy efficiency. These references are not necessarily trackable for all workloads, but should span at least three different operating regions for each workload. For each case, the applications run for a total of 65 s. After the first 5s (warm-up period) the controllers are set to ref_1 for 20 s, then changed to ref_2 for 20 s, and to ref_3 for the remaining 20 s.

	Ctrl 1	Ctrl 2.1	Ctrl 2.2	Ctrl 2.3
Freq. Range	200 – 1800	1300 – 1800	900 – 1200	200 – 800
Stable	✓	✓	✓	✓
Accuracy (MSE)	0.1748	0.03089	0.0005382	0.0003701

Table 3.2: Accuracy of the full- (Ctrl 1) and sub-range (Ctrl 2.x) controllers.

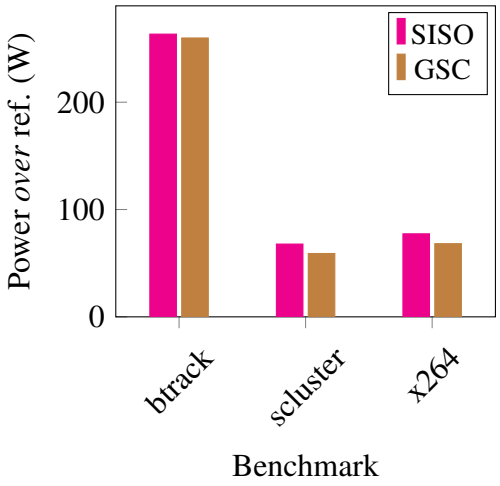
3.3.5 Controller Design Evaluation

We used a first-order system, with a target crossover frequency of 0.32. This resulted in a simple controller providing the fastest settling time with no overshoot. Models are generated with a stability focus and uncertainty guardbands of 30 %.

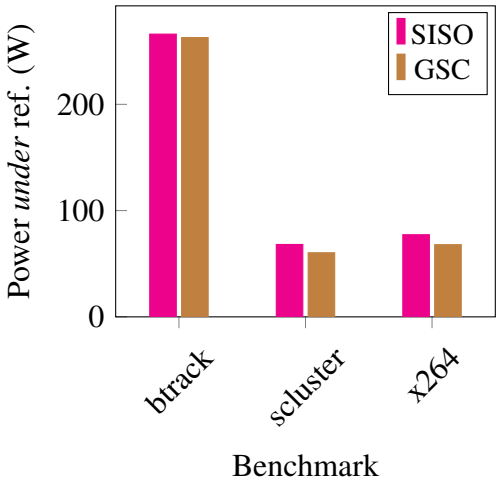
All systems are stable according to Robust Stability Analysis. By design all overshoot values are 0. The settling times of Controllers 2.2 and 2.3 are comparably low at 5 control periods. Controller 2.1 (the most nonlinear operating region) and Controller 1 are slightly higher at 8-9 control periods. The ideal controllers are all very similar in terms of stability, settling time, and overshoot. The primary difference between them is in terms of accuracy. Controllers 2.1-2.3 achieve an order of magnitude better accuracy than Controller 1 (Table 3.2). This means that the region controllers are equally as responsive as the full-range model in achieving a target value while achieving the value more accurately.

3.3.6 Controller Implementation Evaluation

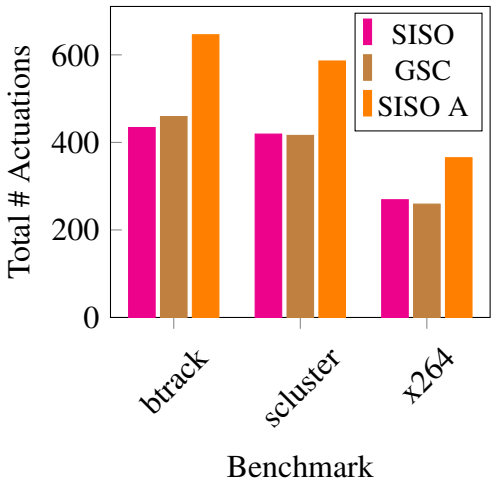
We now evaluate the effectiveness of our nonlinear control approach implemented in software on the Exynos CMP for multithreaded mobile workloads. Traditional SASO control analysis gives us a way to compare the controllers in theory, but the system-level effects of those metrics are not directly relatable. Therefore, we will compare the runtime behavior of the software controllers using a slightly modified set of metrics: power over target, power under target, number of actuations, and response time. These metrics are shown in Figure 3.5.



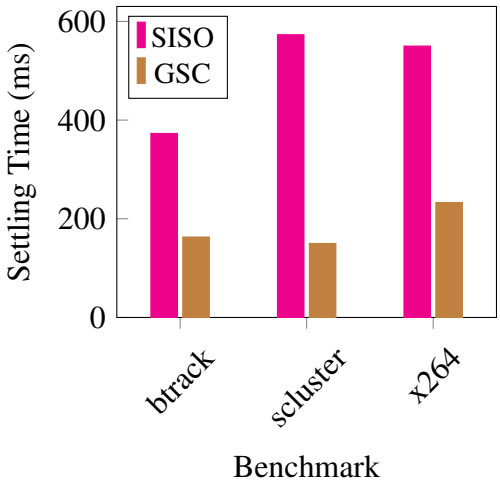
(a) Total power over ref.



(b) Total power under ref.



(c) Total actuations



(d) Average response time

Figure 3.5: Comparison of GSC with Controller 1.

The *power over target* is the total amount of measured power exceeding the reference power throughout execution (Fig. 3.5a). This is the area under the output and above the reference. It represents the amount of power wasted due to inaccuracy, and can also represent unsafe execution above a power cap. Our GSC is able to achieve 12 % less power over target than the linear controller for `x264` and `streamcluster`. `bodytrack` is the most dynamic workload and results in the noisiest power output. In this case the GSC only improves the power over target by 1 % compared to the linear controller.

The *power under target* is the total amount of measured power falling short of the reference power throughout execution (Fig. 3.5b). This is the area under the reference and above the output. A lower value translates to improved performance (i.e. lower is better). Similarly to the power over target, our GSC is able to reduce power under target by **12%** for `x264` and `streamcluster`, and 1 % for `bodytrack`.

The *number of actuations* is simply a count of how many times the frequency changes throughout execution, and is a measure of overhead (Fig. 3.5c). The GSC's actuation overhead is lower than the linear controller for `bodytrack`, `streamcluster`, and `x264` by 8 %, 1 %, and 4 % respectively. This is expected, as the controller's resistance to actuation is related to the crossover frequency specified at design time. For the same crossover frequency, the GSC benefits are primarily in the accuracy (power over/under target) and response (settling) time. To illustrate this tradeoff, we performed the same experiments for a full-range linear controller with a target crossover frequency of 0.8 (Controller 1b). We arrived at this value empirically: Controller 1b achieves comparable accuracy to the GSC. However, GSC reduces the actuation overhead by 29 % for all workloads compared to Controller 1b.

The *response time* is the average settling time when the target power changes, indicating the controller's ability to respond quickly to changes (Fig. 3.5d). Figure 3.5d shows the average response time for each workload for both controllers. The GSC is able to improve the response time over

Controller 1 by more than **50%** in each case. The GSC's overall average response time is 182ms, which is less than 4 control periods.

The implementation overhead of the GSC w.r.t. the linear controller is negligible: it requires a single execution of Algorithm 1 upon each invocation, and storage for a K_P, K_I, and $offset$ value for each operating region. Although workload disturbance plays a significant role in determining the magnitude in imporovement of a nonlinear GSC over a state-of-the-art linear controller, a clear trend exists, and these advantages would increase with the modeled system's degree of nonlinearity.

3.4 Challenges of Model-dependence

The GSC controller specified previously provides optimization by adaptively swapping out three static SISO controllers at runtime. The GSC controller yields benefits for a simple case: achieving a target power for a single-core processor by controlling the voltage-frequency pair. However, contemporary resource managers are required to manage complex systems with multiple knobs, and potentially multiple cores, the dynamics of which may not be fixed, and are not practical to extract ahead of execution with fixed models.

Consider the DVFS feedback controller shown in Figure 3.6. The controller sets the operating frequency (and voltage) of a single-core system to achieve a desired heartbeat rate. The heartbeat rate is a quality-of-service (QoS) metric the application designer specifies through source code annotation [37].

In the case of control theory, the controller is designed based on a static model that identifies the achievable heartbeat based on the operating frequency. This assumes that a physical system is available for observation of system dynamics, which is required to generate the model used to design the controller ahead of deployment. Furthermore, the frequency→HB relationship is application-specific. In other words, (a) workloads must be known ahead of *design* time, (b) sys-

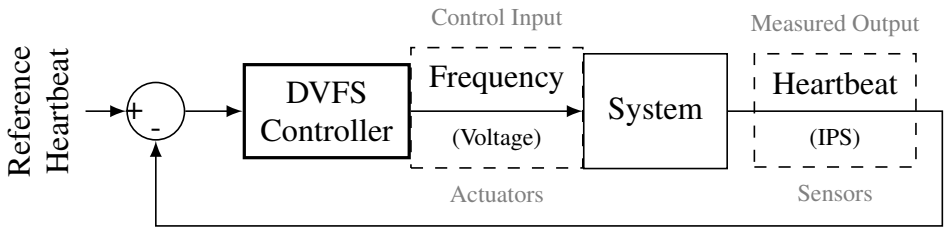


Figure 3.6: Feedback controller with frequency as control input and heartbeat [37] rate as measured output.

tems must be available for observation of known workloads, and (c) each resulting controller only applies to the specific workload it was designed for. These are impractical assumptions when designing controllers for general-purpose systems with dynamic and unpredictable workloads. More challenges arise due to changes in system dynamics over time or between devices. Consider the effects of process variability on the behavior of different devices with respect to operating frequency and voltage. It is impractical to expect an optimized model to be derived at design time for each device that utilizes the controller. As a result, a system may display non-ideal behavior according to the model used to design the controller. The controller would make potentially poor decisions due to an inaccurate ideal model [20].

The challenges outlined so far assume that the systems dynamics being modeled can be estimated with a simple linear equation, which is the case for the frequency→HB controller. However, some knobs have more complex system dynamics and are not practical to model with control, e.g., task migration. Complex models with large configuration spaces such as task migration are good candidates to apply learning. It is important to manage the scale of a complex model if it is to be learned at runtime. A model-based learner that uses a *static* model would face the same challenges as described thus far. To solve these issues we can employ online reinforcement learning. Online reinforcement learning addresses the static-model challenges by continuously updating the system model based on runtime observations. If we take care to implement a lightweight solution, we can perform the learning on-device, and capture complex dynamics such as task migration.

3.4.1 Benefits of Reinforcement Learning

Consider again the DVFS feedback controller shown in Figure 3.6. We implement the feedback controller in two different ways: (1) using single-input-single-output (SISO) control theory, and (2) using a learning classifier table (LCT) – an implementation of rule-based reinforcement learning. Figure 3.7 shows the accuracy achieved by SISO (blue) and LCT (orange) controllers tracking a specified HB for the `k-means` clustering algorithm executing on a simulated ARM core (detailed in Section 3.6.1). The SISO begins with an error of 20 %, and is able to eventually reduce the error to less than 5 % after 2 s of execution. The LCT begins with near 100 % error, and is able to to reduce the error to SI15% after 2 s of execution, and eventually down to 7 % after 5 s. Although the SISO is robust, its design requires a model at design time. The LCT is a blank-slate that learns the model during execution, which is why it begins with nearly 100% error. In this instance we did not tweak LCT parameters or optimize rules – with some design iterations, we could reduce the error further. The LCT's ability to learn to manage HB on the fly indicates that there is opportunity to exploit this approach to coordinate knobs for subsystems in the context of a resource management hierarchy.

In the example, we define both the LCT and SISO objective in terms of IPS, and use a HB→IPS converter to set HB references. Although the converter is a requirement for the classical controller due to its inability to adapt to each application's unique frequency→HB model, it is not a restriction on the LCT. Our design decision is made for fairness in the comparison, but the online learning done by the LCT allows it to define its objective in terms of HB directly, with the ability to adapt to different applications. A controller using a fixed model (e.g., classical controller) simply *cannot* model the relationship between application-level metrics to hardware-level knobs at runtime for a dynamic workload. The ability to specify an objective for a low-level controller in terms of an application-specific user-defined metric is a significant advantage when providing self-optimization.

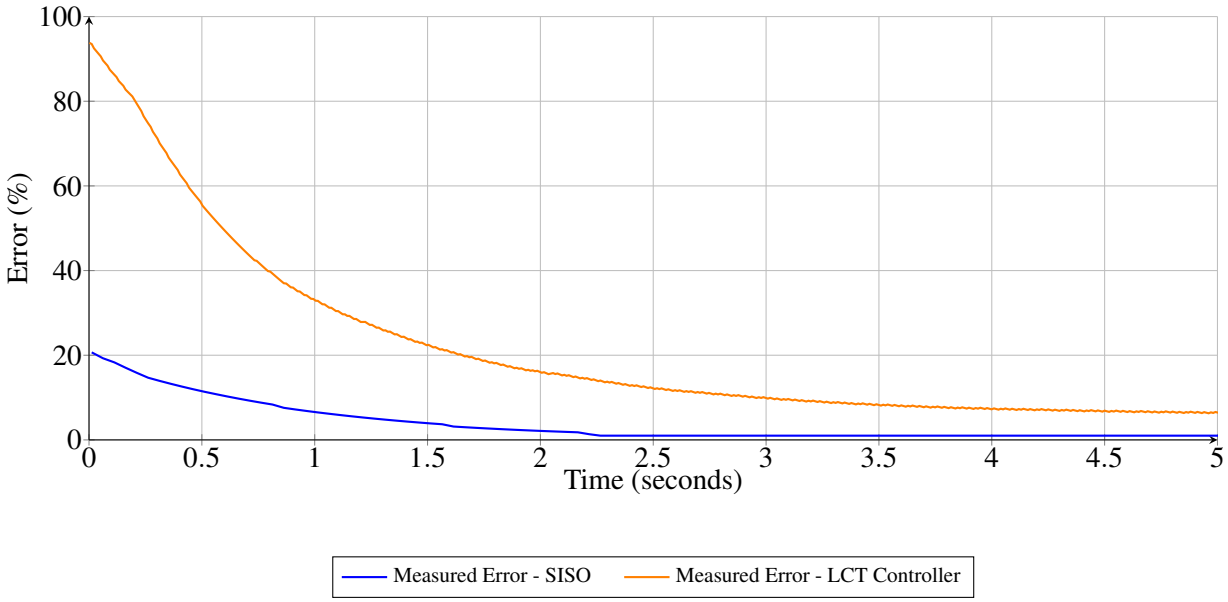


Figure 3.7: Accuracy of classical (SISO) and learning (LCT) controllers tracking application heartbeat rate using core operating frequency.

3.5 Self-Optimization through Learning Classifier Tables

In general, a controller's task is to find the optimal actuation knob configuration for a given system state. The optimum is defined by some (measurable) metric, the objective function. The objective function (δ) can take one or more objectives ($\delta_1, \delta_2, \ldots, \delta_n$) into account

$$\delta\left(\delta_1, \delta_2, \ldots, \delta_n, \ldots\right) \tag{3.2}$$

to compare the desirability of different system states. In our hierarchical setup, the low level controllers' task is to execute actions that result in a more desirable *sub*system state over time according to the given objective function.

Zeppenfeld and Herkersdorf use learning classifier tables (LCTs) as low-level controllers in ASoC [104], which is an approach exploiting autonomic principles for runtime task distribution and frequency scaling. In ASoC, low-level controllers are unsupervised. The overall system goal emerges from the controllers' objective function definition and coordination.

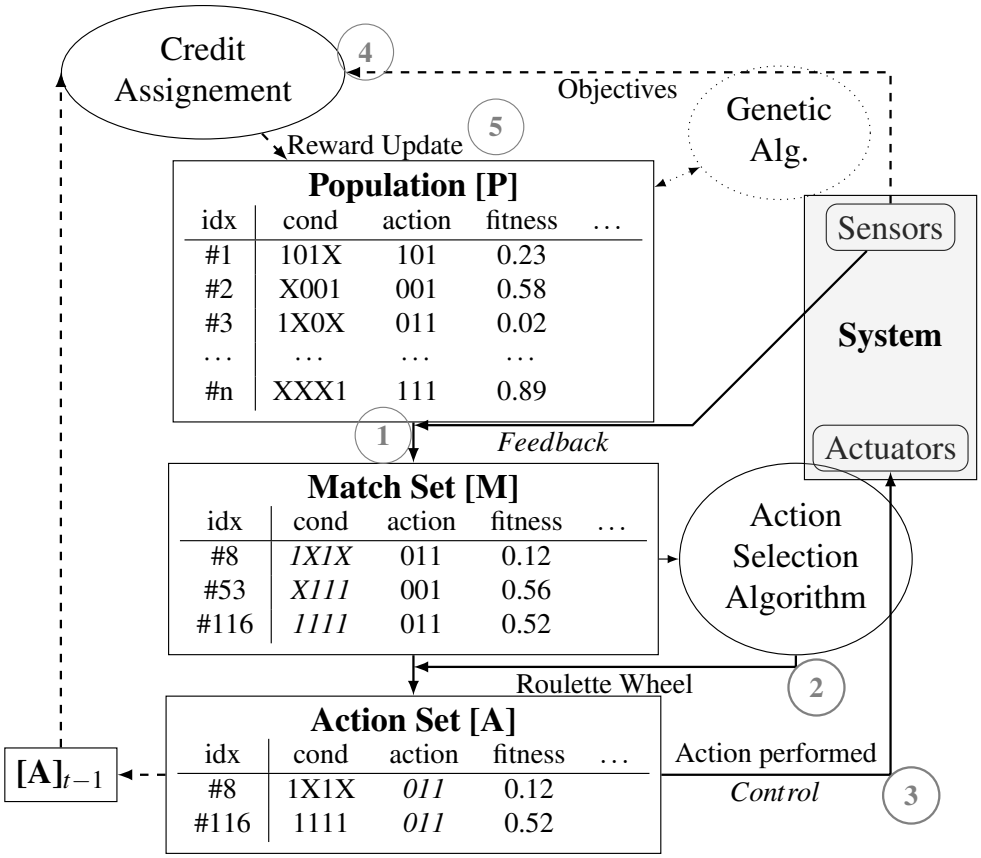


Figure 3.8: Overview of the LCT logic (dashed lines correspond to the fitness update path; dotted entities are part of LCSs, but not LCTs).

LCTs are a subset of Wilson's XCS [98], which is a type of learning classifier system (LCS). LCSs describe the system they control by 'if `condition` then `action`' rules, i.e., classifiers. The `condition` corresponds to a specific set(s) of possible sensor values (i.e., system state). The `action` modifies the actual knob configuration to change system settings. Each rule contains a fitness value which describes its capability to improve the system state according to the objective function. A rule's fitness is updated each time the rule's condition matches the system state and its action is applied. The fitness is updated based on the rule's effectiveness toward achieving the objective. This is how the LCT builds and updates the system model, without any initial training required.

A set of rules, population [P], is needed to model system dynamics. Different rules may have overlapping conditions or actions. Figure 3.8 represents the general operating mode of LCS implemented as an LCT. This operating mode consists of five periodically occurring steps:

1. By comparing the sensor values to the conditions of all rules in the population [P], a match set [M] is generated. Conditions can contain wildcards to match several sensor values.

2. Next, the roulette-wheel selection algorithm decides on an action based on the fitnesses of the rules in [M]. All rules in [M] with the same action selected by the roulette wheel are saved as the action set [A].

3. The selected action is forwarded to the actuators to apply it to the (sub)system. Further, the action set is saved $[A]_{t-1}$.

4. After applying the action, the effect of the action is measured by sensors, and based on the effectiveness toward achieving the objective, some reward is given by the credit assignment component. The fitnesses of the rules in the saved action set $[A]_{t-1}$ get updated based on the reward according to a modified Q-learning which was proposed by Wilson [97] (see Equation 3.6).

5. Finally, the updated rules are forwarded to the population [P] for the next evaluation.

To calculate the reward a rule receives, the credit assignment component observes changes of the objective function (δ). The objective function is calculated as normalized error. The error is the difference between the measured sensor value and its target value. For example, we define our objective function in terms of a performance metric ($PERF$):

$$\begin{aligned} \delta &= \frac{|PERF - ref_{PERF}|}{max_{PERF}} \\ \delta &\in \{x \,|\, 0 \leq x \leq 1\} \end{aligned} \tag{3.3}$$

This equation describes a performance optimization objective with a target performance target (ref_{PERF}). We can further constrain this function, e.g., to maximize $PERF$ within a power budget ($constr_{Power}$), by setting ref_{PERF} to a large value and subjecting the equation to

$$Power \leq constr_{Power} \tag{3.4}$$

The observed change in the objective function results in different rewards (*reward*) according to the following reward function:

$$reward = 1 - \delta \qquad reward \in \{x \,|\, 0 \leq x \leq 1\} \tag{3.5}$$

In the case that the constraint is violated, the reward is set to 0. This reward function with a discrete range supports the ability to distinguish between two different actions for the same condition which both improve or degrade the system state to varying degrees.

Based on the reward, the fitness (fit) of $[A]_{t-1}$ is updated by reinforcement learning using a modified Q-learning algorithm

$$fit \leftarrow fit + \beta \left[\left(reward + \gamma \cdot max \left(fit_{[A]_{t-1}} \right) \right) - fit \right] \tag{3.6}$$

with the learning rate parameter β and the discount factor γ. The discounting $\left(\gamma \cdot max \left(fit_{[A]_{t-1}} \right) \right)$ is omitted in single-step problems like DVFS [98], and therefore also in SOSA.

Our reward and fitness function prevents fitness values of constantly improving/degrading rules from ending up at the minimum or maximum value on the long term. The selected fitness update procedure does not necessary result in stable fitnesses over time for all kind of rules (e.g., general rules which include a lot of don't-cares for a single or multiple sensor inputs). Accuracy-based genetic algorithms are able to recognize unstable rules [98]. Genetic algorithms have also been used in LCS's to discover new rules. Identifying and removing unstable rules, and generating and

testing new rules are not currently parts of the LCT, and therefore out of the scope of this work. We are addressing this in ongoing work in order to enable LCTs to add potentially beneficial rules to the ruleset and remove unstable rules during execution.

3.6 Case Study: Replacing Classical Controllers with LCTs

Figure 3.9 shows an overview of our evaluation platform. We target a 4-core homogeneous CMP consisting of high-performance ARM cores. We consider a typical embedded scenario in which a performance-sensitive application (focus application) is running concurrently with various other (background) applications starting and stopping unpredictably. This mimics a typical use-case in which the main purpose of the device is performed in the foreground in conjunction with background debugging, reporting and logging.

We deploy the SOSA resource management hierarchy. **The system goals** are twofold: i) meet the performance requirement of the foreground application while minimizing its energy consumption; and ii) ensure the total system power always remains below the Thermal Design Power (TDP). In other words, the performance is a *target*, while the power is an *upper bound*. There is no advantage to exceeding the performance requirement.

We consider two actuation decisions: one to set the operating frequency and associated voltage of each core; and one to migrate tasks between cores. We measure the power consumption of each core, and simultaneously monitor the performance (in IPS) of the designated application to compare it to the required performance (IPS_{ref}).

Supervisory control attempts to meet the performance requirement while honoring the power budget. The supervisor prioritizes IPS and power in each **LCT** appropriately based on total system-wide power measurements. Supervisory control commands guide the LCTs to determine the operating frequency of each core and migrate tasks to the core. The LCTs set local core operating

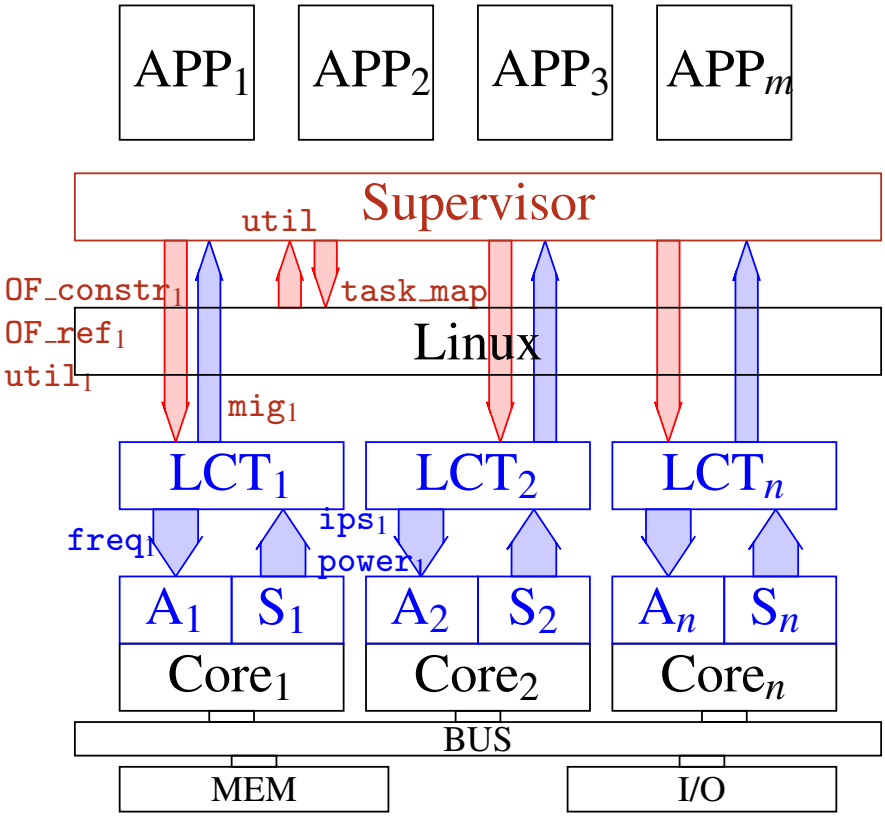


Figure 3.9: Resource management hierarchy (SOSA) implementation on an MPSoC. Each core has an associated LCT, with local sensors (IPS, power, ...) and actuators (core frequency). The software supervisor communicates directly with LCTs to: (a) send global sensor data (per-core utilization); (b) update rules, or objective functions (targets, constraint); (c) receive hardware sensor data (migration request). In this example, the supervisor also communicates with Linux to receive software sensor data (utilization) and send software actuation commands (task migration).

frequency directly via hardware actuator. The task migration actuator is implemented in software as part of the supervisor. The supervisor coordinates the migration flags sent by each LCT to perform global task migration.

An **objective function (δ)** is used to set the optimization objective of the LCTs. The objective functions are defined in terms of IPS or power, each with a constraint (see Equations 3.3 and 3.4). We define two objective functions for this case-study: 1) *IPS-oriented* function ensures that the focus application can meet the performance target value, and 2) *Power-oriented* function limits the power consumption while possibly sacrificing some performance if the system is exceeding the power budget threshold.

We use the **reward function** to enforce the constraints ($constr_{IPS}$, $constr_{Power}$). For the IPS-oriented objective function (δ_{IPS}), we set the reward to 0 when $IPS < constr_{IPS}$; for the Power-oriented objective function (δ_{Power}), we set the reward to 0 when $Power > constr_{Power}$. This has the effect of "forbidding" the violation of the desired system state in each respective objective function. Subsequently, we embed our optimization within the learning mechanism: for δ_{IPS}, we set $ref_{Power} = 0$, so that while the objective function achieves ref_{IPS}, it also minimizes $Power$. The result is the following objective functions:

$$\delta_{IPS} = \frac{Power}{max_{Power}}, \text{ subject to } IPS \geq constr_{IPS} \tag{3.7}$$

$$\delta_{Power} = \frac{|IPS - ref_{IPS}|}{max_{IPS}}, \text{ subject to } Power \leq constr_{Power} \tag{3.8}$$

3.6.1 Learning Classifier Table Evaluation

We demonstrate SOSA's ability to self-adapt and self-optimize by building a model at runtime. We compare to control-theoretic-based resource managers.

Simulation Setup

We perform our evaluations for the platform described in our case study (Figure 3.9) using the gem5 architectural simulator in full-system mode. We implement all resource management software using the MARS middleware framework[5] for Linux, and LCTs are implemented as gem5 simulated hardware modules, mimicking the hardware design as closely as possible. The MARS Linux userspace daemon process invokes the supervisor every 100 ms. The supervisor communicates directly with LCTs using memory-mapped I/O to: (a) send software sensor data; (b) update rules or objective functions; (c) receive hardware sensor data; (d) receive software actuation commands (task migration). LCTs are invoked every 10 ms. We use a combination of ARM's Performance Monitor Unit (PMU) and simulator hardware sensors for the performance and power measurements required by the resource managers. The IPS performance target is provided by the focus application. The learning is done in the gem5 simulated hardware modules, whereas their actions are carried out by the OS (Linux) for the task migration and by the gem5 simulated hardware for the frequency changes. This accounts for the timing and performance overhead of actuations. The frequency of task migration is inherently limited: we allow a maximum of one task migration each time the migration policy is invoked.

The resource managers for comparison use control-theoretic low-level controllers designed with the Matlab System Identification Toolbox [57].[6] We generate training data by executing a microbenchmark (from MARS) and varying control inputs in the format of a staircase test (i.e., a

[5] https://github.com/duttresearchgroup/MARS

[6] We generate the models with a stability focus. All systems are stable according to Robust Stability Analysis. We use Uncertainty Guardbands of 50 % for IPS and 30 % for power, as in [68].

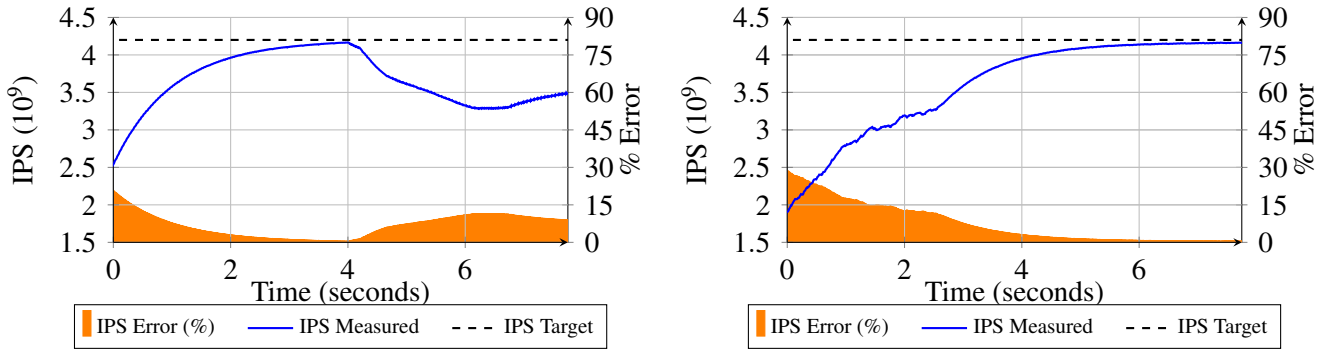
sine wave). The microbenchmark consists of a sequence of independent multiply-accumulate operations performed over both sequentially and randomly accessed memory locations, thus yielding various levels of instruction-level and memory-level parallelism. The range of exercised behavior resembles or exceeds the variation we expect to see in typical embedded workloads, which is the focus application domain of our case studies.

We use the machine-learning `k-means` clustering algorithm as our focus application to evaluate the resource managers. We launch four instances of the algorithm to emulate data-parallel multi-threading.

Model-independence Evaluation

To show SOSA's ability to identify system dynamics from scratch, we study its ability to track a fixed goal for a fixed workload. Our execution scenario consists of a single focus application with an achievable IPS target within the power budget. The goal is to meet target IPS. We compare to a baseline resource manager (BASE) which uses SISO controllers, one for each core. The SISOs have a frequency control-input and IPS measured-output (Frequency→IPS), and are invoked every 10 ms. BASE includes a software supervisor to provide target coordination to the SISOs as well as a simple task migration heuristic. Task migration is performed at the same frequency as low-level controllers are invoked. In all of our evaluation scenarios, we seek to distribute the total system utilization equally among cores. Therefore, our focus applications consist of four threads, and our target for each LCT are one quarter of the total system target. All threads are initially mapped to the same core, so the resource manager is completely responsible for migration. Distribution of system-wide budgets and targets is the subject of orthogonal research, e.g., [9].

Figure 3.10 shows SOSA's and BASE's ability to track a fixed IPS target for a fixed workload. The first 4 s of execution consist of only the focus application `k-means`. Figure 3.10a demonstrates the classical controller's ability to achieve the target performance in an ideal execution scenario, based



(a) BASE instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.

(b) SOSA instructions-per-second (IPS). Error measured as % of the max power possible *below* target.

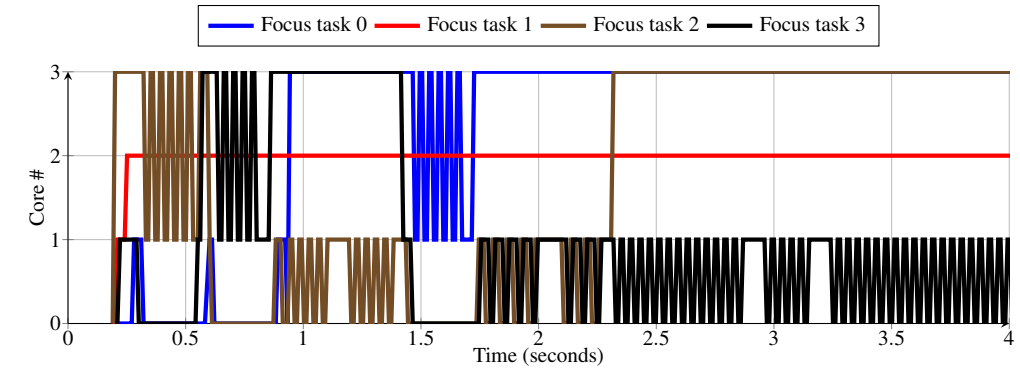
Figure 3.10: IPS tracking for `k-means`. First 4 s consist only of `k-means`. After 4 s, microbenchmarks are added for disturbance. Measured values are smoothed using averaging.

on the system model identified at design time. Figure 3.10b demonstrates the LCT controller's ability to achieve the target performance by learning the system dynamics during execution, without prior workload observation.

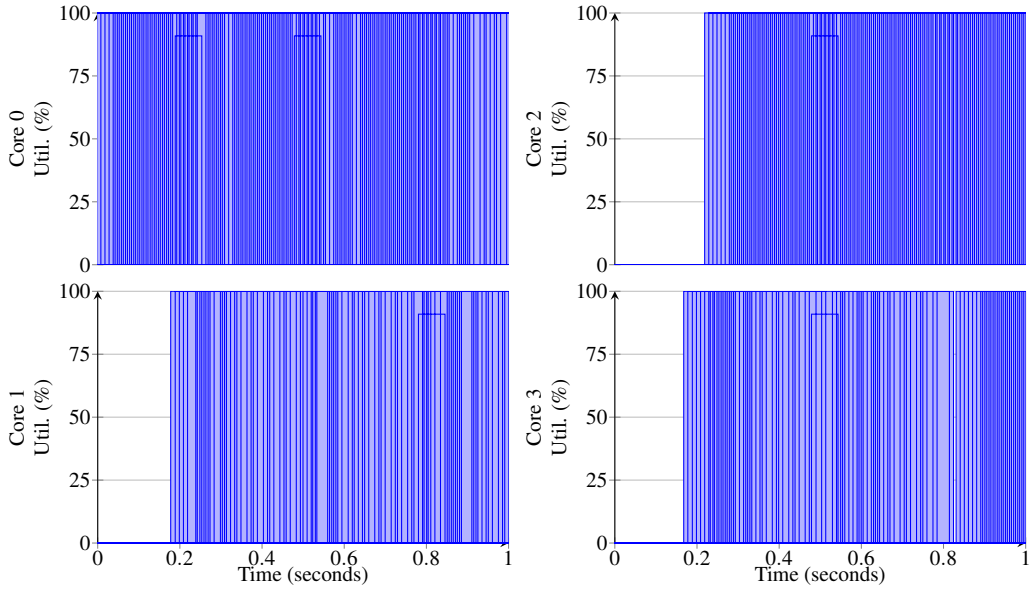
Self-Optimization Evaluation

To show SOSA's ability to identify system dynamics at runtime for unpredictable workloads, we study its ability to track a fixed goal for a dynamic workload. Our execution scenario consists of a single focus application with background tasks entering in the middle of execution (to induce interference from other tasks). The goal is to meet target IPS, which is achievable within the power budget. We compare again to BASE.

Figure 3.10 shows SOSA and BASE ability to track a fixed IPS target for a dynamic workload with tasks coming and going unpredictably. After 4 s of execution, backgrounds tasks are started, consisting of single-threaded microbenchmarks. Figure 3.10a shows that once disturbance is introduced, the controller experiences up to 14 % performance degradation. Figure 3.10b demonstrates the LCT controller's ability to self-optimize to find a configuration that achieves IPS with <5 %



(a) SOSA task migration between cores.



(b) SOSA per-core focus-task utilization.

Figure 3.11: Migration and utilization of SOSA for `k-means`.

error once disturbance is introduced in the form of workload variability, eventually settling with error $<1\,\%$. We confirm SOSA's ability to learn the model from scratch at runtime and optimize in the face of disturbance are in line with our hardware evaluation.

Global Coordination Figure 3.11a shows the migration of focus tasks between cores by SOSA over the first 4 s of execution of the `k-means` benchmark. Each task is represented by a different color line, and the y-axis represents the four different cores. Though it is difficult to see, all tasks are to be initially mapped to core 0. Observe that over time, the migration policy spreads the tasks

among all four cores. There are some exploration periods (e.g., in the first second), and some tasks experience oscillating migration more than others (e.g., task 3 from 2-4 seconds), but the policy accomplishes its goal of spreading the utilization out among the cores. Figure 3.11b shows the focus task utilization of each of the four cores through the first 2 s of execution. Observe that each core is utilized completely for almost the entire execution. After all focus tasks are initialized on the same core, they are migrated within the first 250ms. This confirms that the migration policy achieves its goal.

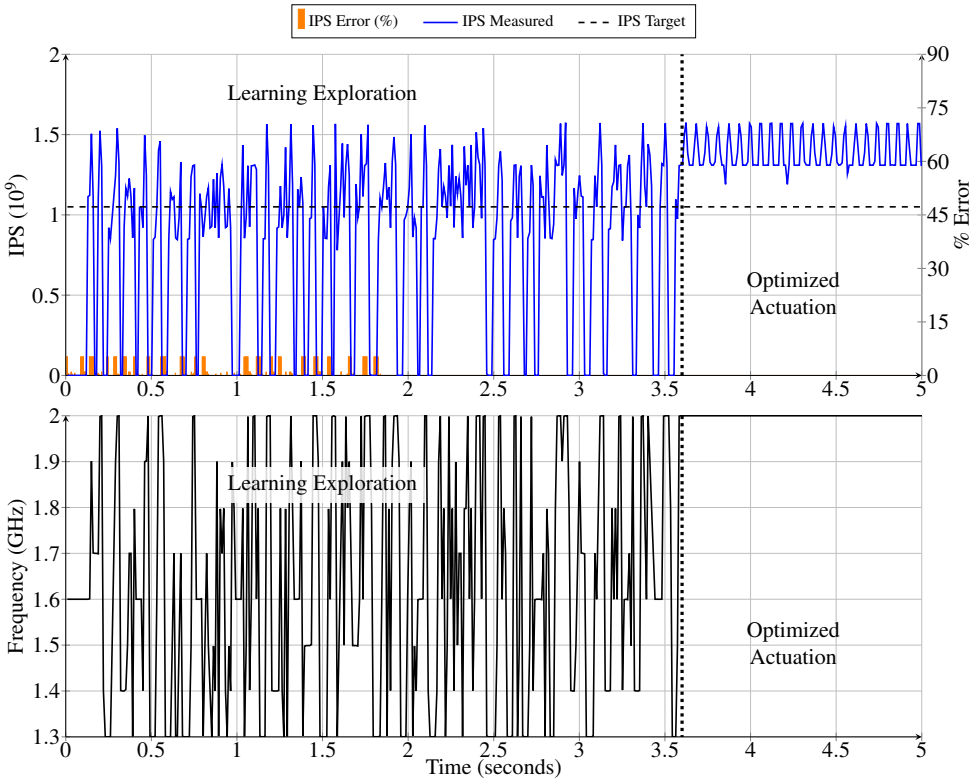


Figure 3.12: Single-core optimization of SOSA for `k-means` over the first 5 s of execution for Core 3. The top plot shows the core IPS achieved, and the bottom plot shows the core frequency.

Local Optimization In Figure 3.12 we take a closer look at the behavior of a single core through the first 5 s of `k-means` controlled by SOSA. Observe the variation in core frequency displayed in the lower plot. The LCT explores the configuration space through the first 3.5 s of execution. After 3.5 s, the LCT settles on the maximum frequency in order to achieve the target IPS for the remainder of execution. This is in line with our system-wide observations made in Figure 3.10b.

Other Benchmarks Figure 3.13 shows the self-optimization evaluation execution scenario for three additional focus applications: linear regression (`linreg`) and k-nearest-neighbors (`knn`) machine learning kernels, and `blackscholes` from the PARSEC [6] benchmark suite.

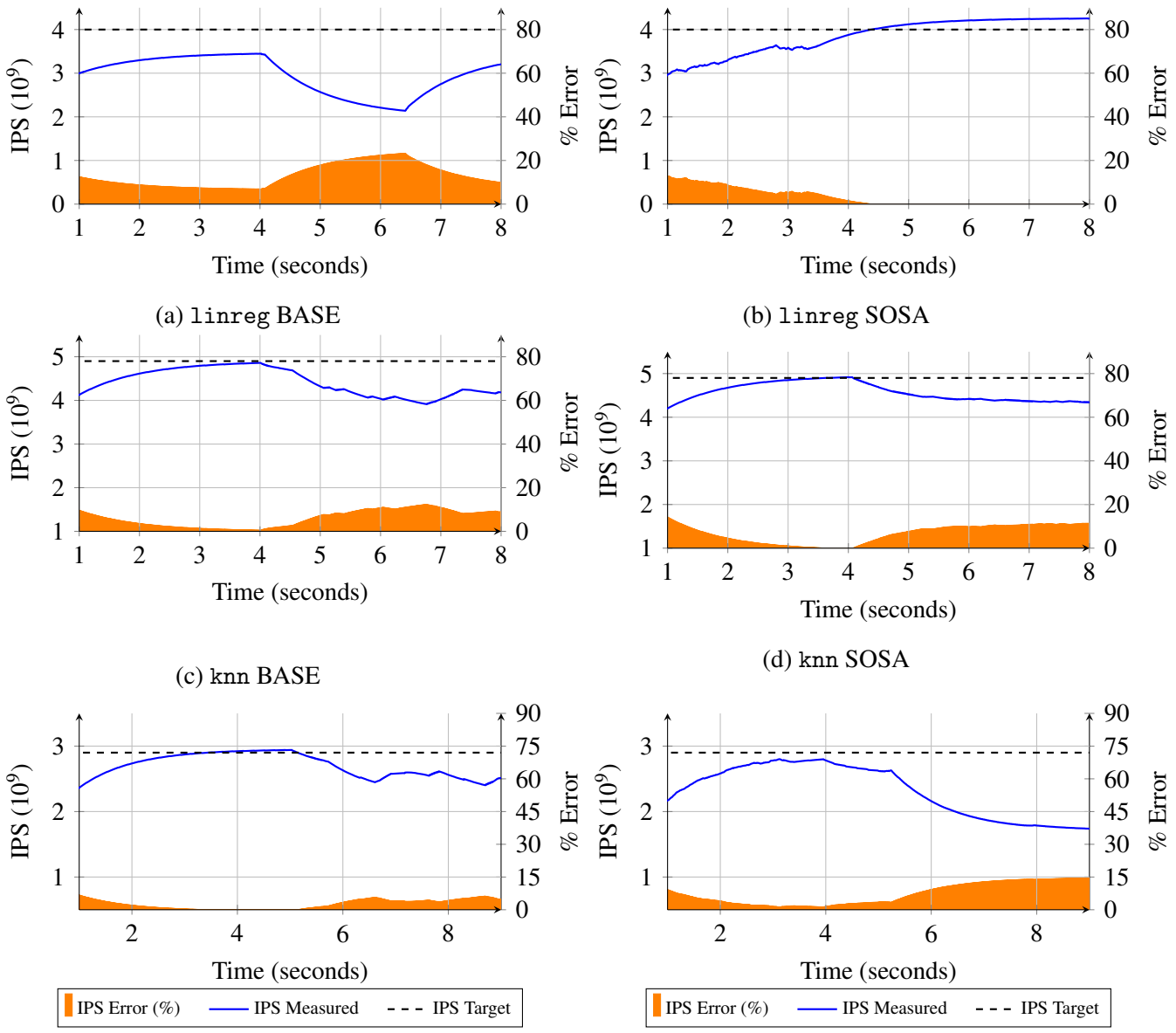
SOSA's result for `linreg` (Figure 3.13b) mirrors that of `kmeans`: learning occurs for the first ˜3 s, after which the target IPS is achievable even through disturbance. The BASE manager struggles significantly with managing through the disturbance (Figure 3.13a). The performance degradation from 4-6 seconds is due to poor migration decisions – the BASE manager eventually recovers at 6 s.

SOSA and BASE perform similarly for `knn` (Figure 3.13d,3.13c): both managers achieve target IPS at a comparable rate in the first 4 s. After disturbance is introduced, both managers are unable to achieve target IPS. Eventually SOSA settle's at 14 % error, and BASE at 9 %.

The results for `blackscholes` (Figure 3.13f,3.13e) are opposite of `linreg`: in this case, SOSA struggles to achieve target IPS once background tasks are introduced. Although BASE also experiences degradation of performance after 4 s, SOSA's error is 2× higher. SOSA's performance degradation from 4-6 seconds is due to poor migration decisions.

3.7 Summary

Self-models are the core components of self-awareness. In computer systems, system dynamics can be complex. When utilizing a self-model at runtime for reflection, models must be simple and sufficiently accurate. The more accurate the self-model, the more effective the decisions made by a resource manager can be toward achieving a given goal. We propose a simple way to improve the accuracy of self-models for resource managers employing classical controllers: gain scheduling. Gain scheduled control generates multiple controllers based on optimized fixed models for different operating regions of the system, and can deploy the most accurate control at runtime



(a) `linreg` BASE

(b) `linreg` SOSA

(c) `knn` BASE

(d) `knn` SOSA

(e) `blackscholes` BASE

(f) `blackscholes` SOSA

Figure 3.13: Additional benchmarks.

based on the system state. This is an improvement over using a single controller based on a single fixed model with minimal overhead. In our first case study, the gain scheduled controller more effectively provides dynamic power management of a single-core processor.

However, using a static model for resource management is not sufficient in complex mobile systems. System dynamics may change between applications or devices, and fixed models may not remain accurate over time. For effective reflection, self-optimization is necessary: the ability to continuously update the self-model based on observation. We improve on classical control methods by building controllers using reinforcement learning, called learning classifier tables (LCTs). The self-models are populated at runtime, and continuously updated. This provides the ability to derive all types of system dynamics, as well as application-specific dynamics. In our second case study, our LCTs show improvement over classical controllers when managing quality-of-service for dynamic workloads.

Chapter 4

Conclusions

In this thesis, we addressed key challenges of adaptive resource management of mobile multi-processors by applying principles of computational self-awareness. We provided self-adaptivity through supervisory control theory, and self-optimization through gain scheduling and reinforcement learning. Our case studies deployed resource managers to manage QoS and power, and demonstrated the promise of incorporating self-awareness. The degree of self-awareness and how it manifests is in a resource management hierarchy is case-specific, and should be carefully considered by the designer: more self-awareness and complexity is not always better. As a result of our initial efforts, we identify opportunities to expand upon:

Coordinating Models at Scale In our SOSA case study, one of the configuration knobs is task migration. The supervisor serves as a central coordination point for sharing global sensor information with the distributed low-level controllers (LCTs), as well as the global decision-making mechanism for migration actions. As mobile systems scale in number of resources, subsystem decomposition in the resource management hierarchy will be crucial for runtime management. However, coordinating all subsystem controllers globally in a single centralized entity becomes impractical: the communication overhead in number of messages and message latency will dom-

inate resource utilization. We must find a way to make decisions in subsystems that have global impact in a manageable way. Costero et al. [12] propose a method for coordinating multiple dependent decision-making reinforcement learning agents in the context of a cloud application. A solution applicable to embedded systems is an open problem.

Updating High-Level Models In LCTs, we demonstrate the utility of continuously updated self-models when deploying low-level controllers. However, in both SPECTR and SOSA, the supervisors are designed with fixed high-level models. Supporting continuously updating self-models at the supervisor level would enable another level of reflection. In fact, such updates would be required for handling dynamic sets of goals. Additionally, high-level model updates could enable self-optimization of tradeoffs between global goals and constrains.

Controlling Application-Specific Metrics We identify the potential of extracting system dynamics between application-level metrics and hardware-level knobs in Section 3.4. However, we did not have the opportunity to explore it further in our use-cases, and it remains an unexplored problem in embedded systems.

Bibliography

[1] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[2] ARM. big.LITTLE Technology: The Future of Mobile. Technical report, 2013.

[3] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini. A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2011.

[4] N. Beckmann and D. Sanchez. Maximizing Cache Performance Under Uncertainty. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[5] M. W. Bertil A. Brandin and B. Benhabib. Discrete Event System Supervisory Control Applied to the Management of Manufacturing Workcells. In *Computer-Aided Production Engineering, C. Venkatesh and J.A. McGeough, eds. (Amsterdam: Elsevier)*, 1991.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[7] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *International Symposium on Microarchitecture (MICRO)*, 2008.

[8] K. K. Chang, A. G. Yağlıkçı, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. *Measurement and Analysis of Computing Systems*, 2017.

[9] Z. Chen and D. Marculescu. Distributed Reinforcement Learning for Power Limited Many-core System Performance Optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[10] S. Choi and D. Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *International Symposium on Computer Architecture (ISCA)*, 2006.

[11] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[12] L. Costero, A. Iranfar, M. Zapater, F. D. Igual, K. Olcoz, and D. Atienza. MAMUT: Multi-Agent Reinforcement Learning for Efficient Real-Time Multi-User Video Transcoding. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[13] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[14] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Application-aware prioritization mechanisms for on-chip networks. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[15] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. AéRgia: Exploiting Packet Latency Slack in On-chip Networks. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[16] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *International Conference on Autonomic Computing (ICAC)*, 2011.

[17] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[18] A. S. Dhodapkar and J. E. Smith. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *International Symposium on Computer Architecture (ISCA)*, 2002.

[19] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016.

[20] B. Donyanavard, A. M. Rahmani, T. Muck, K. Moazemmi, and N. Dutt. Gain Scheduled Control for Nonlinear Power Management in CMPs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[21] B. Donyanavard, A. Sadighi, F. Maurer, T. Mück, A. M. Rahmani, A. Herkersdorf, and N. Dutt. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In *International Symposium on Microarchitecture (MICRO)*, 2019.

[22] C. Dubach, T. M. Jones, and E. V. Bonilla. Dynamic Microarchitectural Adaptation Using Machine Learning. *Transactions on Architecture and Code Optimization (TACO)*, 2013.

[23] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *International Symposium on Microarchitecture (MICRO)*, 2010.

[24] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[25] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-aware shared-resource management for multi-core systems. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[26] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[27] S. Fan, S. M. Zahedi, and B. C. Lee. The Computational Sprinting Game. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[28] X. Fu, K. Kabir, and X. Wang. Cache-Aware Utilization Control for Energy Efficiency in Multi-Core Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[29] U. Gupta, R. Ayoub, M. Kishinevsky, D. Kadjo, N. Soundararajan, U. Tursun, and U. Ogras. Dynamic Power Budgeting for Mobile Systems Running Graphics Workloads. 2017.

[30] M.-H. Haghbayan, A. Miele, A. M. Rahmani, P. Liljeberg, and H. Tenhunen. Performance/Reliability-Aware Resource Management for Many-Cores in Dark Silicon Era. *Transactions on Computers*, 2017.

[31] V. Hanumaiah, D. Desai, B. Gaudette, C.-J. Wu, and S. Vrudhula. STEAM: A Smart Temperature and Energy Aware Multicore Controller. *Transactions on Embedded Computing Systems (TECS)*, 2014.

[32] Hardkernel. ODROID-XU. Technical report, 2016.

[33] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[34] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2007.

[35] J. P. Hespanha. Tutorial on Supervisory Control. In *Lecture Notes for the workshop Control using Logic and Switching for the Conference on Decision and Control*, 2011.

[36] H. Hoffmann. CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[37] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. A generalized software framework for accurate and efficient management of performance goals. In *International Conference on Embedded Software (EMSOFT)*, 2013.

[38] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-aware Computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[39] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[40] A. Jantsch, N. Dutt, and A. M. Rahmani. Self-Awareness in Systems on Chip– A Survey. *Design & Test*, 2017.

[41] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[42] H. Jung, P. Rong, and M. Pedram. Stochastic modeling of a thermally-managed multi-core system. In *Design Automation Conference (DAC)*, 2008.

[43] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A Control-theoretic Approach for Energy Efficient CPU-GPU Subsystem in Mobile Platforms. In *Design Automation Conference (DAC)*, 2015.

[44] A. Kanduri, M.-H. Haghbayan, A. M. Rahmani, P. Liljeberg, A. Jantsch, N. Dutt, and H. Tenhunen. Approximation knob: Power Capping meets energy efficiency. In *International Conference On Computer Aided Design (ICCAD)*, 2016.

[45] C. Karamanolis, M. Karlsson, and X. Zhu. Designing Controllable Computer Systems. In *HoTOS*, 2005.

[46] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[47] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical report, UT Austin, 2010.

[48] D. J. Leith and W. E. Leithead. Survey of gain-scheduling analysis and design. *International Journal of Control*, 2000.

[49] P. R. Lewis, M. Platzner, B. Rinner, J. Torresen, and X. Yao. *Self-aware Computing Systems*. Springer International Publishing, 2017.

[50] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *International Conference on Computer Design (ICCD)*, 2013.

[51] L. Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, 1999.

[52] L. Ljung. Black-box models from input-output measurements. In *Instrumentation and Measurement Technology Conference (IMTC)*, 2001.

[53] D. Lo, T. Song, and G. E. Suh. Prediction-guided Performance-energy Trade-off for Interactive Applications. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[54] K. Ma, X. Li, M. Chen, and X. Wang. Scalable power control for many-core architectures running multi-threaded applications. 2011.

[55] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *Conference on Decision and Control (CDC)*, 2010.

[56] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh. Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[57] MathWorks. System Identification Toolbox. Technical report, 2017.

[58] A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das. Cpm in cmps: Coordinated power management in chip-multiprocessors. In *SC*, 2010.

[59] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[60] S. Morse. *Control using logic-based switching*. Springer, 1977.

[61] T. Mück, S. Sarma, and N. Dutt. Run-DMC: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2015.

[62] T. S. Muthukaruppan, A. Pathania, and T. Mitra. Price Theory Based Power Management for Heterogeneous Multi-cores. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[63] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In *Design Automation Conference (DAC)*, 2013.

[64] H. Nejatollahi and M. E. Salehi. Effect of voltage scaling on symmetric multicore's speed-up. In *Iranian Conference on Electrical Engineering (ICEE)*, 2014.

[65] H. Nejatollahi and M. E. Salehi. Voltage scaling and dark silicon in symmetric multicore processors. *The Journal of Supercomputing*, 2015.

[66] H. Nejatollahi and M. E. Salehi. Reliability-Aware Voltage Scaling of Multicore Processors in Dark Silicon Era. *Advances in Parallel Computing*, 2018.

[67] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker. Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[68] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas. Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[69] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas. Yukta: Multilayer Resource Controllers to Maximize Efficiency. In *International Symposium on Computer Architecture (ISCA)*, 2018.

[70] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-performance modeling on asymmetric multi-cores. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013.

[71] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *International Symposium on Computer Architecture (ISCA)*, 2008.

[72] A. M. Rahmani, B. Donyanavard, T. Mück, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt. SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[73] A. M. Rahmani, M. H. Haghbayan, A. Kanduri, A. Y. Weldezion, P. Liljeberg, J. Plosila, A. Jantsch, and H. Tenhunen. Dynamic power management for many-core platforms in the dark silicon era: A multi-objective control approach. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[74] A. M. Rahmani, M. H. Haghbayan, A. Miele, P. Liljeberg, A. Jantsch, and H. Tenhunen. Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era. In *Transactions on Very Large Scale Integration Systems (TVLSI)*, 2017.

[75] A. M. Rahmani, A. Jantsch, and N. Dutt. HDGM: Hierarchical Dynamic Goal Management for Many-Core Resource Allocation. 2017.

[76] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. In *Proceedings of the IEEE*, 1989.

[77] M. H. Safanov. *Focusing on the knowable: Controller invalidation and learning*. Springer, 1997.

[78] J. S. Shamma and M. Athans. Analysis of gain scheduled control for nonlinear plants. *Transactions on Automatic Control*, 1990.

[79] E. Shamsa, A. Kanduri, A. M. Rahmani, P. Liljeberg, A. Jantsch, and N. Dutt. Goal-Driven Autonomy for Efficient On-chip Resource Management: Transforming Objectives to Goals. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.

[80] shekhar Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: Controlled shared cache management in chip multiprocessors. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[81] K. Singh, M. Bhadauria, and S. a. McKee. Real time power estimation and thread scheduling via performance counters. *Computer Architecture News*, 2009.

[82] S. Skogestad and I. Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons, 2005.

[83] B. C. Smith. *Reflection and Semantics in a Procedural Programming Language*. Phd, MIT, 1982.

[84] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[85] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[86] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *International Symposium on Microarchitecture (MICRO)*, 2015.

[87] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[88] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive Control of Approximate Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[89] P. Tembey, A. Gavrilovska, and K. Schwan. A Case for Coordinated Resource Management in Heterogeneous Multicore Platforms. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[90] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, 2008.

[91] J. Thistle. Supervisory control of discrete event systems. In *Mathematical and Computer Modelling*, 1996.

[92] V. Vardhan, W. Yuan, A. F. Harris, S. V. Adve, R. Kravets, K. Nahrstedt, D. Sachs, and D. Jones. GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy. 2009.

[93] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[94] X. Wang, K. Ma, and Y. Wang. Adaptive power control with online model estimation for chip multiprocessors. *IEEE TPDS*, 2011.

[95] X. Wang and J. F. Martínez. ReBudget: Trading Off Efficiency vs. Fairness in Market-Based Multicore Resource Allocation via Runtime Budget Reassignment. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[96] Y. Wang, K. Ma, and X. Wang. Temperature-constrained Power Control for Chip Multiprocessors with Online Model Estimation. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[97] S. W. Wilson. ZCS: A zeroth level classifier system. *Evolutionary computation*, 2(1):1–18, 1994.

[98] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary computation*, 3(2):149–175, 1995.

[99] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebook's Data Center-Wide Power Management System. In *International Symposium on Computer Architecture (ISCA)*, 2016.

[100] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[101] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ACM SIGARCH Comp. Arc. News*, 2004.

[102] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh, and D. W. Clark. Formal control techniques for power-performance management. 2005.

[103] K. Yan, X. Zhang, J. Tan, and X. Fu. Redefining QoS and customizing the power management policy to satisfy individual mobile users. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[104] J. Zeppenfeld, A. Bouajila, W. Stechele, and A. Herkersdorf. Learning Classifier Tables for Autonomic Systems on Chip. *GI Jahrestagung (2)*, 134:771–778, 2008.

[105] H. Zhang and H. Hoffmann. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.