UNIVERSITY OF CALIFORNIA,
IRVINE

Runtime Resource Management of Emerging Applications in Heterogeneous Architectures

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Kasra Moazzemi

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Alex Nicolau
Professor Fadi Kurdahi

2020

DEDICATION

In memory of my Grandmother and Grandfather. Both left fingerprints of grace on my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my adviser, Professor Nikil Dutt. Nik you not only have been a great PhD advisor but also a fantastic mentor in life. Your guidance and positive perspective towards contributing to society has helped me to grow as a person.

I would like to thank the rest of my dissertation committee members and mentors Professor Alex Nicolau, Professor Fadi Kurdahi, Professor Rainer Domer and Professor Amir Rahmani for their time, support and invaluable advice.

For my mother and father who helped me in all things great and small.

I am deeply thankful to my brother and sister for their love, continued support, and sacrifices.

I thank my friends and colleagues at UCI, who made a great impact on my life. In particular, I thank Hamid, Majid, Hossein, Donny, Tiago, Roger, Emad and Deep for their friendship, feedback, guidance, and collaborations

I must thank my dear friends Mahdi, Ashkan, Saman, Mohammad, Delaram, James, Hamid, Saman, Balint, Mozafar, Amin and Zoya for their support and the good memories we have created together

Finally, my work would not have been possible without funding from the Department of Electrical Engineering and Computer Science and Donald Bren School of Information and Computer Sciences as well as the NSF (grant CCF-1704859), or permission from the ACM, IEEE, Elsevier and now publisher inc to include content from my previously published work in [150, 178, 144, 157, 196, 195, 47, 187, 149, 151, 182].

VITA

Kasra Moazzemi

EDUCATION

Doctor of Philosophy in Computer Engineering **2020**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Engineering **2014**
Northeastern University *Boston, MA*

Bachelor of Science in Computational Sciences **2011**
Shahid Beheshti University *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2014–2020**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management **2019**
ACM Transactions on Embedded Computing Systems (TECS)

On-chip dynamic resource management **2019**
Foundations and Trends® in Electronic Design Automation

Design methodology for responsive and robust MIMO control of heterogeneous multicores **2018**
IEEE Transactions on Multi-Scale Computing Systems

On the feasibility of SISO control-theoretic DVFS for power capping in CMPs **2018**
Microprocessors and Microsystems

REFEREED CONFERENCE PUBLICATIONS

Spectr: Formal supervisory control and coordination for many-core systems resource management **2018**
International Conference on Architectural Support for Programming Languages and Operating Systems

Dependability evaluation of SISO control-theoretic power managers for processor architectures **2017**
IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)

Gain scheduled control for nonlinear power management in CMPs **2018**
Design, Automation and Test in Europe Conference and Exhibition (DATE)

Design methodologies for enabling self-awareness in autonomous systems **2018**
Design, Automation and Test in Europe Conference and Exhibition (DATE)

Trends in On-Chip Dynamic Resource Management **2018**
Digital System Design

The information processing factory: a paradigm for life cycle management of dependable systems **2019**
International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)

HAMEX: heterogeneous architecture and memory exploration framework **2016**
International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype

ABSTRACT OF THE DISSERTATION

Runtime Resource Management of Emerging Applications in Heterogeneous Architectures

by

Kasra Moazzemi

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2020

Professor Nikil Dutt, Chair

Runtime resource management for heterogeneous computing systems is becoming more and more complex as workloads in these platforms get increasingly more diverse and the conflicts grow between heterogeneous architectural components and their resource demands. The goal of these runtime resource management mechanisms is to achieve the overall system goal for dynamic workloads while coordinating system resources in a robust and adaptive fashion.

To address the complexities in heterogeneous computing systems, state-of-the-art techniques that use heuristics or machine learning have been proposed. On the other hand, conventional control theory can be used for formal guarantees, but may face unmanageable complexity for modeling system dynamics when dealing with heterogeneous computing platforms.

In this thesis, we initially analyze a variety of runtime resource management methods and introduce a classification for these methods capturing the utilized resources and metrics. We cover heuristic, machine learning and control theory methods used to manage resources such as performance, power, energy, temperature, Quality-of-Service (QoS) and reliability of the system.

In addition, we explore a variety of dynamic resource management frameworks that provide significant gains in terms of self-optimization and self-adaptivity. This includes simulation

infrastructures, hardware platforms enhanced with multi-layer management mechanisms and corresponding software frameworks that enable management policies for these systems in an effective and adaptive manner.

Ultimately, we address the problem of optimizing energy efficiency, power consumption, performance and QoS in heterogeneous systems by proposing adaptive runtime policies. The proposed methods in this thesis, take into account the constraints and requirements defined by user, dynamic workloads and coordination between conflicting objectives.

The projects presented in this dissertation show effectiveness in responding to abrupt changes in heterogeneous computing systems by dynamically adapting to changing application and system behavior at runtime, and are thus able to provide significant improvement compared to commonly used static resource management methods.

Chapter 1

Introduction

Dynamic resource management has been established as an effective technique to improve performance, efficiency, and reliability of computer systems [177]. Adaptability of these resource management methods becomes more important with modern multicore and many-core systems due to diverse workloads running in parallel which exhibit dynamic behavior at runtime. This dynamic behavior manifests itself across the homogeneous system as varying shared resource demand, bottleneck or sometimes even conflicting goals. This issue exacerbates when heterogeneity is applied to our computing resources similar to emerging heterogeneous multicore processors (HMPs) or platforms using CPU, GPU and other accelerators along side each other on a single chip.

In this context, computer architects use several approaches to perform dynamic resource management. Machine learning methods learn the best input values for different observed conditions. Model-based and Rule-based heuristic methods use a model or an encoded algorithm to make decisions during runtime. Optimization methods minimize/maximize an objective while considering certain constraints. Finally, control theoretic techniques, use their intrinsic feedback loop to adapt to conditions.

In this thesis, we initially present a classification of methods used in dynamic resource management which highlights the trend in utilization of self-adaptivity and self-awareness concepts in designing policies for resource management of modern System on Chip (SoC) platforms. Thereafter, we propose methods with inherent adaptivity and scalability to address challenges of dynamic resource management in heterogeneous systems.

1.1 Challenges

The evolution in computing resources from single- to multi- and many-core platforms and eventually to heterogeneous systems with variety of processing elements and accelerators has opened the path for accommodating more complex and diverse set of workloads. This manifestation of heterogeneity in architecture, applications and objectives leads to many challenges:

- Increasing complexity in architecture:
  - Higher operating frequency
  - Scale up in number of cores
  - Heterogeneity in computing resources
- Diverse workloads with higher order of parallelism
- There is a renewed move towards programmable architectures tuned to certain application domains (machine learning and neural networks)
- Coordination of multiple objectives that are subject to change at runtime while satisfying several constraints

1.2 Key properties in dynamic resource management

As computer architects trying to address challenges described in Section 1.1, we should be able to answer some key questions in order to achieve desired properties for managing resources for a modern heterogeneous system.

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

In this thesis, we focus on exploring methods that can partially or fully address these questions.

1.3 Thesis Statement and Organizations

This dissertation explores four problems in runtime resource management of computer systems: (1) defining system goals in policies using designer's expertise (2) Providing formal guarantees regarding robustness of the system (3) coordinating multiple objectives such as performance and energy efficiency (4) adapting to application or system changes at runtime.

The rest of this dissertation is organized as follows. We position our work with related efforts in Chapter 2. Chapter 3 dives deeper into questions discussed in Section 1.1 by presenting a

classification on dynamic resource management methods and discussing their pros and cons. We present our body of work in terms of simulation, hardware and software frameworks in Chapter 4. Chapter 5 outlines case studies done to highlight our efforts to enable adaptive in dynamic resource management. Finally, Chapter 6 concludes this thesis.

Chapter 2

Background and Related work

We start with an inventory of *resources* that are subject to resource management. Three sub-categories of resources can be identified according to the functionality they provide:

- **Computation resources** are processing elements which perform tasks.
- **Communication resources** are utilized by tasks to exchange information with other tasks or the environment.
- **Memory resources** are used by tasks to store and retrieve data.

Resources are physical hardware blocks that can be allocated to tasks by resource management. Allocation choices of resource management are binary: a resource is either allocated to a task at a time or not. However, a resource may have the capacity to serve more than one task if the hardware is provided. Each resource has a maximum capacity of tasks that it can serve at a given time, which depends on its hardware structure. For instance, a CPU with two ALUs could process two instructions at the same time.

Control decisions of resource management also tune the operation of resources. The operation

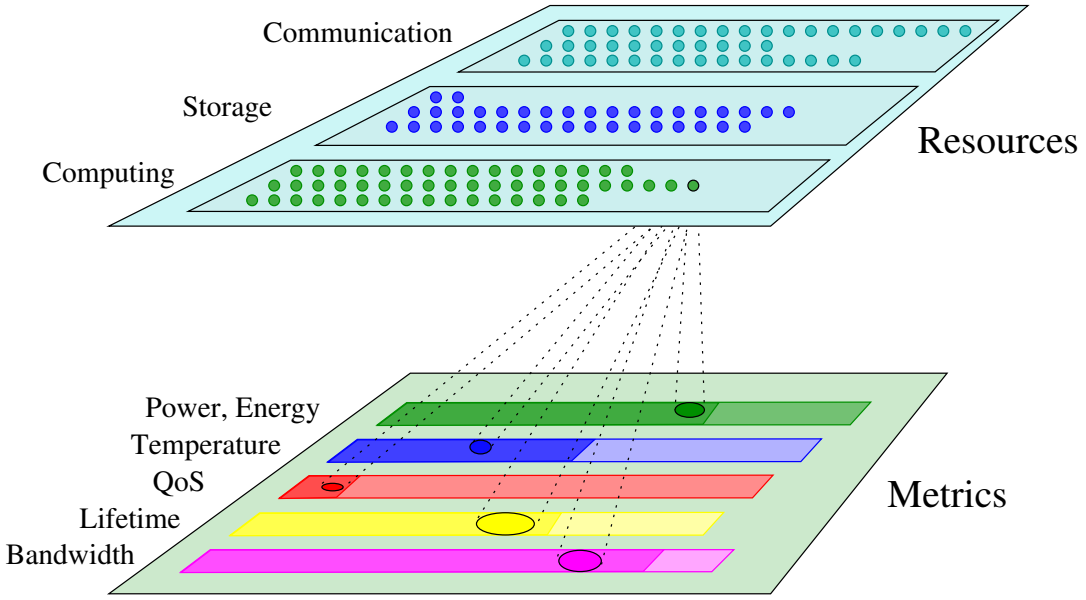


Figure 2.1: Capturing the relationship between resources and metrics. The upper plane represents resources while the lower shows metrics. The allocation and usage of each resource throws shadows into the metrics plane representing how it contributes to the operation with respect to different metrics.

of the system can be characterized by various *metrics*, see Fig. 2.1. Note that metrics characterizing one resource are typically interdependent. For example, scaling up frequency of a computation resource results in higher power dissipation and temperate but also increases execution speed.

Resource management controls resources by allocating tasks and tuning operation parameters. All actions of resource management are based on the observed operation of the system, that is metrics. By allocating and controlling resources properly, resource management steers the system to meet objectives for optimizing some metrics. Negative consequences of over-utilizing the system needs to be avoided, too. While aiming at meeting requirements with respects to objective metrics, resource management takes constraints on other metrics into account. Heuristics that aim at containing *constraint metrics* below or above given limits are needed for realizing proper resource management. Note that metrics cannot be generally sorted into groups of objectives and constraints as their role is dependent on the considered resource management technique.

2.1 Resources

Resources are physical entities that can be allocated to tasks by resource management. Control decisions of resource management also tune the operation of resources, which affects operation of the system and hence the metrics.

Allocation choices of resource management are binary: a resource is either allocated to a task at a time or not. However, a resource may have the capacity to serve more than one task if the hardware is provided. Each resource has a maximum capacity of tasks that it can serve at a given time, which depends on its hardware structure. For instance, a CPU with two ALU could process two instructions at the same time.

Examples of resources are cores, buses, IO pins, links, and memory locations. A link between two routers can be allocated to only one packet at any given time. As another example, a router in a communication network can only process four packets simultaneously coming from its four input ports. An on-chip network may be used by several tasks simultaneously at an abstract level but each physical component such as a register, a link, or a switch-box can serve only one or a limited number of packets at any moment. The network as a whole can transmit many packets and it can be considered as one communication resource with a maximum capacity of C bits/sec. Fractions of C can be allocated to individual users of the network by resource management which initiates communication over the network but the actual route and scheduling of the transmission is managed by the routers.

Table 2.1: Examples of resources

Category	**Examples**
Computation resources	cpu, gpu, dsp, fpga
Communication resources	buses, networks, I/O pins, interrupt controllers
Memory resources	main memory, cache, register file

Resources can be divided into 3 sub-categories (Table 2.1) according to the functionality they

provide as follows.

Computation resources are processing elements (PEs) which perform tasks. A PE can be a General purpose GPU, a dedicated accelerator e.g., GPU and DSP, or reconfigurable e.g., FPGA, that are able to serve different purposes. Different PE support various control features which are to be decided by the resource management. Processing element may support different dynamic power management (DPM) techniques, some provide hardware knobs for tuning the precision of arithmetic computations, reconfiguration is a point of control for reconfigurable PE. Computation resources may be managed on the core level, but are often treated on a more abstract level as a pool of cores for particular resource management techniques.

Communication resources are utilized by tasks to exchange information with other tasks or the environment. Communication resources include buses, networks, IO pins, and interrupt controllers. On-chip networks are complex resources, which are built up from several physical blocks implementing the functionality. Those internal resources, like buffers and routers, must be revealed for efficient resource management. Nevertheless, network-on-chips (NoC) are also managed as a whole without considering actual hardware blocks. Communication resources may support various control features e.g., DVFS for NoC.

Memory resources are used by tasks to store and retrieve data. Memory resources are typically organized in a hierarchical structure. some storage resources, e.g. on-chip main memory and off-chip memories, are under software control, while others, e.g. the cache hierarchy, operate in a hardware-defined manner. Off-chip resources are out of the scope of this survey. It is also noteworthy that while off-chip memory resources are under software control, interfaces between them might involve logic beyond that control e.g., hardware-controlled buffers and DMA facilities. DPM techniques, typically power gating, may be supported by

memories.

2.2 Metrics

Non-functional characteristics of the system are represented in various metrics: scales that provide means to evaluate particular aspects of the operation of the system. The metrics considered in this chapter are

- performance.
- power, energy, temperature.
- reliability.
- QoS.

About the connection between resources and metrics, note that metrics characterizing one resource are typically interdependent. For example, scaling up frequency of a computation resource results in higher power dissipation and temperate but also increases performance.

2.3 Objectives and constraints

The subjects of resource management are the resources which are directly controlled by allocating tasks and tuning their operation. However, all actions of resource management are based on metrics that characterize the operation of the system.

Resource management makes its allocation choices and control decisions with respect to objectives to be accomplished. The objectives are defined as maximizing or minimizing some

metrics. By allocating and controlling resources properly, resource management steers the system to meet requirements and target goals on *objective metrics*.

Allocation and control decisions of resource management might also have negative consequences. Particularly, over-utilizing the system, leading to excessive energy consumption and higher temperature, degrades system health and affects desired objectives negatively. While aiming at meeting requirements with respect to objective metrics, resource management must also take constraints on other metrics into account. Proper resource management is based on heuristics aiming at containing some metrics below or above given limits and optimizing others.

Note that metrics cannot be generally sorted into groups of objectives and constraints as their role is dependent on the considered resource management technique. The surveyed techniques are structured according to their objective metrics.

2.4 Observing and Predicting

Resource management is to optimize and contain some problem-specific characteristics of the system, that is metrics. Decisions need to be made so that metrics exhibit a desirable behavior in the future. Also, decisions can be made based on the current and possibly past states of the system.

The current state of the system is observed by sensors and future behavior is predicted by models. Note that the terms sensing and monitoring are used in literature with similar meaning to observation here.

Sensors provide means for resource management to observe the current state of the system as in current values of metrics. Beyond taking notice of the current state, historical data of

metrics can be collected over time.

We can identify physical sensors (e.g., power sensor, temperature sensor, hardware performance counter) and cyber sensors (e.g., abstract QoS, instrumented software, logs). Example of physical sensors can be the sensors used in SoCs that require several die temperature sensors to be integrated in a chip to manage the performance because die temperature directly affects leakage current level and performance of clock-based digital circuits. Cyber sensors can provide a standard method for an application to directly communicate its performance and goals. As an example, [133] provide a framework that allows applications to express their performance in terms of a desired heart rate and/or a desired latency between specially tagged heartbeats.

Models are used to predict the future behavior of the system based on collected state information and planned control actions. Resource management uses models to evaluate possible control actions with respect to past behavior, current state, and desired future behavior of the system.

Models may be used explicitly or implicitly in resource managers. An explicit model is present as a component of the resource manager and does compute its predictions whenever required. Different models may provide different accuracy because of predicting with different levels of details being taken into account. The more the details and accuracy, the more the computational complexity. The trade-off between accuracy of prediction and computational complexity is to be tuned with respect to the problem at hand.

In some cases, accurate-enough and yet relatively simple rules can be derived from abstract – perhaps simplified or estimate – models. Resource management may use such rules directly as part of the decision mechanism. Instead of implementing the underlying model explicitly, the model – as well as prediction – is implicit behind the incorporated rules. Such rules are

called heuristics.

In order to better classify the related work to our scope of runtime resource management, following categorization based on metrics such as power, thermal and energy is used. In addition, in Chapter 3 we highlight some of the major efforts based on the methods utilized in each work. A comprehensive survey of the related work can be found in [144].

2.5 Dynamic power management

Dynamic power management(DPM) techniques have been proposed decades ago. Designers used DPM in the 90s [32] with the available run-time configurations such as scaling the supply voltage to lower the power consumption [159]. Most opportunities to reduce power consumption comes from non-optimal configurations in hardware and software components.

Before diving into the analysis of efforts done in this domain, it is important to make a distinction between power-aware and low-power systems. The focus of low-power systems design is to minimize power. On the other hand, for a power-aware system meeting power and energy goals is a significant design consideration and in which the system modifies its behavior based on current power/energy availability. Some power-aware design goals may even increase power or energy consumption. Consider the case of a design for decreasing peak power in a processor: one method to attain this goal would be to use schemes that would intentionally delay the issue of some instructions to smooth the instruction issue distribution and, thus, decrease the peak consumed power. However, delaying some instructions could lead to the application being finished later than it otherwise would, therefore increasing the energy consumption. Thus, this scheme would be a power-aware, but not a low-power, design [222]. In the following, we mainly focus on DPM for power-aware systems based on the layer (hardware or software) that each technique mainly resides. This categorization is summarized

in Table 2.2.

Table 2.2: DPM techniques

Power management	Techniques
Hardware	[104] [86] [28] [116] [108] [128] [123] [29] [174] [84] [9] [170] [196] [24] [179] [163] [162] [110] [102] [33] [72] [38]
Software	[121][201] [80] [109] [229]

2.5.1 Hardware methods

One of the main techniques used in DPM is to use the inherent properties of hardware components to reduce power consumption. This can be achieved by reducing the voltage, frequency or even shutting down the processing units.

DVFS and power gating

DVFS is a common method in DPM. The computing unit (or communication media) must be augmented with hardware blocks that allow for changing the supply voltage dynamically. This is common in recent processors. Although, the definition in [104] proved that task allocation and scheduling optimization problems using DVFS are NP-complete, heuristics to tune voltage and frequency are widespread because they save substantial power. However, reducing the frequency causes a slowdown in the execution of programs with potentially

detrimental effects. Thus, DVFS heuristics usually trade off power savings against delay. One of the earliest works on DVFS has been presented in [238] in which they propose a method to manage frequency and voltage which is called the *Average Rate* heuristic which sets the speed of the processor to the sum of average rate requirements of tasks in the frame. Theorems proposed for power-delay optimization in [86] utilize an ILP problem to minimize energy consumption under an execution time constraint. Approach approach [116] partitions each task into time slots which enables DPM to change voltage at specific intervals. A hybrid method is used by [108] using two algorithms consisting of an online phase, in which voltage settings are selected to reduce energy consumption assuming that tasks complete in their WCET. Considering that many tasks finish well before their WCET, this method uses an online phase which adjusts the voltage settings on-the-fly to reclaim any resources released by such tasks. A mechanism proposed by [128] performs variable-voltage scheduling via efficient slack time re-allocation, which helps reducing the average discharge power consumption as well as smooths the discharge power profile.

Authors of [123] focused on power-aware scheduling in mission critical embedded systems. Their approach is incremental by solving one type of constraint at a time. First, a time-valid schedule is constructed from a constraint graph of the task. Next, this schedule is validated against the maximum power constraint to remove power spikes, and finally it is compared against the minimum power constraint and tasks are reordered to reduce power gaps and power utilization. Mechanism in [29] employed a fine-grained offline scheduling approach that saves power by combining multiple instructions into one complex instruction with lower power consumption, or by using low-power versions of instructions while considering task deadlines. Authors in [174] proposed a DPM method using Markov decision processes. The problem of DPM in such a system is formulated as a policy optimization problem and solved using an efficient "policy iteration" algorithm. Idea of managing power of voltage islands in SoC has been well studied [110]. In this domain, partitioning method in [72] shows the trade-offs involved in the choice of both DVFS control scheme and method by which the

processor is partitioned into voltage/frequency islands and presents potential in using DVFS for dynamic power management in CMPs. Cochran et al. [33] proposes a control technique to make DVFS and thread packing control decisions in order to maximize performance within a power budget using a multinominal logistic regression classifier. Das et el. [38] proposes an approach for DVFS in smartphones, which uses reinforcement learning to explore the trade-off between power saving opportunities using DVFS and dynamic core selection and application's performance at run-time. Architecture-independent imitation learning methodology is proposed in [102] for DVFI control in many-core systems by using controllers that leverage the structural relationships between VFIs.

Another approach [84] monitored the run-time application in order to optimize power consumption by setting per core DVFS using a global system manager. Azevedo et al. [9] employed an intra-task dynamic voltage scaling technique under compiler control using program checkpoints. Checkpoints are generated at compile time and indicate places in the code where the processor speed and voltage should be recalculated. These checkpoints are used at run-time to recalculate voltage and frequency settings. To reduce the overhead of dynamic scaling [170] introduced DVFS in operating systems. Since then, the majority of operating systems have simple settings to benefit from DVFS even in embedded devices. A feedback control was used by [196, 195] to manage the power consumption using DVFS knobs.

The trend towards multi/many-core platforms requires techniques that can formally guarantee power management of the system given a power budget. Bogdan et al. [24] proposed a paradigm shift from power optimization based on linear models to control approaches based on fractal-state equations. [179] developed a multi-objective DPM method that simultaneously considers limits on the total power consumption, dynamic behaviour of workloads, processing elements utilization, per-core power consumption, and the load on the NoC. This work uses fine-grained voltage and frequency scaling, including near-threshold operation, and per-core

power gating to optimize the performance and power consumption. In addition, a disturbance rejecter is designed that proactively slows down running applications when a new application commences execution, to prevent sharp power budget violations. Furthermore, authors in [47] propose a gain scheduling method to deal with non-linearity in DPM. Tilli et al. [218] proposes a low overhead hierarchical model-predictive controller (MPC) for managing thermally safe sprinting with predictable resprinting rate, which ensures the correct execution of mixed-criticality tasks. A methodology for multi-clock/voltage domains is proposed by [163, 162] by adaptively partitioning and voltage assignment using state-space feedback control strategy to dynamically scale the operating voltage and frequency around the static values and load balance the network traffic in the presence of workload and parameter variations. Another approach [28] presents an On-line Distributed Reinforcement Learning (OD-RL) based DVFS control algorithm for many-core system performance improvement under power constraints that uses per-core reinforcement learning method for frequency management and a control theoretic method for global power budget allocation. This method takes advantage of fast response of the per-core reinforcement learning while making sure the global power consumption remains under the budget using Maximize-the-Max method. Muck et al. [157] proposes Multi-Input Multi-Output (MIMO) for controlling various actuators in HMP in order to control both power consumption and overall system performance. SPECTR [178] is proposed as a supervisory control approach (SCT) to manage power along side other objectives in emerging systems. Moazzemi et al. [151] propose using fuzzy control theory for DPM in heterogeneous systems.

2.5.2 Software methods

Software power management techniques can be categorized into two closely related areas of research. First, different studies explored the properties of workload variations and developed methods to identify and follow different execution behavior, commonly referred to as “phase

analysis". Second, a large complementary set of research studied dynamic, on-the-fly system management techniques that can adaptively respond to these differences in application behavior (e.g. [85]).

Task scheduling and thread optimization

DPM mechanisms in real-time systems become more complex as the system has to meet certain deadlines while keeping the power below a certain budget. A synthesis algorithm by [121] uses a software-based cache partitioning and reservation technique to guarantee cache hits for some tasks and therefore improve task schedulability. The scheduling algorithm used in this work is Earliest Deadline First (EDF). In EDF, the task with the earliest deadline has the highest priority. The method proposed by [201] and implemented in a kernel module, yields power reduction by exploiting slack times, both those inherent in the system schedule and those arising from variations of execution times. As an example, Huange et al. [80] proposes adaptive DPM for hard real-time systems. In their work, based on real-time calculus, event arrivals and resource services are modeled by arrival curves and service curves in the interval domain, respectively, and an online algorithm to adaptively control the power mode of the device is proposed, that postpones the processing of arrival events as long as possible. In a similar spirit, many cache aware methods have been proposed, such as [109], to benefit from software optimizations to reduce power consumed by cache and memory subsystems. Further analysis of system-level power-aware design techniques is presented by [222] who cover techniques ranging from the circuit and device level, to the architectural, compiler, operating system, and networking layers. In the case of many-core systems, scalability of scheduling algorithm becomes an important factor which [229] analyzes some of these algorithms in terms The computational complexities of thread scheduling and global power management techniques.

2.6 Run-time energy efficient managers

Although for many computer systems, reducing power will lead to reduced energy consumption but it does not necessarily mean a resource management mechanism with the objective of power management will deploy same policies as a resource management method with energy efficiency goal. Energy directly relates to both *power* and *performance*. Final goal for energy efficient management methods is to find the optimal spot in power consumption while delivering the required performance over time. To this end, we take a closer look at energy efficient managers in this section. Computer systems are designed to deliver peak performance, but are often idle or used to perform tasks that do not require such performance.

Energy efficiency has become a major concern while dealing with high performance computing systems [56, 78]. Architectural optimization to achieve a high performance with minimal power consumption has been a common practice for emerging applications. Approach in [175] proposed a new pipelining mechanism with selectable voltage for each pipeline stage to minimize energy consumption. Evaluation of energy efficiency of a system can be related to both power consumption and performance of the running application. A common metric for the evaluation of energy efficiency is EPI, in Watt/MIPS or Joule/Instruction. Other metrics such as Energy Delay Product (EDP), which was initially proposed by [77], and ED2P are used also in latency performance architectures as they assign a weight to the amount of time needed for an instruction to be processed [7].

Authors in [137] proposed task scheduling algorithms that minimize energy or minimize power for the case when the tasks have various arrival times, deadline times, execution times and switching activities. The relation between the operating voltages for the minimum energy (power) assignment is determined theoretically and a polynomial time scheduling algorithm that uses this relation is developed to minimize energy consumption. The authors improved this method [138] by first applying the existing task scheduling algorithms [137] to

obtain a feasible schedule and then distribute the available slack using an iterative algorithm that satisfies the theoretically obtained relation for minimum energy. Shafique et al. [194] considered self-adaptive many-core systems to reduce the energy-delay^2 product. To avoid frequent allocation and de-allocation, this work enables applications to temporarily reserve their resources and to perform local power management decisions.

Diversity and complexity in HPC systems require specific solutions for DPM, as presented by [186] for server clusters and by [14] for cloud systems. Method in [81] targeted multimedia applications by using both architectural adaptation and dynamic voltage scaling. Similarly, Unsal et al. [221] proposed two complementary media-sensitive energy-saving techniques that leverage static information. First, a compiler-controlled data remapping scheme directs scalar accesses to a small scratchpad SRAM area. Second, a media-sensitive software-controlled caching framework eliminates cache tags. The same authors further improve their own results by showing that media applications are mapped more efficiently when scalar memory accesses are redirected to a mini-cache [223]. Using the Combined Static/Dynamic scheduler in the operating system as basis, [130] developed an Energy-Adaptive scheduler with an energy-aware scheduling algorithm that executes tasks to achieve effective use of limited energy by favoring low-energy and critical tasks.

Baynes et al. [12] evaluated energy consumption in various RTOS including preemptive systems and cooperative systems. Anther approach [1] proposed real-time dynamic voltage scaling that modify the operating system's real-time scheduler and task management service to provide significant energy savings while maintaining real-time deadline guarantees. Mishra et al. [148] proposes a probabilistic graphical model-based learning system to provide accurate online estimates of an application's power and performance in order to optimize energy efficiency of the system.

Table 2.3: Run-time energy efficient management techniques

Energy management	Techniques
Mapping	[12], [1], [221], [223], [137], [138], [130], [194], [7], [148]
Scheduling	[81], [56], [78], [186], [14]

2.7 Dynamic thermal-aware management methods

Delivering high performance in computation does not only come with the cost of power consumption. Often circuits that perform at their peak suffer from thermal issues such as overheating or faults due to thermal emergencies. Many dynamic management methods try to avoid such conditions. On the other hand, minimizing power does not necessarily avoid thermal issues. In many cases, concentrated power usage in a small part of the electronic circuit can cause high a temperature on that spot leading to thermal failures although the power usage of the whole system might not be high.

Thermal induced problems can appear in various forms. *Thermal hot spots* accelerate failure mechanisms. Failure cases increase exponentially with temperature [143]. Hot spots also cause performance loss and lead to higher leakage of power [224]. *Spatial variations* can cause clock skew resulting in transient or intermittent delay faults. Finally, *temporal variations* induce thermal cycling [36] that can cause violation in completion of the cycle that is large enough to cool the component.

Thermal management techniques try to control the chip temperature. Many of the power management methods in this chapter are concerned with temperature while focusing primarily on overall power consumption. The goal of DTM is to address thermal hotspots or reduce

spatial and temporal temperature variations. Frequency scaling, DVFS, Decode Throttling, Speculation control and cache toggling are some of the Dynamic Thermal Management (DTM) techniques described by [26]. Authors in [44] use temperature aware scheduling for multi-threaded processors by taking advantage of Simultaneous Multi-Threading (SMT) unique flexibility of having multiple threads to adaptively counteract and prevent hot spots by selectively managing the execution of available threads. Liao et al. [122] describes how smart performance and power modeling can reduce the power leakage and limit temperature increase which can eventually improve performance and power consumption. On the other hand, approach [92] proposes a framework in which thermal states are controlled by stochastic processes, i.e., partially observable semi-Markov decision processes. By using multi-objective design optimization methods such as collaborative optimization operating temperature is reduced.

Thermal management is an important issue in embedded systems due to limited area and cooling methods. Utilizing on-line monitoring, [31] proposes a response mechanism using a distributed power management algorithm for FPGA to evenly reduce and normalize power transients and achieve a power-and thermal-aware coherent system. Ayoub et al. [8] presented a DTM mechanism for memory subsystems which intelligently allocates workload pages to few memory units and powers down the rest of the memory.

Thermal management became a prominent challenge in fighting the expanding dark, nonactive, silicon areas on chip. Hajimiri et al. [66] proposed thermal-aware computation using proactive memory-based computing to reduce the peak temperature of applications. This technique proactively transfers the instructions with frequent operand pairs to memory. In [94] a dark silicon aware run-time mapping method was proposed that activates and deactivates cores as needed in order to evenly distribute power density across the chip.

Authors in [115] and [127] consider 3D stacking architectures and the thermal limitations for such chips. The method proposed by [115] combines dynamic cache management such as

resource allocation, way-based power gating, and data migration with dynamic voltage and frequency scaling of processing cores in a temperature- and energy-aware manner. Another approach [127] proposes a thermal-aware dynamic operating system page allocation using future access pattern to find a best performance-oriented setting of the above factors. Also, An analytic model has been proposed to estimate the system performance considering the memory interference, the bandwidth variation, and the throttling impact.

Thermal aware communication systems can reduce the possibility of thermal emergencies in the system [230]. Chou et al. [30] proposed a thermal aware method for dynamic buffer allocation for NoC systems. Zhang et al. [243] introduces a job allocation technique that minimizes the temperature gradients among the ring filters to improve the application performance in silicon-photonic NoCs. In the context of local temperature hotspots in a load-imbalanced network, approach in [156] demonstrated power and thermal profiles improvement by utilizing congestion-avoidance routing with network-level DVFS in a mm-wave small-world wireless NoC.

Finally, we can observe a trend in industry towards utilizing machine learning [93, 166] and neural network in power and energy management. This has been enabled by easy access to configuration knobs such as power gating, thread scheduling and frequency scaling. This ease of access to configuration knobs has been granted due to effectiveness of many of the approaches we discussed in this section. Therefore, the current trends will open the path for more complex management methods. Possible open challenges in power, energy or thermal management might be hierarchical management schemes, adaptive control or hybrid approaches comprised of machine learning, heuristics and control theoretic methods.

Table 2.4: Thermal management techniques

Thermal management	Techniques
Topology-aware mapping	[143], [8], [224], [44], [36], [122], [92], [243], [94], [30]
Resource allocation	[26], [203], [66], [31], [115], [127], [230], [156]

Chapter 3

Runtime Resource Management

Dynamic resource management has been established as an effective technique to improve reliability, efficiency, and performance of computer systems [177]. Managing shared resources during runtime becomes more complex with modern multicores which support diverse workloads that exhibit varying resource demands, sometimes with conflicting limitations. This dynamic behavior of workloads which vary across concurrent applications, creates significant challenges for homogeneous architectures. The need for a holistic dynamic resource management technique becomes more vital in emerging heterogeneous multicore processors (HMPs) where heterogeneous compute units are deployed on a single chip, allowing trade-offs between objectives such as maximizing performance and minimizing power consumption [180].

In this context, computer architects use several approaches to perform dynamic resource management. Model-based and Rule-based heuristic methods use a model or an encoded algorithm to make decisions during runtime. Optimization methods minimize/maximize an objective while considering certain constraints. Machine learning methods learn the best input values for different observed conditions. Finally, control theoretic techniques, use their intrinsic feedback loop to adapt to conditions.

In the following sections, some of the note worthy efforts for each of these resource management mechanisms will be discussed.

3.1 Heuristics and Optimization Methods

3.1.1 Power and Energy

Nowadays, computer systems design is confronted with delivering high performance while limited with their power consumption. The diversity in the type and increasing complexity of applications demands higher computation power. To deliver this performance, designers have to consider the reasonable autonomy in battery-powered systems, operational cost of cloud servers as well as reduction in the environmental impacts of power consumption.

Dynamic Power Management (DPM) and system design with the goal of energy efficiency has been studied in details in the past decades [15]. Designers used DPM in the 90s [32] with the available run-time configurations such as scaling the supply voltage to lower the power consumption [159]. Run-time monitoring of application behavior lead to improved optimization in power consumption [85]. The trend towards multi/many core platforms required techniques that can formally guarantee power management of the system given a power budget which is addressed in [24, 179]. On the other hand, in cases where power consumption is not predictable at design time adaptive approaches such as [194] can be used. For instance [80] uses an adaptive power management technique for hard real-time systems and [94] proposes a runtime mapping for many-core systems.

3.1.2 Temperature

Delivering high performance does not only come with the cost of power consumption. Often circuits that perform in their highest computation power suffer from thermal issues such as overheating or faults due to thermal emergencies. Many dynamic management methods have this in mind during runtime and try to avoid such conditions. Thermal induced problems can appear in various forms such as hot spots [143], spatial variations [2] and temporal variations [36]. The goal of dynamic thermal management (DTM) is to address thermal hotspots or reduce spatial and temporal temperature variations. Clock frequency scaling, DVFS, Decode Throttling, Speculation control and cache toggling are some of the DTM techniques described in [26]. Temperature aware scheduling for multi-threaded processors can reduce hot spots [44]. Smart performance and power modeling can reduce the power leakage and limit temperature increase which can improve performance and power consumption [122]. Thermal management became a prominent challenge in dark silicon era [8]. Furthermore, another approach [66] proposes a thermal-aware computation in nano scale technologies. Mechanisms in [115, 127] consider 3-D stacking architectures and the thermal limitations for such chips. Thermal aware communication systems can reduce the possibility of thermal emergencies in the system [230]. The mechanism in [30] proposes a thermal aware method for dynamic buffer allocation for network-on-chip based systems. In the same domain, authors in [62] propose a runtime workload mapping on network-on-chip based systems considering ripple effect of applications.

3.1.3 QoS

Quality-of-service (QoS) is a primary metric to qualitatively evaluate the system's efficiency in satisfying applications' requirements. Applications from different domains have different QoS metrics such as frame rate (multi-media) [67], latency-per-query (web search and

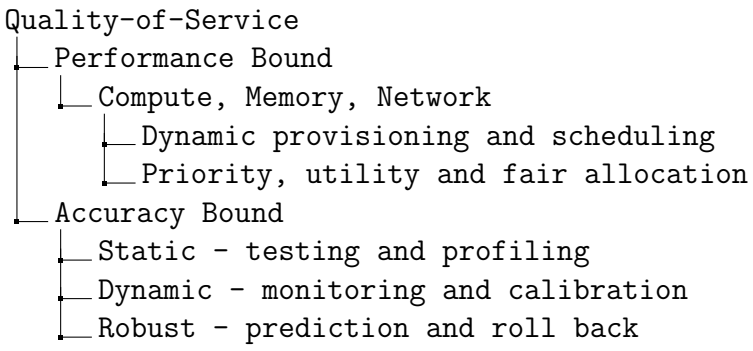


Figure 3.1: Abstract classification of QoS management.

financial) [126], throughput (data analytics and streaming) [39], responsiveness (user centric) [165], end-to-end latency and privacy (social media) [183], etc. Run-time QoS management becomes necessary and challenging with i) variable workload characteristics ii) variable QoS requirements of applications, iii) identification and translation of QoS metrics into system level parameters for provisioning and iv) resource contention and arbitration among concurrent applications. Meeting QoS requirements of applications are largely based on:

- the nature of computation - compute, memory and I/O intensity, streaming inputs and batch processing
- the nature of end result - numerical, perceptive, soft and hard real-time, and user-interaction

We abstractly classify QoS management techniques as *performance-bound* and/or *accuracy-bound*, as shown in Figure 3.1. We present major underlying approaches and strategies for performance bound QoS guarantees through provisioning compute, memory and network bandwidth resources and accuracy-bound QoS through quality monitoring and control.

Performance Bound QoS

Performance bound QoS can be guaranteed with compute, memory, network and I/O bandwidth provisioning with dynamic priority identification.

Compute: Allocating more and/or suitable cores, CPU time slices, exploiting core-level asymmetry to fit application's QoS requirements are common approaches for QoS guarantees [168] [126] [39]. Under workload diversity, smart co-location - scheduling an optimized combination of latency and throughput sensitive applications together, exploits under-utilized resources to satisfy QoS of both types of appplications [140] [236] [168] [39] . All these techniques feature user/application defined QoS metrics such as latency and throughput bounds or dynamic identification of critical resource contending regions of code [215] and measure QoS in terms of IPC and harmonic speed up for scheduling decisions. Monitoring QoS based on IPC and satifying applications requirements through optimized time slice sharing among concurrent applications is proposed in [43]. Combining a set of cores, memory and network bandwidth into a package to provision isolated resources for applications as per their QoS requirement is proposed in [246], to provide infrastructure as a service. All the provisioning techniques prioritize applications based on QoS requirements and dynamically adapt further by monitoring resource utilization upon provisioning.

Memory and Storage: With the widening compute-memory performance gap, allocating larger cache slices and higher memory bandwidth significantly enhances performance bound QoS metrics [214] [216] [211]. Another approach [219] introduces a computational storage device that allows big-data applications to be processed in the storage systems instead of moving huge amounts of data from storage units to the processing units. This drastically improves the performance of big-data applications. Using cache partitioning to provide either larger/sufficient cache slices is a common approach to meet QoS requirements of latency critical applications [99] [87]. Identifying application/thread priority and scaling cache allocation accordingly, following utilitarian principles is another strategy to improve overall

throughput metrics [73] [200] [214]. Optimizing for memory controller proximity [13] [211] and allocating higher bandwidth can enhance QoS of memory intensive applications [239] [244]. All the dynamic memory provisioning techniques however require micro-architectural/OS level extensions to identify and translate between user/application defined QoS performance metrics to system level QoS utilization metrics [118] [119] [73].

Network and I/O: Allocating higher network and I/O bandwidth to prioritized applications can guarantee latency and throughput QoS requirements. Existing techniques have used customized router architecture, virtual channels, flow control and frame scheduling to provide higher network bandwidth for dynamically identified priority applications. Classification of network into shared resource and non-shared resource clusters to allocate non-QoS and QoS tasks respectively through novel router architecture was proposed in [58]. Assigning each flow into frames and intelligent scheduling globally synchronized frames to optimize for latency is proposed in [114]. The same idea is extended by [164] with a flexible local frame scheduling and pre-emptive flit reservation for more bandwidth for high priority applications. Distinguishing between latency and throughput sensitivity of best effort (BE) and guaranteed throughput (GT) to optimize their respective flow control is proposed by [41] [42]. While BE applications are prioritized by default, priority is inverted to GT when BE applications have used enough buffer space reflecting in a certain throughput guarantee. Assigning a fixed bandwidth to each flow and monitoring its bandwidth utilization to prioritize and allocate network resources to utility frames is proposed in [59]. A similar approach with hybrid fair and elite round robin bandwidth allocation using weighed priorities is proposed in [70]. Each of these techniques dynamically determine priority of packets (originating from priority applications) and route them first, while other low priority packets wait in the queue.

Accuracy Bound QoS

Approximate computing leverages inherent error resilience of applications from domains such as machine learning, multi-media processing, streaming, data mining and analytics due to algorithmic nature, redundant input data, and perceptive end results for performance and energy gains [145]. However, reasoning for accuracy loss, guaranteed error bound and control on quality of result (QoR) is crucial for viability of approximation techniques. Existing approximation techniques use profiling, calibration and light-weight checks for nominal quality control [112]. We divide quality assurance techniques into three categories: viz., *static* - profiling, *dynamic* - calibration and *robust* - control and roll back.

Static Techniques: Profiling techniques validate results of an approximated code block over an exhaustive set of inputs against the accurate result to derive empirical guarantees on error [190] [202] [10]. Static techniques are as effective as the input data coverage i.e., error bounds can be guaranteed for input sets that are pre-evaluated at profiling phase, which can in turn be used at run-time for quality control.

Dynamic Techniques: Dynamic quality control techniques use calibration - executing each candidate block of code over both accurate and approximate methods to determine nature and extent of error induced at run-time [52] [202]. These approaches then either rely on user-defined or application level accuracy requirement targets to determine whether approximate execution is within an acceptable quality range. Some techniques use the target accuracy requirement as a feedback to explore accuracy-performance Pareto space to configure the extent of approximation [202] [10]. Dynamic techniques are efficient in providing empirical and/or statistical guarantees on quality, however they require additional hardware/software overhead for continuous monitoring and execution of both accurate and approximate versions. Reducing sampling rate of monitoring might ignore errors induced during the un-sampled interval.

Robust Techniques: Robust quality control techniques monitor accuracy loss at run-time

and can roll back for more accurate execution in case of errors induced beyond acceptable thresholds. Robust techniques address the limitations of static techniques which can provide guarantees only over tested inputs, and dynamic techniques which have overheads and lesser coverage within sampled invocation. Robust techniques use predictive, online learning, light-weight checks and monitoring strategies to compute quality loss and predict the extent of quality loss for subsequent inputs [112] [226]. The quality loss is compared against user defined accuracy requirements to either tone down aggressive approximation or choose a different type of approximation technique [234] [57]. In case of unacceptable results, these approaches roll back i.e., re-execute the candidate code blocks in accurate mode to cover for the accuracy loss. Robust techniques include re-configuring the extent of approximation [226] [57], re-generation of type of approximation used iteratively [235] [152] and pro-actively [212] [112] and roll back by re-executing the code block accurately [101] [135].

3.1.4 Reliability

In the last decade, reliability has become a major issue in digital circuits [193]. The aggressive scaling to nanoscale CMOS structures has caused a variety of reliability threats such as aging and wear-out acceleration due to the increased power densities and consequent thermal stress, higher susceptibility to soft errors not only in harsh environments but also at ground level, device variability leading to timing errors and other effects, etc. This issue has been even more exacerbated by the pervasiveness of computing systems in nowadays life spanning from smart environments to datafarms devoted to control and support of decision processes.

Device aging and wear-out are some of the predominant reliability issues since they cause a sensible shortening of the lifetime [90] (lifetime variation may be also quantified in 2x [98]). Unfortunately, DTM does not suffice since aging control cannot be performed only by limiting hotspots and temperature variations. Instead, as discussed in the literature [209, 207], it can

only be fully achieved by monitoring the "cumulative" degradation behavior of the aging phenomena and accordingly managing resources in an aging-aware way for executing the workload. Nevertheless, this strategy is particularly effective for multi-core or many-core platforms (both homogeneous and heterogeneous ones) thanks to the availability of a large set of "programmable" processing resources, representing a sort of redundancy, that can be dynamically tuned and selected for the execution of the various applications composing the workload, that is dynamically changing as well.

The first Dynamic Reliability Management (DRM) approach [209] focusing on a single general purpose processor was proposed in 2004. After that, following also the architectural progresses in the subsequent years, different types of platforms have been considered spanning from the classical homogeneous multi-core architecture [35][129][63], where processing units are connected on a single bus and with a shared memory, to the NoC-based many-core architecture [131] [213] [103]. Recently, heterogeneous architectures [11][27][113], integrating asymmetric processors, GPUs or custom accelerators, have been also addressed in lifetime management. Depending on the specific architecture, the resource management approaches act on application mapping (as in the case of many-cores architectures [213][64]), scheduling (as in the case of shared-memory systems [35]), and/or on power-related knobs (DVFS and per-core power gating [209][35] [129]). Another relevant aspect is that lifetime is only one of considered parameters, thus leading in most of the approaches to a co-optimization with performance or power/energy-consumption. In [185, 184] authors propose a hardware sharing framework for allowing host applications to fully share FPGA-based accelerators in a congestion-free environment.

Another relevant reliability issue in modern technologies is the high susceptibility to soft errors of the devices. Such transient faults use to occur with a given Soft Error Rate (SER) that is dependent also on the operating voltage/frequency levels and may be subject to variability among the various cores of the same device [96]. Therefore, runtime resource

management policies [96][233][132] have been proposed to optimally distribute the workload and tuning architectural knobs to maximize the system reliability, measured as the probability to complete successfully current computations, together with other metrics (performance, power consumption and lifetime).

Finally, it worth mentioning a last class of runtime resource management approaches that prefer to integrate also fault handling in the controller at software level rather than using classical architectural mechanisms. To give few examples, some approaches (e.g. [25]) dynamically replicate application execution to perform fault detection or mitigation w.r.t. both transient and permanent faults, while other strategies (e.g. [65, 204]) schedule at runtime software-based self testing routines to identify permanent damages.

3.2 Machine Learning Approaches

The popularity of machine learning methods has grown in the past decade. The learning nature of these methods make them a good candidate for model prediction and resource allocation in computer system. It should be noted that these methods mostly require a learning phase with a large amount of measurement data to prepare the predicting algorithm for a specific platform. Using the advance machine learning, reinforcement learning and deep neural network methods can bring high accuracy in prediction and tuning of the architectural parameters in case that the system conditions stay the same to the conditions captured in training phase. The advantage of using these methods lie in the data driven identification of relationships that can be used for tuning system configurations at runtime.

We can categorize the machine learning methods used in resource management of computer systems based on their ability to adapt to changes over time. Data intensive and time consuming training phase of these methods often requires a detailed analysis and tuning at

design time. In many cases the platform and the workloads are predetermined which gives the designers the opportunity to train the model once at design time and during execution use the **offline** model for prediction and tuning. On the other hand, **online** methods are needed to learn new changes in workload behavior or environment changes and incorporate that in the machine learning methods. These methods mostly start with a default model trained at design time and improve them at runtime to increase the prediction accuracy and management efficiency.

3.2.1 Offline learning methods

Machine Learning approaches [61, 22, 49, 40, 83] for runtime resource management have gained lots of traction in the past few years, especially in management of high performance systems and cloud servers. Specifically, machine learning techniques have been a promising trend for modeling the complexity of interaction among different on-chip resources and the corresponding effect on resource metrics [60]. Further, these techniques have targeted beyond the conventional fixed single and multi- objective allocation policies, towards dynamically varying goals [199, 91, 198].

Figure 3.2 depicts a general resource management mechanism similar to the method proposed by [23]. This method leverages Artificial Neural Networks (ANN) to manage multiple shared CMP resources in a coordinated fashion to achieve a high-level objective. It's important to note that although the major part of the training phase for machine learning approaches is done at design time, the decision making and often backward learning for adaptation can be done at runtime. Authors in [48] propose using machine learning for microarchitectural adaptivity control. Approaches such as reinforcement learning have been used to design self-optimizing memory controllers [83]. Similarly, [245] proposes a dynamic resource management using deep reinforcement learning.

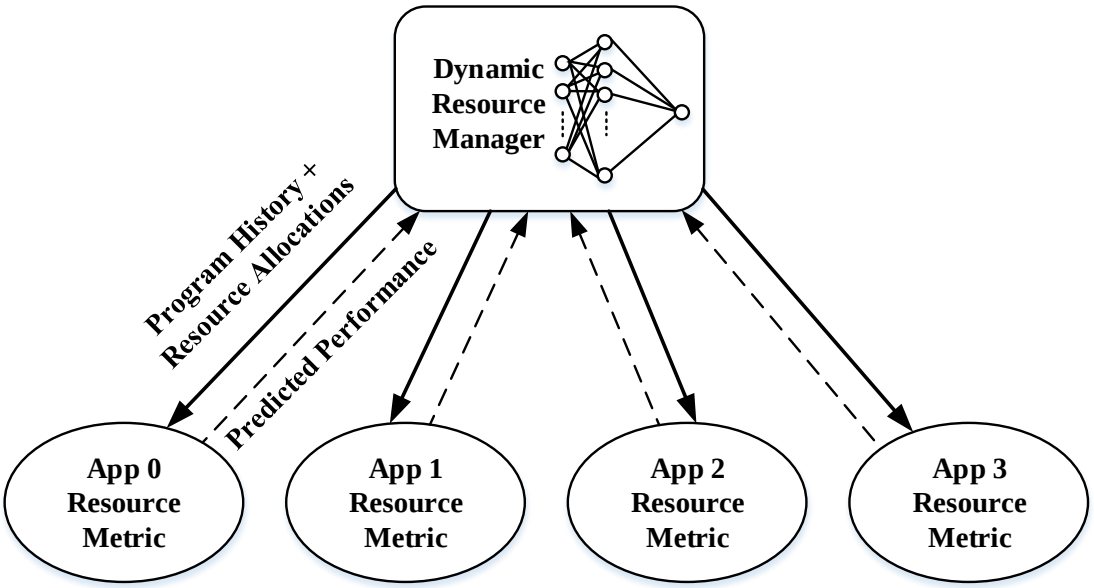


Figure 3.2: High-level view of resource management technique using machine learning similar to method presented in [23].

3.2.2 Online learning methods

Conventional machine learning methods require extensive training to learn the correlation between inputs and outputs of the system. In the case of a new situation at runtime which they were not trained for, they might provide an inaccurate solution. To address this issue, online learning methods [37, 102, 100, 79] show promising results in learning new scenarios at runtime compared to a complete, expensive re-training of the weights and parameters.

These methods need special tailoring before deployment on embedded and real-time systems in order to reduce their high computational overhead at runtime [82, 160]. In future, on-chip dynamic resource management can benefit from data driven identification and high accuracy of prediction in machine learning methods used along side lightweight heuristics or robust control theoretic methods [147].

3.3 Control Theory Techniques

Dynamic resource management for many-core systems is increasingly challenging due to the complex interactions in these systems. Integration of hundreds of cores and uncore components running various workloads with conflicting constrains increase the pressure on limited shared system resources. A promising and well-established approach is the use of control-theoretic solutions based on rigorous mathematical formalisms that can provide bounds and guarantees for system resource management [75, 227] In this context, we discuss some of the efforts that deploy control-theoretic centric run-time management. Starting from simple Single Input Single Output (SISO) controllers used in power management to more complex Supervisory Control Theory (SCT) methods used in dynamic resource management of complex computer systems.

3.3.1 Single-Input Single-Output controllers

Modern many-core platforms provide high performance but are increasingly constrained by power dissipation. In addition, applications typically exhibit dynamic characteristics (e.g., memory-bound, compute-bound) throughout their execution, resulting in continual changes in the power state of the system. It is essential to control the peak and average power based on application behaviour in order to achieve the proper performance with minimum cost [177]. This requires thorough analysis and sophisticated power management methods to control power and provide necessary performance for a diverse set of workloads. Some approaches [75, 134] use analytical models to estimate the average or worst case power consumption of the system based on frequency and voltage level of the system. These methods fail to take into account the effects of workload and input variability during system execution. A promising and well-established approach is the use of control-theoretic solutions based on rigorous mathematical formalisms that can provide bounds and guarantees for system

power consumption. In the past, different control methods have been proposed [232, 173] for resource management in the presence of a specific type of workload running on the system. A majority of these methods use Single-Input Single-Output (SISO) controllers. These SISO controllers often deploy proportional Integral (PI), proportional integral derivative (PID), or lead-lag methods. Although these controllers theoretically provide guarantees for stability and robustness, significant care must be taken in their practical implementation to ensure that these properties continue to hold in the implemented designs. For instance, SISO controller can be implemented at the various layers of the abstraction stack (e.g., application, OS, hypervisor or hardware), resulting in different challenges and design tradeoffs: software controllers provide ease of implementation and flexibility, while, hardware controllers provide higher responsiveness to sensor measurements. In many cases, the controller configuration needs to be changed to manage power for a new set of applications. Software-based controllers provide such flexibility but are limited on response time to changes in the system, currently in the order of milliseconds. This could pose problems when an application's phase can change faster than the settling time of the controller. In addition, some applications cannot be controlled using classic static controllers and require more advanced solutions. These are examples of many issues that demand a thorough analysis of application behaviour early enough (e.g., at the time of system identification) and well before controller deployment.

In our study of SISO controllers, we design and deploy PI controllers for power management. It is important to note that although derivative control law is helpful to add predictability to the controller, stochastic variations in the system output may cause inaccuracy in the controller. This issue becomes more severe in computer systems as they commonly have a significant stochastic component. Therefore, for computer systems PI controllers are preferred over PID controller [71]. PI control benefits from both integral control (zero steady-state error) and proportional control (fast transient response). In most computer systems a first-order PI controller provides rapid response and is sufficiently accurate [71]. Figure 3.3 depicts a first-order feedback PI controller modeled in Z-domain. The error $E(z) = R - Y(z)$ is the

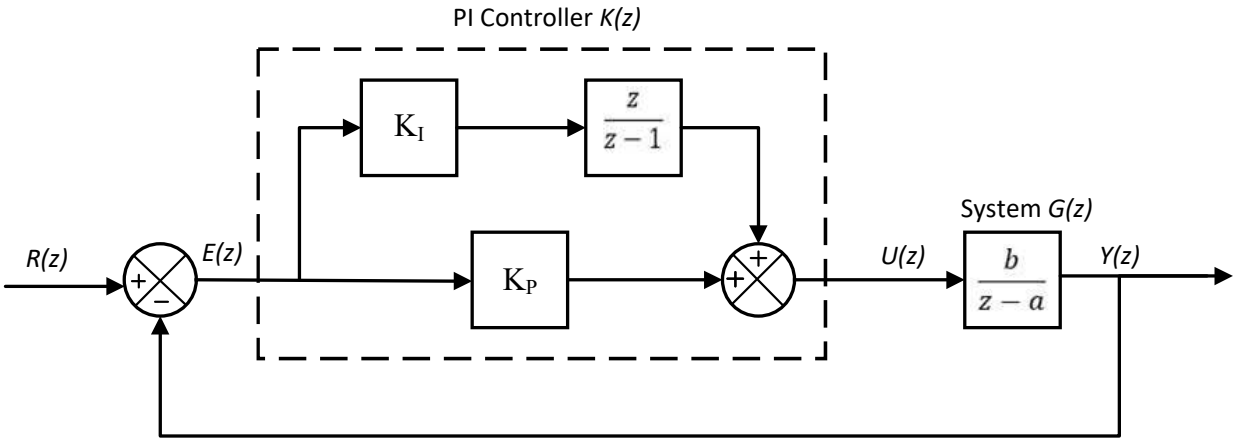


Figure 3.3: Feedback loop with PI control for a first-order system

input to the controller. The control input *U(z)* is a sum of the proportional term $K_P \times E(z)$ and the integral term $K_I \times (z/(z-1)) \times E(z)$.

Equation 3.1 describes a simple discrete PI control form that can later be transformed to transfer function. Note that to compute the current control input *u(k)*, the controller needs to have the current value of the error *e(k)* along with the past value of the error *e(k-1)* and the past value of the control input *u(k-1)*. It is this memory inherent in the PI controller that makes it dynamic (in contrast to the static PI controller). The PI control law has the form:

$$u(k) = u(k-1) + (K_P + K_I)e(k) - K_P e(k-1) \tag{3.1}$$

It is important to note that a power management controller designed for only a specific class of applications might not perform well in managing power for other types of workloads. The merit of a controller is measured in terms of four properties: Accuracy, Overhead, Robustness and Flexibility. Thus, a designer's major concern is to evaluate how well a controller satisfies these properties while executing different types of workloads (e.g., compute-bound or memory-bound). The dependability evaluation presented in this work offers designers a better insight on how to properly model (i.e., identify) their system and what kind of considerations they

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.

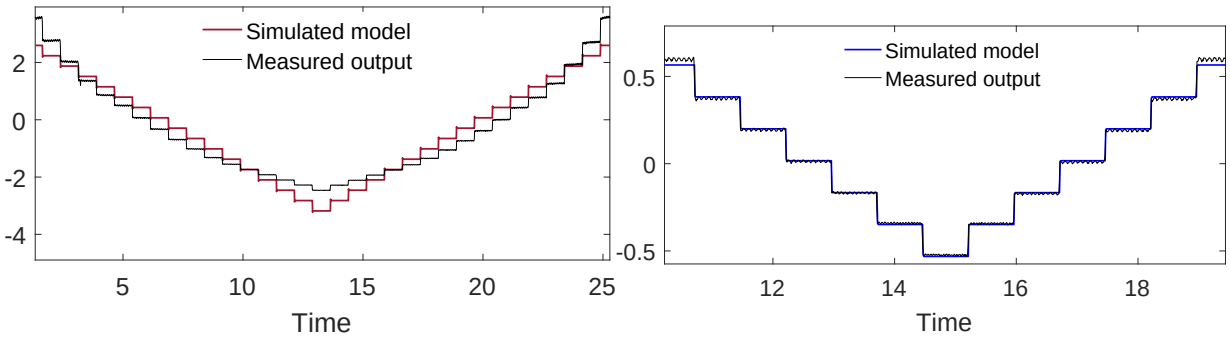
need to take into account when designing controllers for processors.

3.3.2 GSC

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 control law 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.

In this section we describe a novel nonlinear DVFS power management approach using a well-established and lightweight adaptive control theoretic technique called Gain Scheduling. We describe the methodology for integrating multiple linear models within a single controller implementation in order to estimate nonlinear behavior of DVFS for CMPs.

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. 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 [125, 124] for isolating the deterministic and stochastic components of the system to build the model.



(a) Power for full frequency range.

(b) Power for 200–800 MHz.

Figure 3.4: Modeled and observed behavior of nonlinear full-range system (a) vs. linear operating region (b).

Figure 3.4a 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 effect on dynamic power ($P = CV^2f$). 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 [47]. Figure 3.4b 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.

3.3.3 MIMO

Control theoretic approaches for resource management (e.g., [227, 75, 64, 95]) provide formal guarantees for achieving robustness and stability, particularly in the presence of workload variability.

Multiple-Input-Multiple-Output (MIMO) controllers have proven to be effective for coordinating management of multiple goals in unicore processors [172] and HMPs [157]. Consider the MIMO controller in Figure 3.5 that controls a system with two control inputs and two

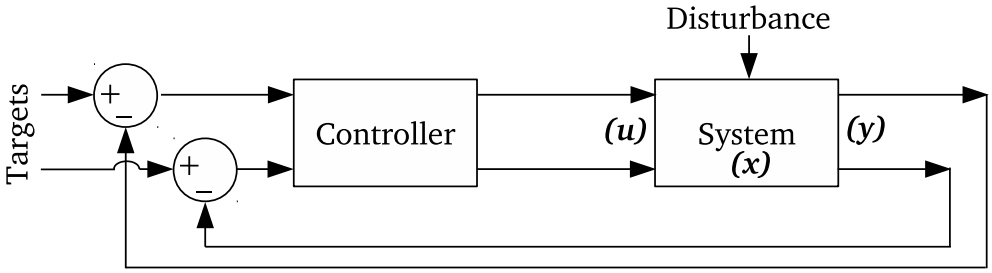


Figure 3.5: Basic 2×2 MIMO.

interdependent measured outputs. Picking actuators and measurement metrics that result in behavior that can be estimated linearly is an important aspect of designing a stable controller [97].

MIMO control for coordinated resource management [172, 171] has generalized management of multiple controllers or objectives for a *single-core* processor. Consider the MIMO controller in Figure 3.5 that controls a system with two control inputs and two interdependent measured outputs. The MIMO is implemented using a Linear Quadratic Gaussian (LQG) controller [205]:

$$x(t+1) = A \times x(t) + B \times u(t) \tag{3.2}$$

$$y(t) = C \times x(t) + D \times u(t) \tag{3.3}$$

where x,y, and u are vectors representing the system state, the measured outputs, and the control inputs, respectively. Coefficient matrices A, B, C, and D capture the system behavior, and their values are obtained through *system identification*. Matrix sizes are determined by both the number of inputs and outputs of the controller as well as the *order* of the controller.

The MIMO design process consists of: 1) defining the system to be controlled by specifying inputs and outputs; 2) using experimental data to identify the system model; 3) designing

and tuning the controller based on the system model; and 4) analyzing and validating the robustness and stability of the designed controller.

In this thesis we focus mostly on steps (1) and (2). Once the controlled system is defined, the first step in system identification is generating test waveforms from training applications in order to create a system model. For complex systems it is more common and feasible to use statistical or black-box methods based on System Identification Theory [125] for isolating the deterministic and stochastic components of the system to build the model. Given an order, the model estimation generates the A, B, C, and D matrices (Equation 3.2, 3.3). The order dictates the dimension of the model (i.e., size of the state space), which is typically a trade-off between accuracy and complexity. Once the model is created, it is cross-validated using a different data set and the *model uncertainty* is assessed using *Robust Stability Analysis*[125]. The higher the uncertainty guardband, the more robust is the model and therefore the generated controller.

Picking actuators and measurement metrics that result in behavior that can be estimated linearly is one of the most important aspects of designing a stable controller [97]. Reducing model uncertainty is crucial for the stability of a controller: *perturbations due to model uncertainty can destabilize a system; if system identification is completed successfully, the remaining steps in controller design are trivial.*

3.3.4 Fuzzy

A fuzzy controller can be viewed as an artificial decision maker that operates in a closed-loop system in real time. Figure 3.6 depicts a simple fuzzy controller in the context of feedback control. As shown in this figure, system output is sensed and represented by y which is compared against the references set by the designer or the user r. This difference is fed to the fuzzy controller to decide system input(s) u (i.e., control output(s)) to guide the system

towards the desired goals. The fuzzy controller consists of four main components: (1) the fuzzification component that interprets the inputs to be matched with the rules, (2) the rule base which is a set of rules that defines the knowledge on how to control the system in different situations, (3) the inference mechanism which matches the rules with the current situation and determines the fuzzy set for control outputs, and (4) the defuzzification component which converts the output of the inference to actual actuator values [1].

Before describing each component in detail, we cover the terminology used in fuzzy control. To specify rules in the rule-base, an expert uses a *linguistic description*. These descriptions are usually in the form of *condition* $\rightarrow$ *action*. In this description, *linguistic variables* are used to describe fuzzy system inputs and outputs, and exist in one-to-one correspondence with numeric variables. For example, QoS-error is a linguistic variable corresponding to the numeric variable for the change in FPS. A linguistic variable takes on linguistic values such as positive-large and negative-large. Such variables indicate, among other information, the direction and magnitude of a variable. To better elaborate on the design components of a fuzzy controller, we use a simple DVFS example for single core power management. This is a simple and classic control problem for which many efficient techniques already exist such as a simple PID controller or a regression model. However, we use this example simply to illustrate the design and basic mechanics of a fuzzy control system. Here, y denotes the power consumption of the core (in Watts), and u is the frequency of the core (in MHz). We will use r to denote the desired power of the processor. The goal is to track this target power reference either specified by the system design manual or imposed by a higher system objective.

[1] We limit our description of fuzzy control to cover the basis of practical control applications. We encourage the avid reader to find a more detailed analysis of fuzzy logic, sets, and systems to consult [106][105][247].

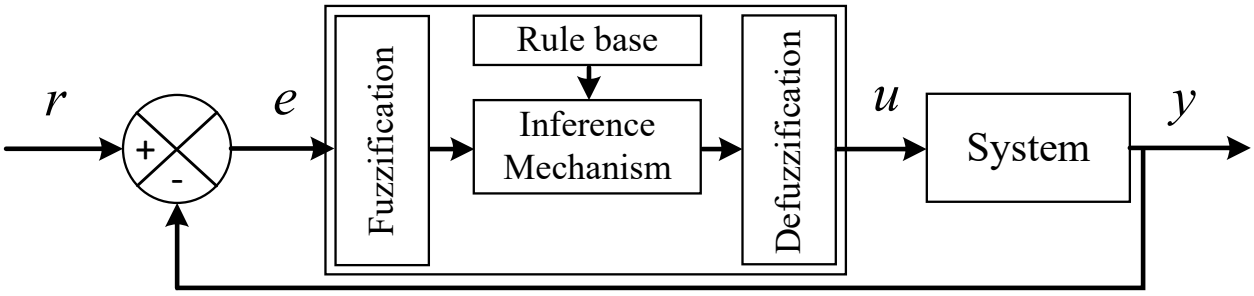


Figure 3.6: Overview of Fuzzy control

Fuzzification

The role of the fuzzification interface is to convert controller inputs into information that can be easily used to process, activate, and apply rules. Fuzzification can be simply defined as the mapping process between an obtained value for an input variable (e.g., IPS value, QoS metrics, execution time) to its numeric value defined in the corresponding membership function (MF). Membership function values can be interpreted as the *encoding* of the fuzzy controller numeric input values. The encoded information is then used in the fuzzy inference process. Depending on the application and the designer's preference, many different choices of membership functions are possible. Membership functions (e.g., trapezoid-shaped, Gaussian-shaped, Sharp peak Skewed triangle, etc.) quantify the meaning of the linguistic statements that experts used in defining the rules in the rule base [167].

In our DVFS example, we use $e(t) = r(t) - y(t)$ as the input to the fuzzy controller which denotes the error between the reference power and the current power value. As $e(t)$ takes on a value of, for example, 100mW at t = 2 (e(2) = 0.1W), linguistic variables assume "linguistic values." That is, the values that linguistic variables take on over time change dynamically. Suppose for the DVFS example that *error* and *change-in-frequency* take on the following values (*negativeLarge*, *negativeSmall*, *zero/hold*, *positiveSmall*, and *positivelarge*). Top part of Figure 3.7 shows membership function for *change-in-frequency* with corresponding values as a sample. This membership function can be used in the next steps to process and determine

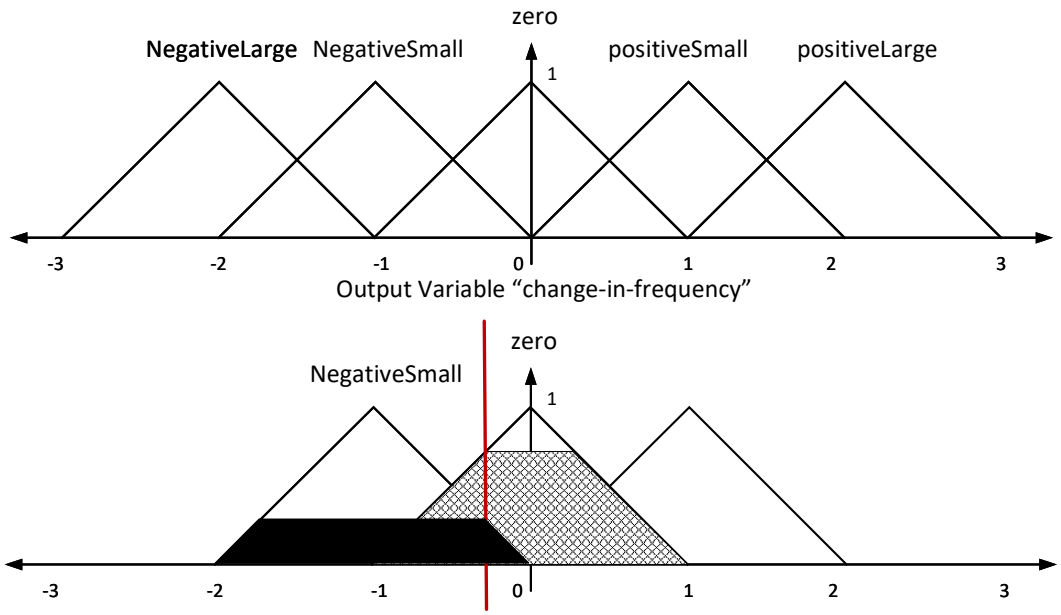


Figure 3.7: (top) Sample membership function for *change-in-frequency*. (bottom) Implied fuzzy sets for two rules in DVFS example

the output of the fuzzy control or simply decide how much frequency needs to be changed to achieve the target reference. We define similar functions for error in power with an adjusted range. Note that we mainly use triangles and skewed triangles for membership functions in this work due to its low computation overhead.

Rule base

Construction of the rule base is where the experience and domain knowledge of experts prove to be beneficial. A deep understanding of the target system dynamics can increase the success of the designed controllers in the deployment process. In this step, values for input and output variable are described. Expert's knowledge alongside the above quantification is then used to specify a set of rules on how to control the system. Fuzzy rules are expressed in terms of linguistic variables. It is important to note that these rules are defined in a way that is easy to understand and interpret by humans. At this point, designer does not need to

focus on details of control parameters and can simply define the rule structure of the control process. This comes from the raised abstraction level in the rule base definition that simply specifies the general idea on how to control a process. Although this might appear to be different from designers utilizing heuristics, this difference is one of the strong points of fuzzy control as these rules are defined in a clear and understandable way which can be subject to test and improvement.

For instance, in our DVFS example if the power consumption is just slightly higher than our target, we want to reduce the frequency a bit to reduce the core power. This can simply be added as a rule that says "if power error is a small positive value then change the frequency by a negative and small value" (equivalent to rule 4 in pseudocode 1). In the field of fuzzy control, there has been a vast body of work on how to automatically tune fine parameters of controller after the initial rule base structure has been defined by the expert designer [88, 45, 237]. In addition, fuzzy control provides multiple analytical methodologies (e.g., The Lyapunov Method, Absolute Stability, and the Circle Criterion [4]) for stability analysis that can analyze the deployed heuristics. Pseudocode 1 shows a sample rule-base for the DVFS example. The intuition behind the rules is simple, assuming the frequency variations change power consumption, based on the *error*, the fuzzy inference decide on the number of steps in the change of frequency. To summarize, the rule base keeps a record of linguistic variables, values, and their associated member functions in addition to the set of all the rules. These rules have the general format of conditional statements making them easy to understand and computationally lightweight, for example, when compared to the matrix algebra used in state-space based MIMO control.

Fuzzy Inference

In the inference mechanism component, the expert's *decision making* is emulated by interpreting and applying knowledge about how best to control the plant. This mechanism is also

Algorithm 1 DVFS rule-base example

Input: error: difference between the current power and the target power
Outputs: change-in-frequency: actuation to the next frequency
(1) if error is *negativeLarge* then change-in-frequency is *positiveLarge*
(2) if error is *negativeSmall* then change-in-frequency is *positiveSmall*
(3) if error is *zero* then change-in-frequency is *zero*
(4) if error is *positiveSmall* then change-in-frequency is *negativeSmall*
(5) if error is *positiveLarge* then change-in-frequency is *negativeLarge*

often called fuzzy inference or inference engine. The inference comprises of two steps. In the first step, the current situation is determined based on the comparison of the premises of all the rules and control inputs. Note that, in this *matching* process, more than one rule can be applied to a situation. Based on the membership functions and the control inputs, we determine the certainty that each rule applies. This simply means the rules that are more relevant to the current status of the system will have a stronger influence on the inference conclusion. This certainty is denoted by $\mu_{premise}$ of that rule. To perform inference, each of the applied rules must first be quantified by extracting the value of each fuzzy controller input terms present in that rule and then applying the fuzzy logic (and/or) operation on them. Usually minimum or product operations can be used here which will lead to a fact for the rules that include multiple input statements, we can be no more certain about the conjunction of two or more statements than we are about the individual terms that make them up. In our example we only have one input stat, if power consumption is not close to the target reference, the matching decides the certainty of rules such as the ones that starts with *negativeLarge* and *negativeSmall* values for *error* in power. If we get a small negative value between zero and one for the error rules 2 and 3 will be picked in the matching process where rule 2 states " error is *negativeSmall* then change-in-frequency is *positiveSmall*" and rule 3 is defined as "if error is *zero* then change-in-frequency is *zero*".

The second step involves determining the controller actions or the *conclusion* process. Every rule that can be applied to the current situation in the control system has a corresponding

action which defines a controller action or a conclusion. Based on the number of active rules in each situation, there can be one or more conclusions with different levels of certainty.

The conclusions are characterized by a fuzzy set that represents the certainty that the control inputs had in the matching process. Next, we consider each conclusion separately to determine what is the action recommended by the associated rule. Bottom part of Figure 3.7 shows an example that the implied fuzzy sets of the inference that matched with rules (2) and (3). We can see that certainty of the rule (3) ($\mu_{premise} = 0.75$) is higher compared to the second rule ($\mu_{premise} = 0.25$) which means that conclusion of this rule will have a stronger influence on the inference conclusion. Based on this we define the conclusion of each rule as:

$$\mu_{(2)}(u) = min\{0.25, \mu_{negsmall}(u)\} \tag{3.4}$$

$$\mu_{(3)}(u) = min\{0.75, \mu_{zero}(u)\} \tag{3.5}$$

In the next step, every recommendation from all the rules are combined to calculate the final controller action value.

Defuzzification

Defuzzification component operates on the implied fuzzy sets produced by the inference mechanism and combines their effects to provide the "most certain" controller output [167]. Basically, Defuzzification is the process of converting the degrees of membership of output linguistic variables within their linguistic terms into crisp numerical values. There are various defuzzification methods that can be used to find these numeric values such as Center of

Gravity (COG), Center of Area (CoA), Modified Center of Area (mCoA), Center of Maximum (CoM), Mean of Maximum (MoM), Center of Sums (CoS) [210]. If one considers fuzzification as an "encoding" process, defuzzification can be seen as a "decoding" mechanism for the fuzzy set(s) obtained from the inference engine to generate numeric values.

Finally, going back to our example, we decode the result of the inference step from something such as a fuzzy set of *negativeSmall* and *zero* change-in-frequency (Equations 3.4, 3.5). We use COG defuzzification mechanisms to find the crisp output for change-in-frequency (e.g., decrease in current frequency by 100 MHz if possible). Note that we can have another input to the fuzzy controller that checks the current frequency and makes sure that change-in-frequency will not lead to an out of range frequency value. We can also do this as part of post processing or a filter after the controller. Adding gains before and after the controller is also a common practice to tune the effect of the control decision on the system.

3.3.5 SCT

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) [181] 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.

Scalability via Supervisory Control

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.

Figure 3.8 illustrates how a supervisory control structure can hierarchically manage control loops. As shown in the figure, supervision is vertically decomposed into tasks performed at different levels of abstraction [217]. The supervisory controller is designed to control the high-level *plant model* P_{hi}, which represents an abstraction of the system. The *plant* is the pre-existing system that does *not* (without the aid of a controller or a supervisor) meet the given specifications. Information channel Inf_{hi} provides information about the updates in the high-level model to the supervisory controller, and the supervisory controller uses the Con_{hi} channel to control this model. However, due to the fact that P_{hi} is an abstract model, the controlling channel Con_{hi} is only a **virtual** channel. In other words, the control decisions of the supervisory controller will be implemented by controlling the low-level controller(s) C_{lo} through commands transmitted via the communication channel Com_{hi_lo}. Consequently, the low-level controller(s) C_{lo} can control one or multiple subsystems using the Con_{lo} channel and gather information via the observation channel Inf_{lo}. The changes in the low-level plant P_{lo} can trigger updates in the state of the high-level model P_{hi} through the information channel

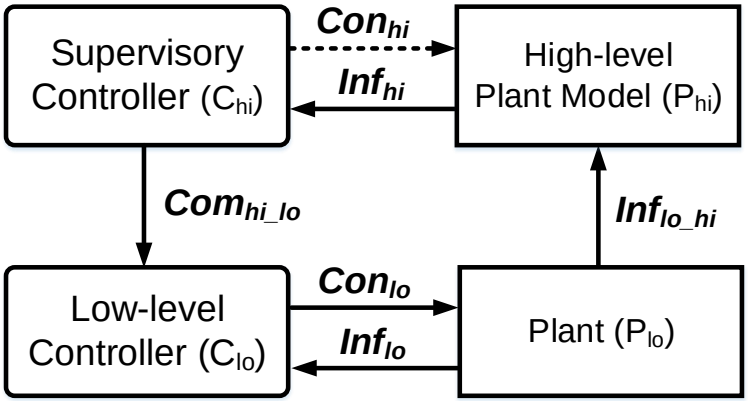


Figure 3.8: Scalability via Supervisory Control Structure.

Inf_{lo_hi}. These updates would reflect the results of low-level controller C_{lo}'s controlling actions. The scheme of Figure 3.8 describes the division of supervision into *high-level management* and *low-level operational supervision*. Virtual control exercised via the Con_{hi} high-level control channel can be implemented via Com_{hi_lo} to adaptively coordinate the low-level controllers, for example by adjusting their operating modes according to the system goal. The important requirement of this hierarchical control scheme is *control consistency* and *hierarchical consistency* between the high-level model and the low-level plant, as defined in the standard Ramadge-Wonham control mechanism [217]. For a detailed description of SCT, we refer the reader to [181, 188, 16, 217].

Autonomy via Supervisory Control

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 control has two key properties: i) rapid adaptation in response to abrupt changes in management policy [74], and ii) low computational complexity by computing control parameters for different policies **offline**. New policies and their corresponding parameters can be added to the supervisor on demand (e.g., by upgrading the firmware or OS), rendering online learning-based self-tuning methods, e.g., least-squares estimation [6], unnecessary.

Figure 3.9 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 [117]. 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 3.2 and 3.3. 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,

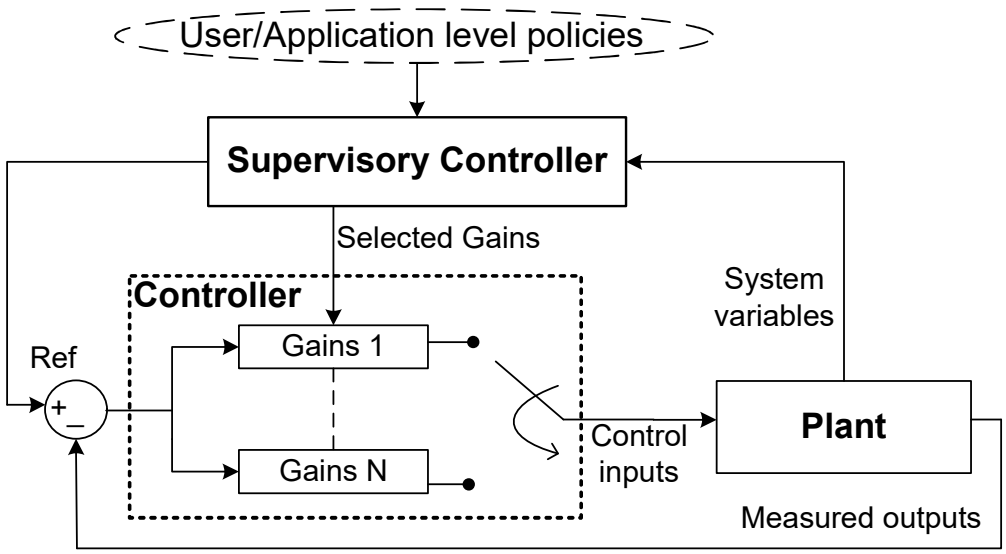


Figure 3.9: Autonomy via gain scheduling in SCT.

this is done by changing priorities of outputs using the Q and R matrices. 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).

3.4 Discussion

To summarize the coverage of existing on-chip resource management methods studied in this work we use Table 4.2. Some heuristic approaches (Row A) focus on **efficiency (3)** and **coordination (4)**, but fail to provide formal guarantees and autonomy to the system. On the

Table 3.2: Major on-chip resource management approaches (∗ = partially addressed)

	Methods	1. Robustness	2. Formalism	3. Efficiency	4. Coordination	5. Autonomy	6. Scalability
A	Heuristic methods			✓	✓		
B.1	Offline Machine Learning methods		✓	✓	✓		⋆
B.2	Online Machine Learning methods		✓	✓	✓	⋆	⋆
C.1	SISO Control Theory	✓	✓	✓			
C.2	MIMO Control Theory	✓	✓	✓	✓		
C.3	Fuzzy Control Theory	⋆	⋆	✓	✓	⋆	⋆
C.4	Supervisory Control Theory	✓	✓	✓	✓	✓	✓

other hand, machine learning methods (Row B) lack robustness against corner cases. Online machine learning methods that can learn during runtime can provide better autonomy to response to abrupt runtime changes in objectives. Single-Input-Single-Output (SISO) control theoretic approaches (Row C.1) provide means to address **robustness (1)**, **formalism (2)** and **efficiency (3)**, while lacking the ability to concurrently coordinate and control multiple objectives in a non-conflicting manner. Recently-proposed Multiple-Input-Multiple-Output (MIMO) control (Row C.2) enables **coordination (4)**, addressing attributes (1) to (4). However, MIMO control lacks **autonomy (5) and scalability (6)** for complex systems. In addition, to take advantage of benefits of heuristic methods and **robustness** and **formalism** of control theory, fuzzy control theoretic methods (Row C.3) have been proposed for runtime resource management of heterogeneous systems. In order to address these six key challenges in dynamic resource management of complex many-core system, Supervisory Control Theory (Row C.4) has been proposed as a **scalable (5)** and **autonomous (6)**.

We are seeing an increasing number of complex cyber-physical systems (CPS) deployed for various applications, such as road-traffic control involving communicating autonomous cars and infrastructure, or smart grids controlling energy delivery down to the individual device.

These distributed applications follow common design objectives, such as energy-efficiency, and require guarantees for high availability, real-time or safety. In this context, autonomy is crucial: multiple system goals varying over time need to be adaptively managed and objectives holistically coordinated. By empowering future CPS with self-awareness, these systems promise to dynamically adapt, learn, and manage unforeseen changes [89].

In the next two chapters we are going to explore self-Awareness for Systems-On-Chip. First, on we are going to explore frameworks that enables sef-adaptivity and self-optimization including simulation, software frameworks and hardware platforms. Thereafter, we will describe some of the case-study projects that demonstrates the significant gains achieved by utilizing self-awareness in managing SoCs.

Chapter 4

Self-Awareness experimental platforms and frameworks

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 [89]. Computational self-awareness in itself is not a new field, but rather a unification of subjects studied disjointly in various fields including control systems, artificial intelligence, autonomic computing, software engineering, among others, and how such research can be applied toward building computer systems with varying degrees of self-awareness in order to accomplish a task [107].

4.1 Cyber-physical Systems-on-Chip

Battery-powered devices are the most ubiquitous computers in the world. Users of battery-powered devices expect support for various high performance applications running on same device, potentially at the same time. Applications range from interactive maps and navigation,

to web browsers and email clients. In order to meet performance demands by users utilizing complex workloads, increasingly powerful hardware platforms are being deployed in battery-powered devices. Systems-on-chip (SoCs) integrate potentially hundreds of (heterogeneous) cores and uncore components on a single chip, constrained by a limited amount of system resources (e.g., power, interconnects), are required to support diverse workload characteristics with conflicting constraints and demands, with increasing pressure on shared system resources from data-intensive workloads. These platforms include a number of configurable knobs throughout the system stack and with different scope that allow for a tradeoff between power and performance, e.g., dynamic voltage and frequency scaling (DVFS), power gating, idle cycle injection. These knobs can be set and modified at runtime based on the 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 huge range of potential tradeoff. With such a large number of possible configurations, SoCs require intelligent runtime management in order to achieve system goals for complex workloads. Additionally, the knobs may be interdependent, so the decisions must be coordinated.

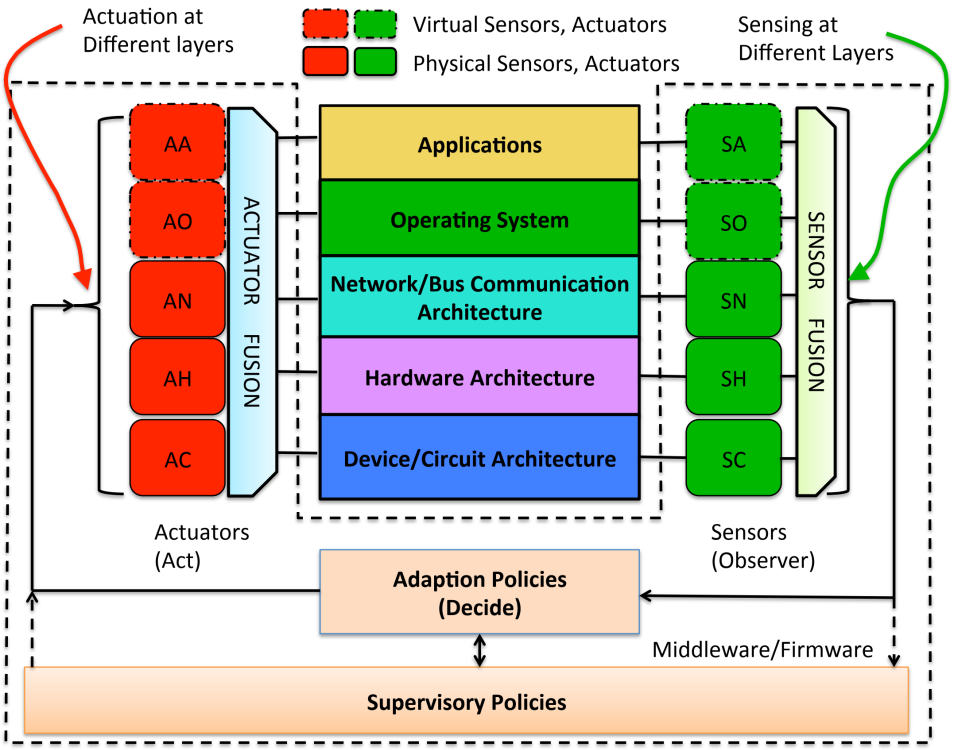


Figure 4.1: CPSoC infrastructure: sensors and actuators throughout the system stack, with support for adaptive policies that enforce a given goal (from [50]).

Cyber-physical systems-on-chip (CPSoC) [191] provide an infrastructure for system introspection and reflective behavior, which is the foundation for computational self-awareness. Figure 4.1 shows the infrastructure of a sensor-actuator rich platform, integrated with decision-making entities that observe system state through virtual and physical sensors at various layers in order to set the system configuration through actuators. The actuations are determined by policies that enforce the overall application goals while considering system constraints. Such an infrastructure can deploy *re*active policies through the traditional *Observe*, *Decide* and *Act* (ODA) feedback loop, as well as *pro*active policies through the augmented *self-aware* feedback loop. Figure 4.2 shows how the traditional ODA loop is augmented with reflection to provide self-aware adaptation. In this chapter we explore the use of computational self-awareness to address challenges of adaptive resource management in cyber-physical systems-on-chip[1].

[1]Throughout the remainder of this chapter we use SoC as an umbrella term that includes CPSoC.

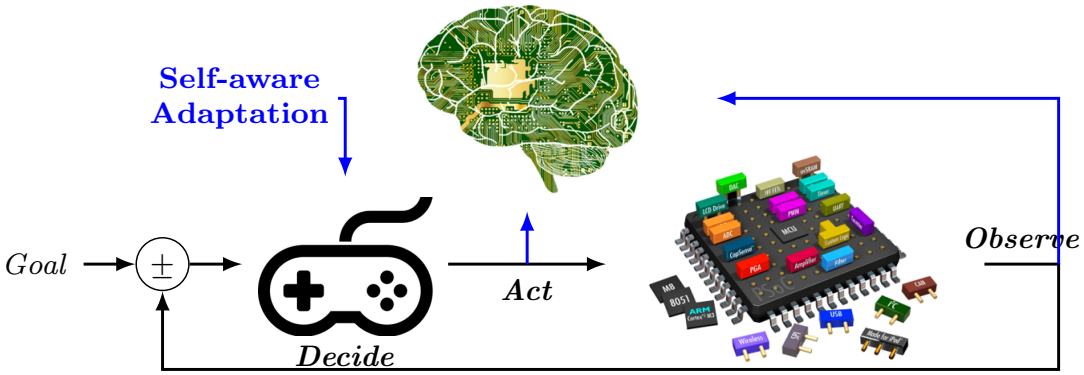


Figure 4.2: Self-aware feedback loop. Policies are deployed to make action decisions toward achieving a goal by controlling the CPSoC based on observations and self-aware adaptation.

4.1.1 Reflective System Models

Traditionally, resource managers deploy an ODA feedback loop (lower half (in black) of Figure 4.3) to manage systems at runtime. However, recent works [46, 208] have shown that a runtime model of the system can better manage the unpredictable nature of workloads.

Reflection can be defined as *the capability of a system to reason about itself and act upon this information* [206]. 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 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 policies invoked more frequently than the decision loop. The top half of Figure 4.3 (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 predictive model is to estimate system behavior based on potential actuation decisions as well as system dynamics.

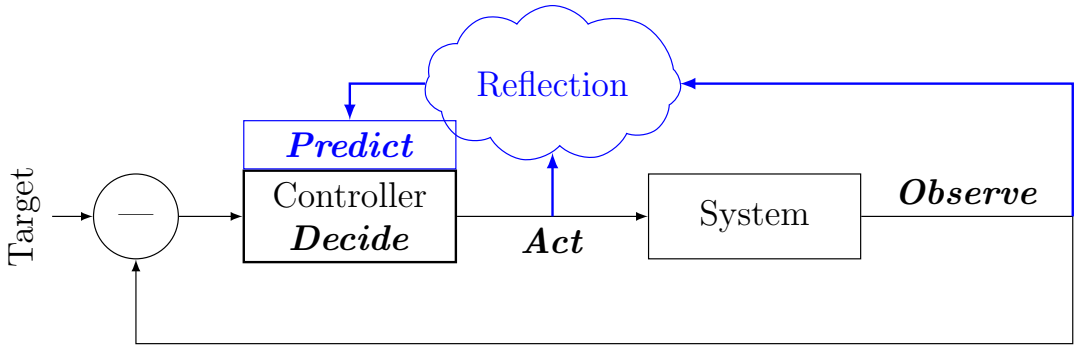


Figure 4.3: Feedback loop overview. The bottom part of the figure represents a simple observe-decide-act loop. The top part (in blue) adds the reflection mechanism to this loop, enabling predictions for smart decision making.

4.1.2 Middleware for Reflective Decision-making

The increasing heterogeneity in a platform's resource types and the interactions between resources pose challenges for coordinated model-based decision making in the face of dynamic workloads. Self-awareness properties address these challenges for emerging SoC platforms through reflective resource managers. Reflective resource managers build a model of the system which represents the software organization or the architecture of the target platform. Resource managers can use reflective models to anticipate the effects of changing the system configuration at runtime. However, with SoC computing platform architectures evolving rapidly, porting the self-aware decision logic across different hardware platforms is challenging, requiring resource managers to update their models and platform-specific interfaces. To address this problem, we propose MARS (Middleware for Adaptive and Reflective Systems), a cross-layer and multi-platform framework that allows users to easily create resource managers by composing system models and resource management policies in a flexible and coordinated manner.

4.2 Simulation frameworks

4.2.1 MARS

Figure 4.4 shows an overview of the MARS framework (shaded), with *Sensors* and *Actuators* interfacing across multiple layers of the system stack: *Applications*, *Linux kernel*, and *HW Platform*. The components of MARS are explained next.

1. **Sensors and actuators**: The sensed data consists of performance counters (e.g. instructions executed, cache misses, etc.) and other sensory information (e.g. power, temperature, etc.). The collected data is used to assess the current system state and to characterize workloads. Any updates to the system configuration (e.g., CPU core frequency, GPU frequency, memory controller frequency, task-to-core mapping) happens through system knobs. Actuators allow system configuration changes to optimize operating point or control trade-offs.

2. **Resource Management Policies**: are platform agnostic user-level daemons implemented in MARS using supported sensors, actuators and reflective system models.

3. **Reflective system model** is used by the policies to make informed decisions. The reflective model has the following subcomponents:

   (a) Models of *policies implemented by the underlying OS kernel* used for coordinating decisions made within MARS with decisions made by the OS.

   (b) Models of *user policies*, that are automatically instantiated from any policy defined within MARS.

   (c) The baseline *performance/power model*. This model takes as input the predicted actuations generated from the policy models and produces predicted sensed data.

4. **The policy manager** is responsible for reconfiguring the system by adding, removing, or swapping policies to better achieve the current system goal.

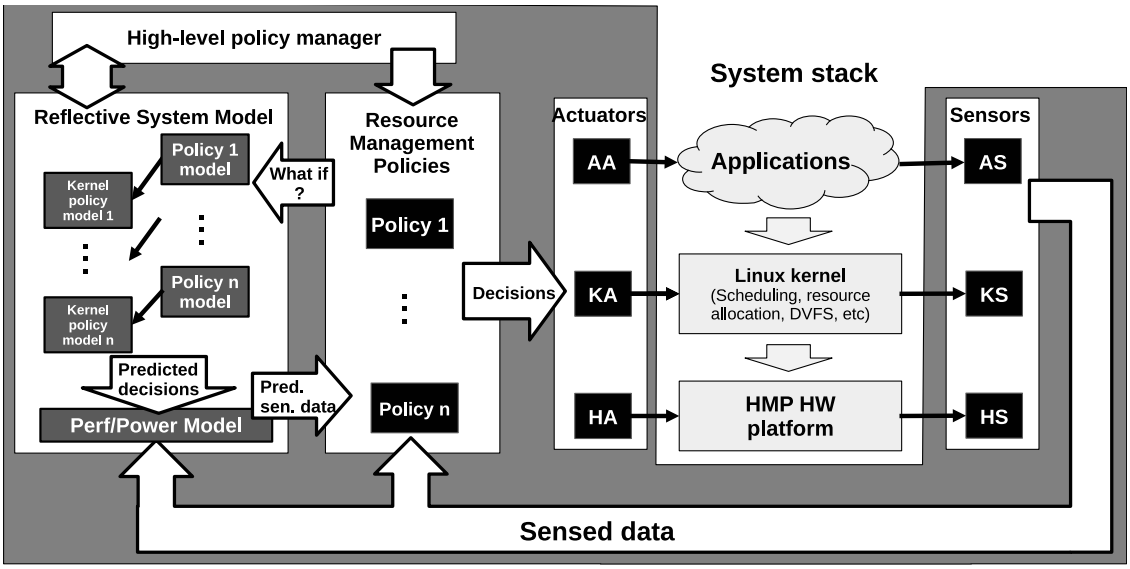


Figure 4.4: MARS framework overview from [155]. Different layers of the system stack coordinate through policies to orchestrate the management of resources: sensors inform policies of the system state; policies coordinate with models to perform reflective queries, and make resource management decisions; policies set actuators to enact changes on the system.

MARS is implemented in the C++ language following an object-oriented paradigm and works on hardware (e.g., Odroid-XU3, Nvidia Jetson TX2), simulated (e.g., gem5), and trace-based offline [154] platforms. The framework is open source and available online.[2] While the current version of MARS targets energy-efficient heterogeneous SoCs, we believe the MARS framework can be ported to a wider range of systems (e.g., webservers, high-performance clusters) to support self-aware resource management.

Self-adaptive software can be defined as "software that evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible"[111]. In this work, we do not address self-adaptive systems directly, as these systems encompass many high-level concepts such as self-configuration, self-healing, self-awareness, self-optimization, and others,

[2]Code repository at https://github.com/duttresearchgroup/MARS.

also referred to as *self-*∗[189] (see [51] for a comprehensive review self-adaptivity for SoCs). However, an infrastructure for system introspection and reflective behavior is an important building block for such systems.

Figure 4.4 shows an overview of the MARS framework (shaded), with *Sensors* and *Actuators* interfacing across multiple layers of the system stack: *Applications*, *Linux kernel*, and *HMP HW Platform*. On-chip resource management requires modeling and runtime policies for these different layers of the system stack. Resource management decisions must be made for each layer, and the frequency at which decisions are made may vary within or between layers. Coordinating multiple resource management policies is challenging, and that challenge compounds when developers are required to design portable policies for multiple platforms. We have developed MARS – *Middleware for Adaptive and Reflective Systems* – to address these challenges.

4.2.2 Interfaces and Policy Design

We first look into how MARS provides a generic user-level sensing/actuation interface that allows for **portable** policy design. Table 4.1 provides a list of common sensors and actuators available at each level of the system stack.

Sensors

Sensor data may consist of physical or virtual performance counters (e.g., instructions executed, cache hits), or other sensory information (e.g., power, temperature). Sensor data assesses the current state of the system and also characterizes the workload. All sensors expose a *virtual interface* for communication with the resource manager. To implement the interface, MARS utilizes function templates to provide a generic `sense` function that returns

the specified sensed metric. Different versions of the function can be implemented for each platform, resource, and sensor-type combination. The function's template parameter specifies the proper function implementation to obtain the requested sensed data. This separates the platform-specific implementation from the user-level policies.

Level	Actuators (**A**)	Sensors (**S**)
Application	Degree of parallelism, algorithmic choice	Performance, Quality-of-Service
Kernel	Task-to-core mapping, task priority, memory allocation	CPU time, utilization, context switch counters, open file, number of tasks
Hardware	Voltage/frequency scaling, clock gating, power gating, cache sizing	Performance counters (e.g., cache misses, instructions executed, branch miss-predictions, etc), power, reliability, critical path delay degradation

Table 4.1: Examples of sensors and actuators available across the system stack

Actuators

Adjustments to system configuration at runtime happens through actuators. Actuators range from application-level choices (e.g., degree of parallelism), to kernel-level choices (e.g., task-to-core mapping), to hardware-level choices (e.g., core operating frequency). The `actuate` function sets a new value for a system actuator using a *virtual interface*. Similarly to `sense`, `actuate` is implemented with function templates so that unique versions may be provided for each platform, resource, and actuator combination. Given the new actuation value, the function's template parameter selects the proper function implementation to perform the requested actuation action. Two additional related functions are provided: `actuationVal` returns the current actuation value set and `actuationRanges` returns the valid range of actuation values for the specified resource. `actuate` functions are implemented on top of standard Linux system calls and modules. For instance, we use CPU affinity system calls to

control task-to-core mappings, and *cpufreq* to control core frequencies.

Policies

Policies contain the decision-making logic which controls the system resources at runtime. Management policies implement the ODA and refelction loops for the resource(s) under control. In MARS, users can define resource management policies by creating a subclass of `Policy`.

All policies make use of generic sensing and actuation interfaces `sense` and `actuate`. At the user-level, all policies share access to the common set of sensors. For each policy, sensor data is aggregated during the *sensing window* (i.e., the time period between policy invocations) defined for the policy. MARS uses `ioctl` system calls to setup the sensing windows for each registered policy function. The policy functions are executed at the end of each sensing window in separate threads that uses blocking `ioctl` calls to synchronize with the kernel module. The kernel module automatically aggregates sensed data on a per-window basis and stores the information in shared memory, so that it can be read directly by `sense` calls with low overhead. This means any number of sensing windows (i.e., policies) executing at different window lengths can access the same sensors concurrently, and still measure the appropriate value for their window length.

4.2.3 Reflective System Model

Previously, we described a scenario where multiple policies operate at different sensing window lengths, or epochs. We now illustrate how MARS uses a reflective system model to make self-aware decisions.

To understand the reflection mechanism in MARS, consider the resource manager shown

in Figure 4.5 that shows a sample task mapping policy interacting with DVFS (Dynamic Voltage and Frequency Scaling) and scheduler policy models. At the finest time granularity, we have the operating system scheduler, whose goal is to select a task to execute on a given core. A new decision must be made whenever a new task is created, a tasks finishes, a task's quantum expires, an interrupt is raised, etc., leading to a time-frame between decisions on the order of microseconds. At a coarser time granularity we have the DVFS policy, which typically executes periodically (10-100 milliseconds) to analyze the system load and select an appropriate operating frequency. At the coarsest time granularity (100-1000 of milliseconds), the task mapping policy runs periodically to define a new task-to-core assignment. Migrating a task from one core to another has significantly more overhead than changing the CPU frequency in a typical HMP [241]. MARS allows users to **coordinate** among different policies through policy models, regardless of varying time granularity.

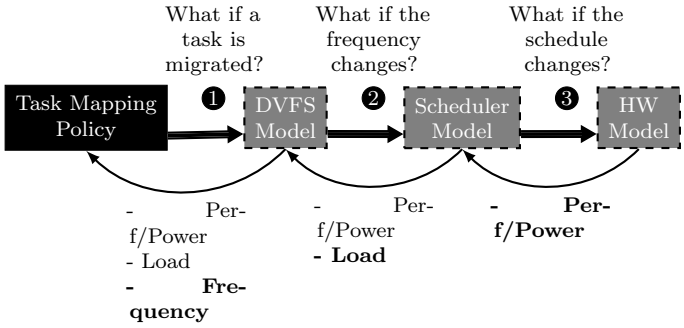


Figure 4.5: Example of a task mapping policy that queries models of OS policies for DVFS and scheduling.

In order to make an informed task mapping decision, for instance, the policy must consider the effects of its decision on the behavior of the underlying DVFS and scheduling policies. Furthermore, the invocation period of actuations dictates how complex the decision making logic can be. For instance, a scheduling decision must be made in the sub-microsecond range in order not to disrupt the system. Task-to-core mapping, on the other hand, is done comparatively infrequently, and affects the system performance over a long timespan. Therefore, the overhead of using complex models to make such decisions can be mitigated by the potential benefits of an informed decision.

The components within the reflective system model interact in a hierarchy defined by the dependencies of the actuations performed in the system. For instance, Figure 4.5 illustrates the scenario within MARS for our example. **Workload models** assume each core can run multiple tasks and there is no formal or explicit dependency between threads. Before the task mapping policy decides to migrate a task, it (1) queries the reflective model asking: *what will be the performance of task A if it is migrated?* (2) The DVFS governor policy model executes (*without* actuating) in order to predict the resulting core frequency provided the hypothetical task mapping. (3) This information is passed on to the performance/power model which predicts the task performance. **Architecture models** define the architectural characteristics of the target platform including *instruction-set architecture* (ISA), number of cores, core types, etc. Finally, the predicted metrics are used by the policy to make the decision, which is passed to the actuator through the actuation interface.

Analogous to the `actuate`/`sense` functions described previously, MARS provides the `tryActuate` and `senseIf` set of functions necessary issuing queries to the reflective models:

- `tryActuate`: Using the same syntax as `actuate`, this function updates the underlying models used to predict the next system state given the new actuation value. It does not set the actual actuation value. A `tryActuationVal` analogous to `actuationVal` is also provided.
- `senseIf`: this function has the same semantics as `sense`, but returns predicted sensed information for the next sensing window, given a new actuation set by `tryActuate`.

4.2.4 Policy Manager

A user-defined subclass of `PolicyManager` is responsible for composing policies and models together. The purpose of the policy manager is to provide resource management autonomy

Table 4.2: Currently supported platforms in MARS and their sensors/actuators

Implementation	Platform	Architecture	Sensors	Actuators
ODROID	Hardware	ARMv7-A	All ARM Performance Monitoring Unit (PMU) counters (e.g., cache misses, instructions executed, branch mispredictions, etc.,), Power sensors for CPU clusters	Voltage/frequency scaling for CPU, Task-to-core mapping, Temperature sensor
NVIDIA Jetson-TX2	Hardware	ARMv8-A	ARM PMU counters, GPU performance counters, CPU Power, Memory Power, GPU Power, WiFi Power	Voltage/frequency scaling for CPU clusters, GPU frequency scaling, Task-to-core mapping mapping
Gem5	Simulation $*$		ARM PMU counters	Frequency scaling
Offline simulator	Simulation $*$		Sensors which are present in the trace	Switch across different traces of execution

in response to changing system goals or execution scenarios, as well as coordinate multiple dependent policies and their objectives. For instance, consider modern smartphones. These devices typically operate in two scenarios: 1) the device is plugged to an external power source; 2) the device is powered by battery. In the case of (1), policies can simply focus on maximizing applications' quality of service (QoS), while for (2), QoS should be balanced with energy efficiency. A third scenario in which the battery charge is critically low is also possible. In this case, policies should focus on minimizing power consumption so the device can continue to operate. Furthermore, direct intervention from the user may cause additional unpredictable scenario changes. Creating a single policy that is able to manage all these scenarios and goals can lead to a overly complex and possibly inefficient implementation. Instead, one may develop multiple policies optimized for specific cases, and have a high-level manager dynamically select the most appropriate policy to apply throughout runtime.

4.2.5 HAMEX

With the diverse application demand in embedded and even high end IOT domains, heterogeneous systems are becoming popular more than ever. Heterogeneity in these systems is manifested in many different parts. It can show itself in computational units; for example heterogeneous processors that accommodate different types of CPU cores or in architectures with various accelerators such as GPU that can perform high throughput computation. Heterogeneity also appears in different interconnect and memory modules, particularly with the availability of many new memory technologies (e.g., NVM, HMC, etc.). Figure 4.6 depicts a general picture of a heterogeneous computer architecture composed of heterogeneous CPUs, GPUs, accelerators and memories. Although many existing tools are available to explore and design each component of Figure 4.6, there is lack of a framework that enables designers to explore rapidly the combined system ensemble of heterogeneous CPUs, GPUs, memories and interconnects. This section presents HAMEX, a Heterogeneous Architecture and Memory EXploration framework that addresses this need and empowers designers to perform rapid Design Space Exploration (DSE) of alternative heterogeneous system architectural configurations which covers the CPU, GPU, memory and interconnect dimensions.

HAMEX DSE FRAMEWORK

The following section explains the overall structure and role of individual components of our HAMEX design space exploration (DSE) framework. HAMEX accepts an input specification file that determines the characteristics of architecture components. In addition to properties of memory subsystem, this file includes high level information about the type and number of CPU and GPU units and their caches.

We integrated cycle accurate Gem5 [21] CPU simulator and Multi2Sim [220] GPU simulator for modeling these computational units. Figure 4.6 shows general view of the HAMEX

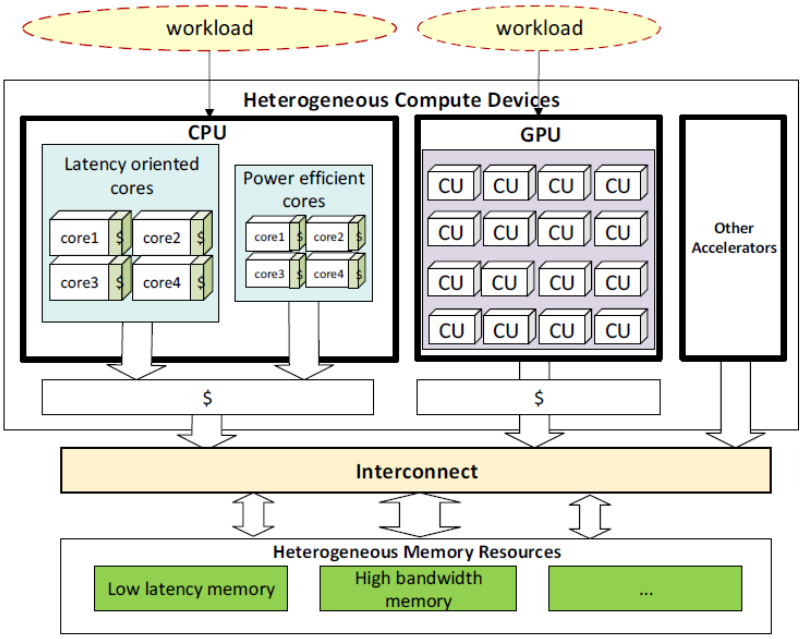


Figure 4.6: HAMEX simulation framework overview [149]

framework. Components defined in the input specification file can have multiple sets of options to enable design space exploration. The output of HAMEX will be generated by the memory exploration component integrated with DRAMSpec and DRAMPower [228] which includes metrics related to bus contention, performance and power of memory modules examined.

Model configuration

CPU simulation: We use Gem5 [21] as our CPU simulator. Gem5 is a cycle accurate micro-architectural simulator which provides simulation for many ISAs. This includes in-order and out-of-order cores from different ISAs that are highly configurable regarding the micro-architectural parameters such as pipeline stages, load/store queue entries and reorder buffer entries. This tool also provides flexible configurations for caches, memories and interconnects. Gem5 has two simulation modes: Full System (FS) and System-call Emulation (SE). FS mode simulates a complete system with devices and operating system. FS mode is particularly

useful for designers that want to analyze effects of OS scheduling on their applications performance. In SE mode, system services are provided by the simulator hence it only needs a statically linked user-space program.

GPU simulation: For simulating general purpose parallel workloads, the Multi2Sim [220] GPU simulator is selected and integrated in our HAMEX framework. For this purpose, GPU and dependent modules architectural characteristics are extracted from the input specification file and fed into the simulator. This specification file determines the type and number of compute units in the GPU and their properties such as frequency, wave fronts and optional pipeline details. In addition, designers may specify the cache and main memory architecture used in GPU memory hierarchy. Multi2Sim requires a series of networks and ports to connect compute units, caches and memory modules, all of which are generated by the HAMEX architecture manager (block B in Figure 4.6). In order to fully model a GPU execution flow, an emulated CPU is added to simulation to start the command queue and transfer the input to GPU unit.

Architecture manager

At this point in the flow of HAMEX, both CPU and GPU workloads have been simulated on their corresponding components and their memory traces have been captured. As shown in block (B) of Figure 4.6, the architecture manager in addition to CPU and GPU traces, receives memory features such as type, size and bandwidth of memory from the input specification file. Using this information, the manager invokes memory exploration component comprised of DRAMSpec integrated with Gem5 memory and protocol buffer traffic generator. The designers have the option to specify a configuration to modify trace replay. For example the architecture manager can add delays to the start of one of the workloads or add priority to one of the compute units, then generate the improved trace for traffic generator units. Interconnect used in memory exploration also can be configured for different width, coherency,

frontend and response latency, frequency, etc.

Memory System

One of the merits of our HAMEX framework is the ability to do extensive exploration on memory systems used in heterogeneous architectures. HAMEX supports two common early design space exploration use cases: 1) comparing common or new memory families, and 2) evaluating emerging new memory families.

Comparing memory components: The memory component and its interconnect used in HAMEX framework is modeled in Gem5 classic memory system integrated with DRAMSpec and DRAMPower for estimating performance and power metrics. There are many DRAM memory modules such as DDR3, LPDDR3, GDDR5,WideIO, etc. already modeled in this environment. This wide variety of available of the shelf components can provide designers with more design choices to explore at early stages of design.

4.2.6 Sniper feedback loop

For processor architectural simulation, we use the Sniper [69] simulator which provides micro-architectural details of power and performance of variety of processors. This architectural simulator enables evaluation of single-core and multi-core processors with different communication mechanisms such as bus and Network-on-Chip (NoC). A series of additions was made to this simulator in order to enable run time closed-loop power capping which are discussed in the rest of this section.

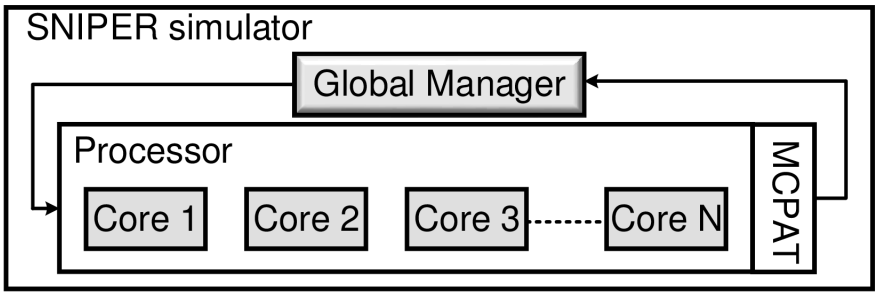


Figure 4.7: Simulation framework overview

Framework Overview

In our work, in order to enable run time power capping using PI controllers, a mechanism called "Global manager" is added to this simulator to manage the DVFS settings at run-time based on computer system response to application behavior. Figure 4.7 represents overview of this framework. By default, the global manager is invoked every 2.5 ms (common software controller epoch) to obtain the state of the computational cores and determine the next level for their frequency. In addition, MCPAT [120] is used to capture and estimate power consumption.

4.3 Hardware platforms

In order to validate the applicability of the proposed resource management methods in this thesis it would be vital to evaluate these methods on actual hardware platforms that are commonly available in market.

4.3.1 ODROID

Figure 4.8 shows the ODROID 8-core big.LITTLE Exynos 5422 HMP platform executing a set of representative applications on top of Linux or Android, thereby emulating the background

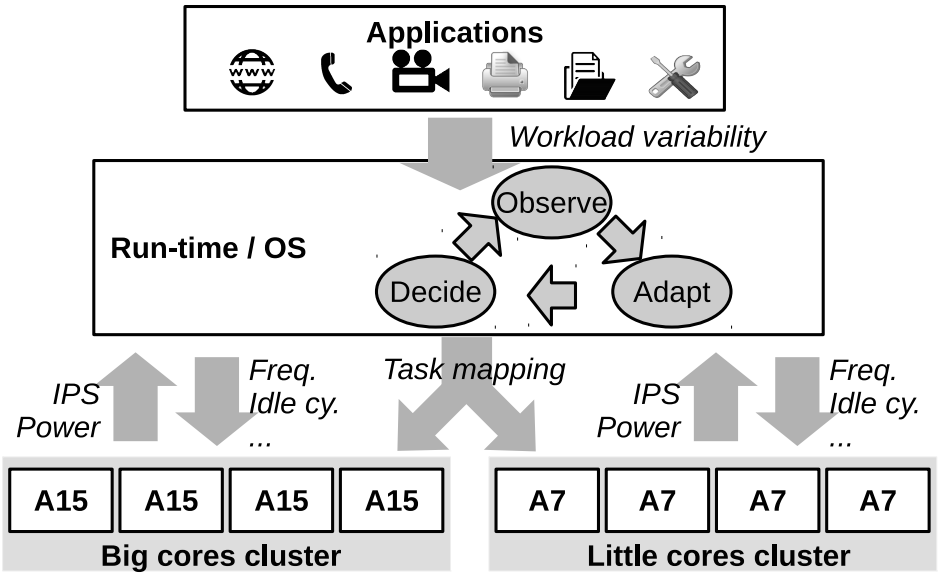


Figure 4.8: Example system overview.

noise in real platforms. This abstraction shows the sensors and actuators available for the 8-core HMP. This platform has an on-chip GPU as well along side the HMP. This platform provides the opportunity for runtime resource management using DVFS and idle core setting while measuring power and IPS.

4.3.2 NVIDIA JetsonTX2

In some of our studies in Chapter 5, we use the NVIDIA Jetson TX2 development board [34], which contains an HMP and a NVIDIA GPU. HMP contains a quad-core ARM Cortex A57 cluster and a dual-core NVIDIA Denver cluster. Similar to Cortex A57 cores, Denver cores implement ARMv8 instruction set and are designed as a processor with 7-way superscalar execution pipeline. The GPU is powered by NVIDIA Pascal CUDA cores. This platform enables us to consider multiple scenarios that are common in mobile devices where CPU runs multiple tasks (possibly one foreground with QoS requirements and others in background) and in full system scenarios GPU is executing a highly parallel kernel concurrently.

Chapter 5

Adaptive runtime resource management case studies

In this chapter we go dive deeper in some of the projects included in this thesis that show case usage of self-adaptivity in runtime resource management.

5.1 Single-Input Single-Output Controllers

5.1.1 Benchmark Categorization

Two sets of workloads have been utilized in our work in order to make a comprehensive study of capabilities of SISO controllers for power capping regarding the wide variety of applications behavior. There have been many efforts to construct benchmark suites that can comprehensively represent real world software execution. For example, SPEC [169] workloads include different high performance computing applications. ALPBench [136] is a suite of multimedia workloads. Minebench [158] includes benchmarks for Data Mining

Workload	Domain	**Problem size**
Barnes	High-Performance	32768 particles
Ocean-Contiguous	High-Performance	1024*1024 matrix
Ocean Non-contiguous	High-Performance	1024*1024 matrix
FMM	High-Performance	32768 particles
Radiosity	Graphics	room
Raytrace	Graphics	Car -m64
Water-NSQ	High-Performance	2197 Molecules
Water-SP	High-Performance	2197 Molecules
Volrend	Graphics	head

Table 5.1: SPLASH-2 benchmark list and their problem size

Workloads. In the recent years comprehensive benchmark suites like SPLASH2 [231] and PARSEC [19] gained a lot of attention as they cover many domains and in addition they scale well for multi-core systems. In our studies we use these two benchmark suites (PARSEC and SPLASH-2) and provide a detailed analysis on the effects of application behavior on controllability of the system. In addition, a set of micro-benchmarks are devised to stress various parts of a system that are further explained in discussion section.

SPLASH2

The SPLASH-2 suite is one of the most widely used collections of multithreaded workloads [231]. Table 5.1 represents a detailed description of workloads included in this benchmark suite. Parallel machines were not as common as nowadays and were mostly used for scientific objectives when SPLASH-2 benchmark suite was released. This fact is reflected in high performance nature of the workloads included in SPLASH-2 benchmarks.

PARSEC

PARSEC as one of the emerging multi-threaded benchmark sets contains applications that have been designed to take advantage of multiprocessor computers with shared memory [19]. Applications included in the benchmark suite are composed of programs from a wide range of

Workload	**Application Domain**	Parallelization
Blackscholes	Financial Analysis	Data-parallel
Bodytrack	Computer Vision	Pipeline
Canneal	Engineering	Data-parallel
Dedup	Enterprise Storage	Pipeline
Facesim	Animation	Data-parallel
Ferret	Similarity Search	Pipeline
Fluidanimate	Animation	Data-parallel
Freqmine	Data Mining	Data-parallel
Raytrace	Visualization	Data-parallel
Streamcluster	Data Mining	Data-parallel
Swaptions	Financial Analysis	Data-parallel
Vips	Media Processing	Data-parallel
X264	Media Processing	Pipeline

Table 5.2: PARSEC benchmark list and their Application Domain [18]

application domains (e.g., engineering, machine learning, storage, finance, Etc.) in order to capture the increasingly diverse ways in which computers are used. Containing applications with different parallel programming models that are geared toward common CMPs, brings out the possibility of using PARSEC benchmark suite to test the performance of a diverse set of computer systems including embedded systems.

5.1.2 Evaluation

In this section, we evaluate two often-neglected important aspects in the design of a controller: **System Identification** and **Performance Analysis** [195, 196]. For system identification, we show examples of both well identified systems and poorly modeled systems with some hints about what kind of behavior in the model may result in imprecise controller design. These evaluations are done for both single-core systems and CMPs with demonstration of effect of increasing number of cores. This can be valuable for cases where a controller must be implemented in hardware and changing its configuration is costly. For performance analysis, we evaluate various types of controllers for SPLASH2 and PARSEC workloads and highlight the pros and cons of each method. In addition, the trend of controlling behavior is analyzed

for multithreaded applications running on different size of CMPs. Furthermore, as part of our evaluation we categorize the workloads based on measurement of their power consumption and instruction per second (IPS) and then analyze the settling time (all time measurements done by epoch units), maximum overshoot and controllability of each class of application.

5.1.3 System Identification

After defining the controlled system, the first step would be to generate test wave forms from training applications for system identification [124]. Ideal training applications represent the behavior of applications to be executed on the real system [125]. A test waveform contains a series of samples for controller inputs and outputs for a training application, and should exercise as many input permutations as possible. Once the training data is collected, the model can be created. During this stage, the system dynamics is exercised often by applying a staircase waveform to the control input (e.g., operating frequency). Such staircase would stimulate system behavior in response to various levels of control input. In our work, we change CPU frequency from 1 GHz to 3.3 GHz with steps of 100 MHz. In this method, training sets use varying frequency (e.g., a set of out-of-phase staircase signals for the control inputs) in order to isolate the deterministic and stochastic aspects of the system. Voltage level is assumed to be fixed in this simulation. This model is then evaluated to predict the expected data from the identified system. Abnormal behavior from this model can raise a flag that the controller to be designed from this model might be inaccurate. In our work, we used Matlab's system identification toolbox for this process [141]. Below, we showcase some of these scenarios.

Figures 5.1 and 5.2 demonstrate positive and negative examples that designers have to look for in system identification phase. These two figures show the result of system identification of two hand-tuned workloads. More precisely, they show how well a model can predict the

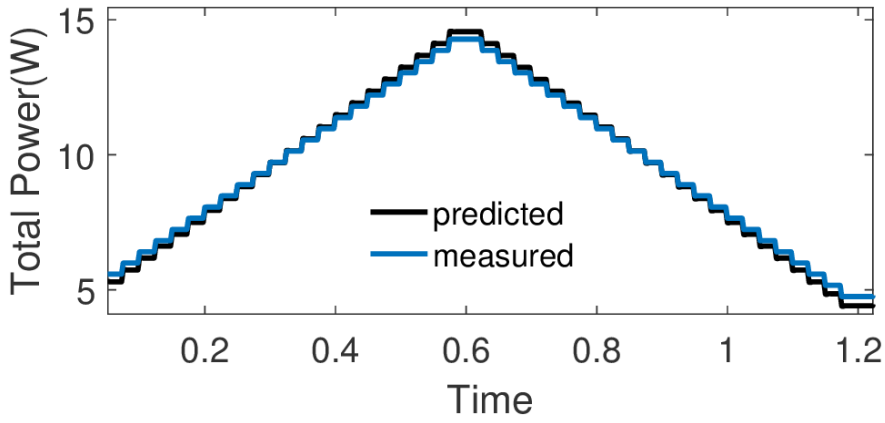


Figure 5.1: CPU bound microbenchmark with well identified model

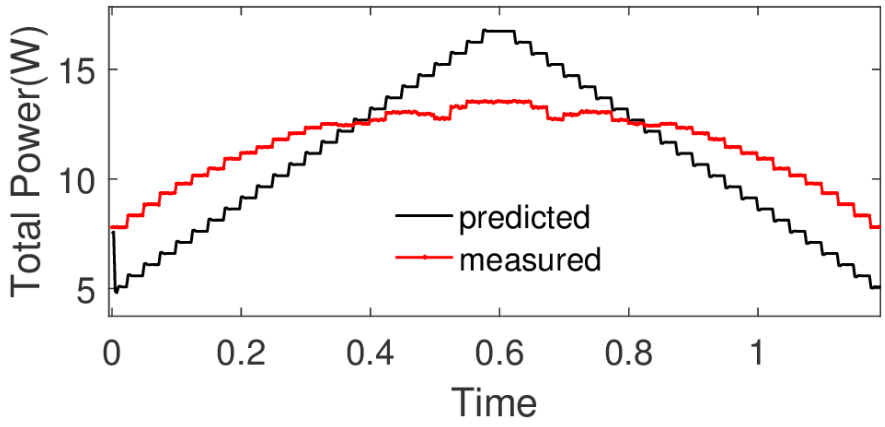


Figure 5.2: Memory bound microbenchmark model with limited tracking range

system's output running an application when operating frequency is changed in a staircase form over time. Figure 5.1 shows that the predicted model for a performance regulated benchmark that closely fits the measured model. On the other hand, Figure 5.2 shows a memory bound benchmark that lacks the ability to fit into the expected model. Although the controller changes the frequency levels, this change does not have a clear correlation to system output due to the system stalling for memory accesses instead of executing instructions for a majority of simulation cycles.

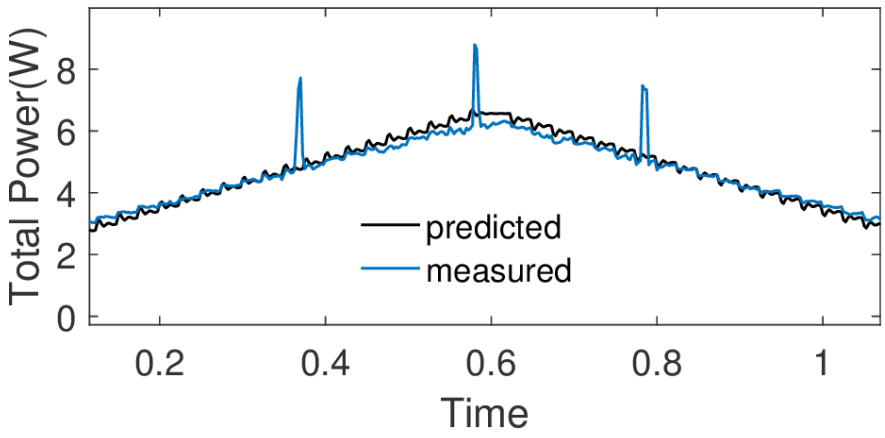


Figure 5.3: Barnes workload well identified model with noise

Single-Core Models

Next, we investigate a selected set of models that show a series of stochastic behavior that can manifest in the system identification stage. System identification results shown in this section are performed for a single core processor running one thread of each benchmark. Majority of these benchmarks are selected from SPLASH-2 benchmark suite to isolate the effect of off-chip memory accesses. Section 5.1.3 focuses on system identification for CMP models that mostly utilizes PARSEC benchmarks optimized for these architectures.

Figure 5.3 shows a part of the Barnes workload that closely fit the expected model while demonstrating spikes at certain points of time. These spikes can be the result of a change in the workload execution behavior which is common in many real-time applications. As the duration of these spikes are very short and the model can rapidly respond to such changes, they are considered as the stochastic part of the system dynamics which should be isolated from the deterministic part, and would not cause any issue in the performance of the system. In contrast, Figure 5.4 demonstrates part of an identified model for the Raytrace workload that exhibit a long period of underestimation. There are restrictions (such as level of aggressiveness and transient state) that can be applied during controller design stage which can mitigate these abnormal conditions, motivating the need to consider these issues upfront.

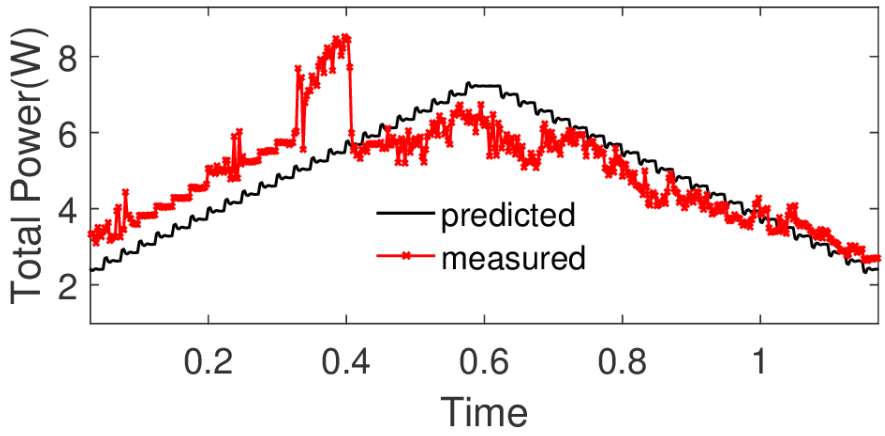


Figure 5.4: Raytrace workload model exhibiting error in prediction

CMP Models

After identifying models for single-core processors, we aim to identify models for CMP systems. This change expands the exploration space for controller design. On one side, exceeding number of cores can be a challenge for system identification phase and controller design process. On the other hand, the multithreaded behavior of applications over different cores can add a large noise when controlling the whole system complex with solely one SISO controller. To address some of the mentioned challenges we study the capability of SISO controllers in power capping of CMPs with following scenarios for the system identification phase:

- We demonstrate a single threaded application running on a multi-core processor. One core is executing the application thread and other cores simply have idle power consumption. The goal of this experiment would be to see the static effect of idle cores on system identification.

- We analyze the power model for CMPs using DVFS. We demonstrate the variety of dynamic power changes on a 4-core processor based on run-time behavior of each application thread. In addition, we evaluate the accumulated total power for the whole

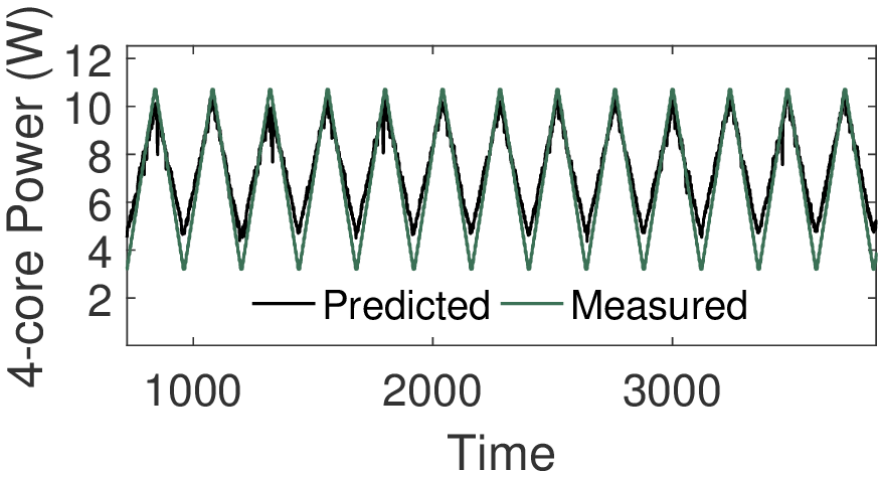


Figure 5.5: System identification for a 4-core system using one thread of Freqmine benchmark executing on one core. Model fits general flow with a static shift at the bottom.

system in respect to power consumption of each core and platform communication mechanism.

- We assess the system identification capability for CMP systems with low number of homogeneous cores.
- We illustrate the trend of increase in inaccuracy of system identification stage as the number of cores grows larger.

The first transition from a single-core processor to a multi-core processor system identification model is extending the architecture while keeping the same configuration for the software execution. This would reveal some of the key points to look after in design process of a controller for the multi-core systems. In order to evaluate this case, we evaluate same benchmarks on a 4-core platform while restricting the number of threads to one. Figure 5.5 shows the result of system identification for Freqmine benchmark. While the application is running on only one of the four cores, the other cores consume power in their idle state. This constant power usage would manifest itself with slight shift at the bottom of each staircase period. If the model can fit the general trend, this shift can easily be eliminated in control design step.

Furthermore, we take a look at general power actuations during black-box system identification for a 4-core multi-processor. Figure 5.6 shows power measurement while execution of Bodytrack benchmark on a homogeneous 4-core system. Top part of Figure 5.6 shows the staircase model of total power of the whole system that shows a good response to changes in system frequency. This model is able to rapidly recover from spikes and slopes that are caused by application stochastic behavior on each individual core. Four bottom system identification models in Figure 5.6 show individual core response to frequency changes. We can observe that each core depending on the running a thread shows unique run-time behavior but at the end we are concerned with aggregated total power model.

Next step is to analyze how well this model can fit the predicted model. Figure 5.7 demonstrates the system identification of the 4-core system that we have seen previously using *Bodytrack* benchmark. This model shows a promising trend for design of a SISO controller for the 4-core CMP. Majority of PARSEC benchmarks show similar results for system identification stage using a 4-core system. We were curious to see if it is possible to identify larger systems for an accurate control design. We extended the simulated architecture to a 64-core network on chip system. Figure 5.8 shows system identification for 64 thread of the same benchmark application (*Bodytrack*) on the 64-core platform. It can be easily inferred from Figure 5.8 that this model cannot be easily identified and the controller designed from this model might lead to unresponsive system.

To demonstrate the trend of decrease in accuracy of system identification, we picked one of the benchmarks (*Swaptions*) that showed well-fitted model for 4-core platform for further evaluation. We extended the simulation to 8, 16 and 32 cores to see the fit to model trend. Table 5.3 shows the decrease in the ability to fit the predicted model while increasing the number of processing cores. The two important notes from system identification stage is to evaluate the responsiveness of the controller to control inputs and grasp a better understanding of stochastic and deterministic behavior of application.

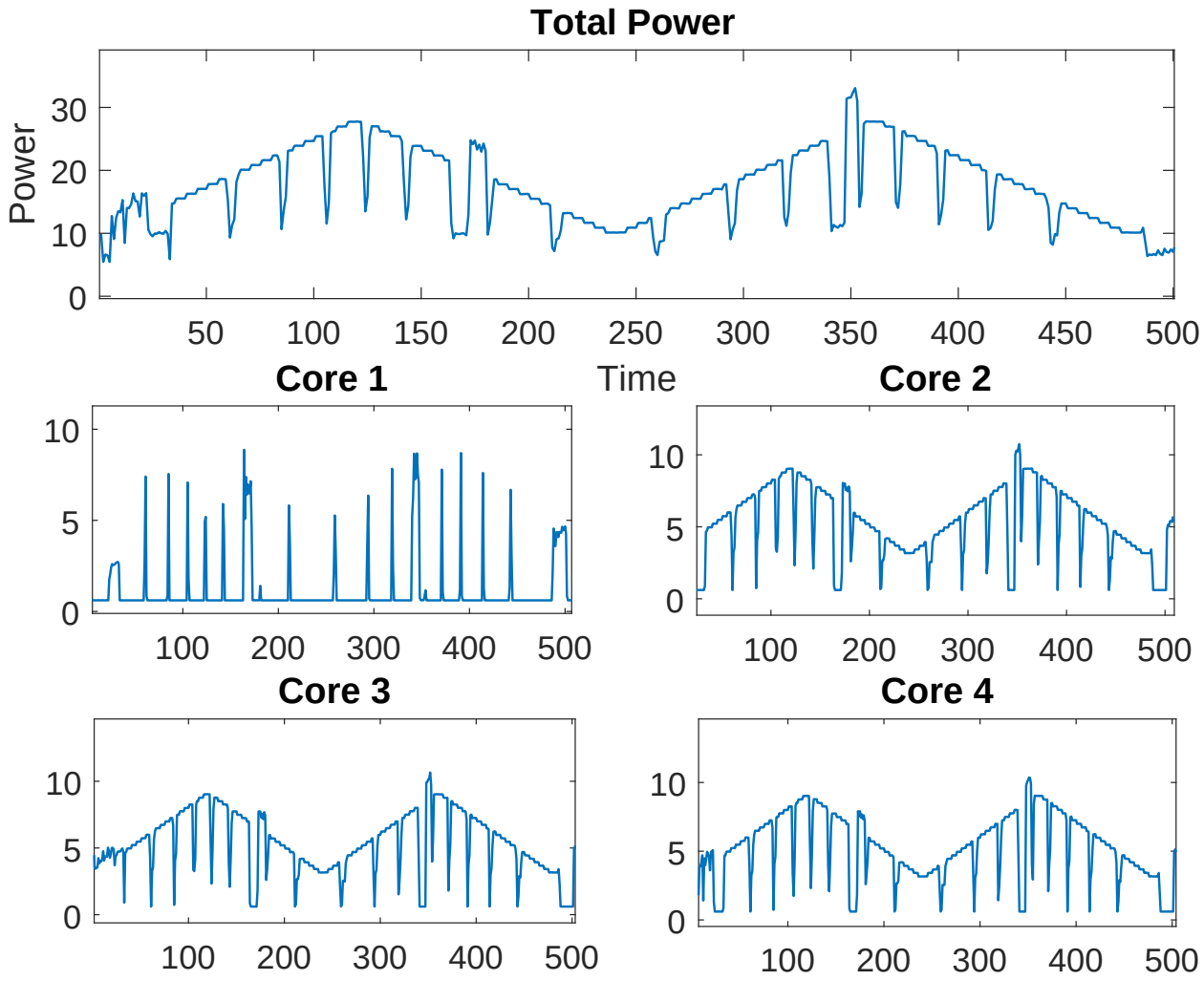


Figure 5.6: Power usage of a 4-core system while tuning Bodytrack benchmark. Top figure represents Total power of the whole system and the rest are break down of each core power.

To give better insight regarding the decrease in accuracy of system identification when moving form small number of cores (4-cores) to a platform with large number of cores (64-cores) a cross-validation of residuals has been done. Figure 5.9 demonstrates both cross-correlation and autocorrelation evaluation for bodytrack benchmark. Residual is the stochastic component (e.g., disturbance, noise, etc.) of the system output, which is not supposed to be included in the model. When validating the model, the model output is compared to noisy system outputs. Therefore we expect the residual to be pure noise. To verify this, the residual is analyzed for correlation. If there is no correlation between the residual and itself or any

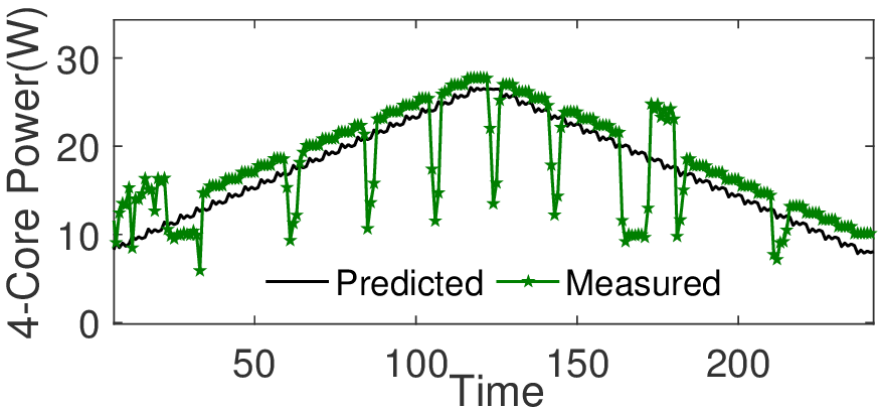


Figure 5.7: 4-core system identification for bodytrack benchmark

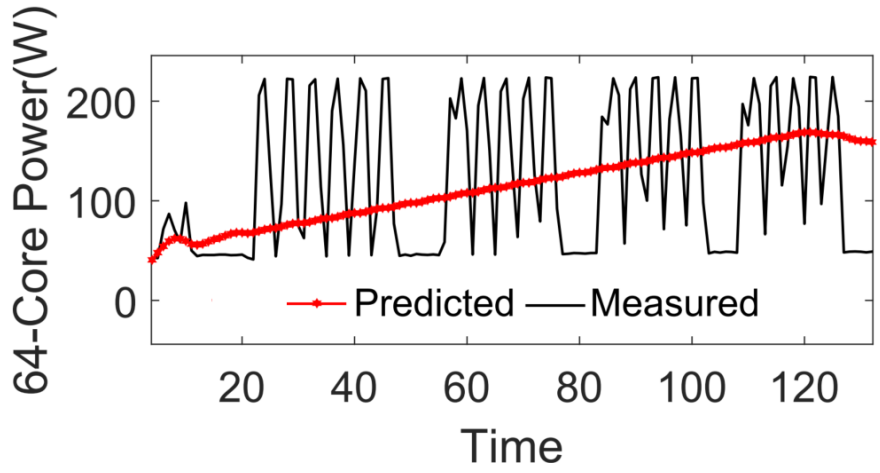


Figure 5.8: 64-core system identification for bodytrack benchmark

inputs, the model is accurate enough. *Confidence* can be used to specify a range. In this work, commonly used 99 percent boundaries have been set for the confidence. A confidence level is the probability with which the true output will fall into the range of confidence boundaries. After an estimated system dynamics is produced using system identification techniques, it is cross-validated using different data sets. Cross-correlation is a standard method of estimating the degree to which two series are correlated. In our case, cross-correlation (bottom part of

Benchmark	**4-cores**	**8-cores**	**16-cores**	**32-cores**
Fit to estimation	78.99%	40.7%	9.529%	3.004%

Table 5.3: Fit to estimation data trend with increase in number of computing cores while executing one thread of *Swaptions* Benchmark on each core.

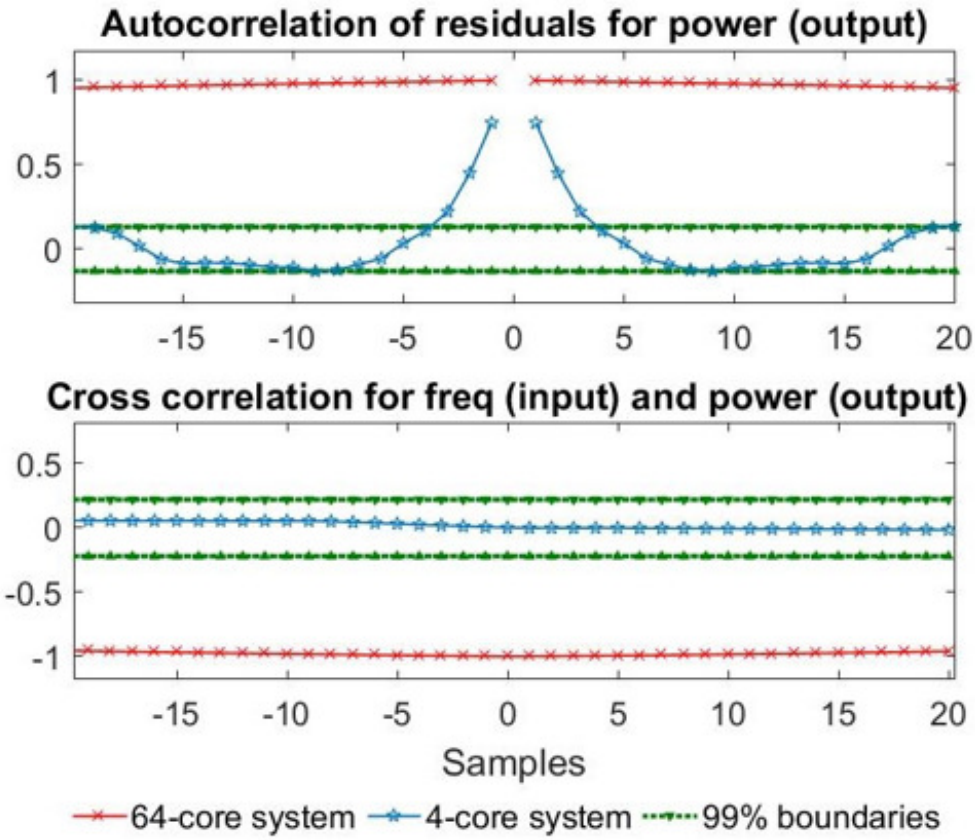


Figure 5.9: Auto/Cross-correlation of residuals for 4-core and 64-cor systems.

Figure 5.9) is evaluated for power as the output of the system based on the frequency as the input of the identified system. We can observe that the 4-core system model can retain in the confidence boundaries while the larger system model is outside these boundaries for all samples. The cross-correlation is similar in nature to the convolution of two functions. In an autocorrelation, which is the cross-correlation of a signal with itself, there will always be a peak at a lag of zero. The top part of Figure 5.9 compares auto-correlation of residuals for these two systems. Similar to the previous part, only the 4-core system identification can stay in the boundaries. These results show controllers for a system with large number of cores are often infeasible to design due to the lack of a sufficiently accurate system dynamics model.

5.1.4 Performance Analysis

After the system identification stage, controller design is performed by using the Matlab PI tuner [142]. Typically there are three ways that designers choose to design a controller for a computer system. The first set of methods take a statistical average of metrics gathered from system identification phase to represent the general case. The second scenario involves designing a controller for a system that runs predefined workloads (i.e., application specific) such as a smart watch or industrial plant machines. In this case, designers have the opportunity to tune the controller based on the application at hand for better accuracy. Table 5.4 shows these workload specific control parameters (i.e., gains) used to control the system running each benchmark (i.e., optimal application specific parameters extracted from Matlab). Finally, the third scenario uses a worst case configuration that performs conservatively for all benchmarks and is more robust against disturbances, however suffers from slow settling time. It should be mentioned that despite all these methods and vast variety in off-the-shelf controllers, there are some applications that cannot be controlled with a simple SISO controller and that would either require more advanced controllers (e.g., non-linear, adaptive, self-tuning) or different/more configuration knobs. We describe these scenarios in Subsection 5.1.4.

Workload	K_P	K_I	**Multithreaded**	K_P	K_I
Barnes	114	229	Blackscholes	10.08	20.17
Ocean-Contiguous	156	226	Bodytrack	12.50	25.10
FMM	114	229	Facesim	8.7	5.4
Radiosity	184	369	Ferret	23.2	46.4
Raytrace	244	247	Fluidanimate	20.17	40.35
Water-NSQ	139	228	Freqmine	68.1	136.01
Water-SP	175	250	Swaptions	39.1	40.2
Volrend	141	282	X264	137.1	47.85
Average	180	240	Average	38.86	45.18

Table 5.4: CPU core configuration for Nehalem-EP

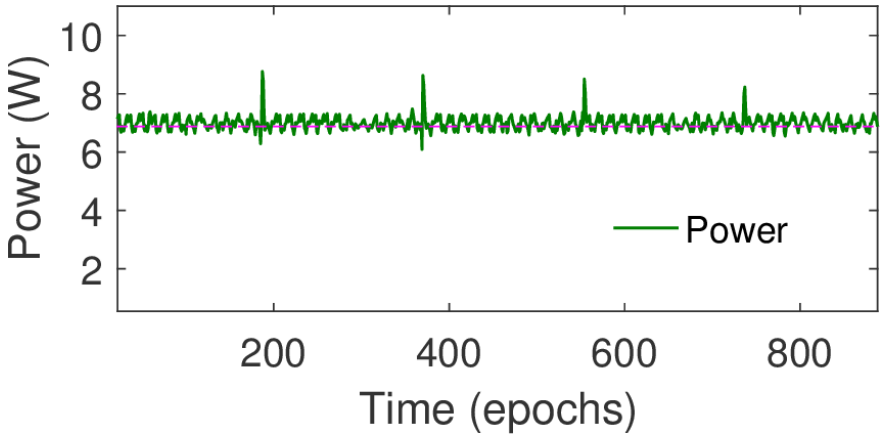


Figure 5.10: Example of well-tuned controller for *Water-NSQ* benchmark following 7W power reference

Customized case

For many systems using control-theoretic power managers, we may have design time knowledge regarding the workloads to be executed. This enables control designers to customize the power manager based on these pre-defined applications. System identification and controller design stages are performed individually on each application. Table 5.4 shows these workload specific K_P and K_I configurations. Figure 5.10 shows proper behavior of the *Water_nsq* benchmark in tracking the 7 Watts power reference. Ability to track a specific reference would be essential later on when DVFS manager wants to set a power reference to optimize energy efficiency. Examples from multithreaded applications are discussed in Section 5.1.4. We observe similar trends for all other workloads except the two benchmarks discussed in Section 5.1.4.

Average and worst case

Many general-purpose systems do not have the flexibility to accommodate customized controllers either due to variety of system workloads or because the controller cannot be easily reconfigured. In these situations, designers choose a representative configuration that can meet their requirements. Here two commonly reported control strategies use a statistical

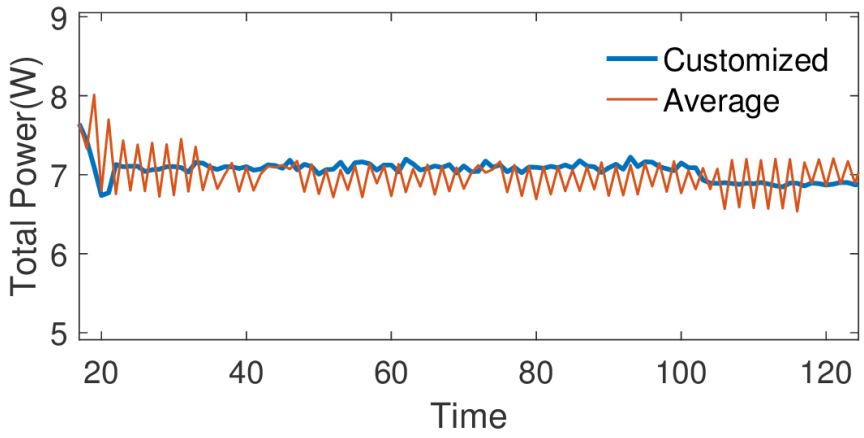


Figure 5.11: FMM benchmark with average and customized case

average case of predicted applications [146] or use a worst case scenario that can respond with slower speed but which provide larger margins of guarantees.

As an example for the average case, Figure 5.11 shows the difference between the customized controller and average case controller for the *FMM* benchmark in tracking the 7 Watts power reference. Both cases can keep the power close to the reference but the customized controller minimizes the tracking error with minimal deviations from the reference, while the average controller oscillates over the power reference. This is due to the fact that fine-grain step of the average case controller is larger than what this workload requires. For the worst case, Figure 5.12 shows the comparison between the customized case and worst case for the Raytrace workload. As expected, the worst case scenario has slower settling time due to smaller steps (smallest K_P and K_I) but after reaching 7 Watts, it can reliably follow the power reference.

CMP Controllers

Nowadays, most of computer systems including embedded systems utilize Chip Multi-Processors (CMPs) or Heterogeneous Multi-Processors (HMPs). The advantage of using multiple cores on a single die is that these multiprocessors become available commodity for

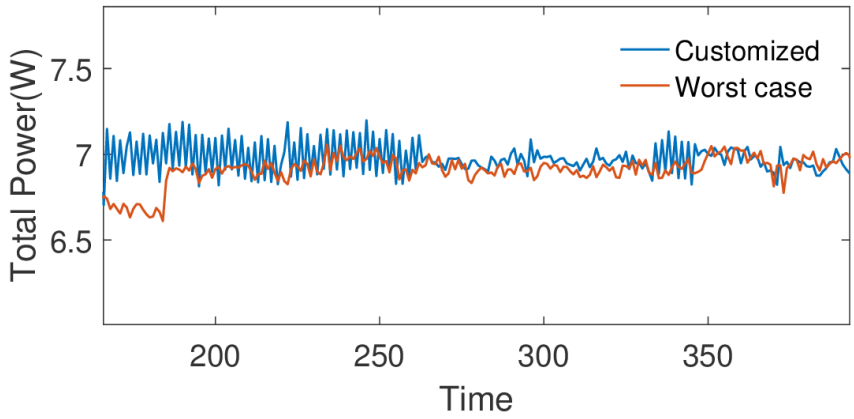


Figure 5.12: Raytrace benchmark with average and customized case

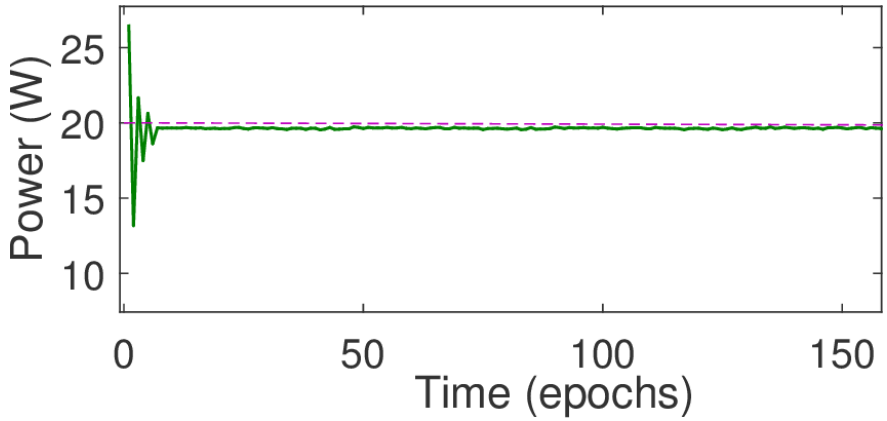


Figure 5.13: 4-core controller tracking 20 Watts for *Swaptions* benchmark.

parallel applications. In comparison to majority of SPLASH-2 benchmarks evaluated in previous section which are designed for high performance computers, PARSEC applications are optimized to take advantage of CMPs. Table 5.4 specifies the customized control parameters for some of these benchmarks.

To show some of the selected controllers designed for CMPs, we identify all PARSEC benchmarks for a 4-core system using bus communication and share memories. Insights from system identification phase were presented in Section 5.1.3. Here, we select some of these applications for control design using methods discussed in the beginning of Section5.1.4. Figure5.13 shows power reference tracking (20 Watts) for a customized controller for *Swaptions* benchmark running on a 4-core system. As shown here the SISO controller designed from a

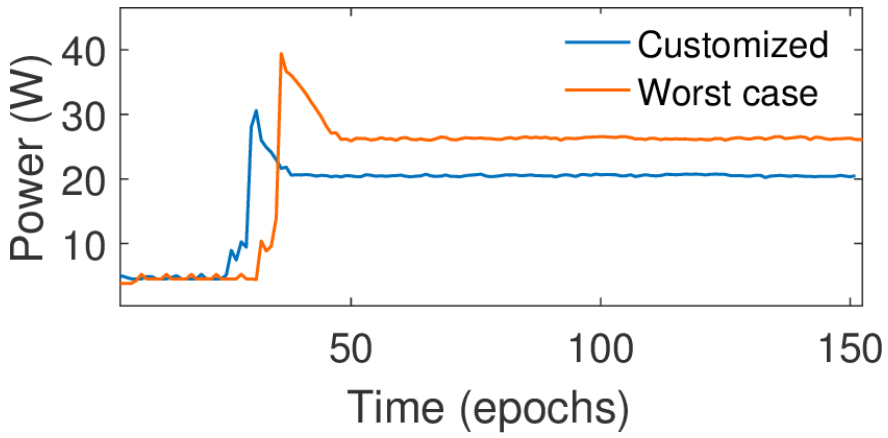


Figure 5.14: Comparison of customized and worst case controller for 4-core system tracking 20 Watts for *Facesim* benchmark.

well-identified model has no trouble controlling the total power of a 4-core system.

Figure 5.14 compares two controllers designed for *Facesim* benchmarks. First controller is a custom designed controller for this benchmark which shows rapid conversion to 20 Watts power reference with low overshoot. Second controller is a worst case controller that follows the same trend but slower and more sluggish. Also there is a bigger overshoot and steady-state error.

Corner cases

So far we evaluated both dynamic and static methods to design and deploy a PI controller for power capping. Using lessons learned from these evaluations, designers can choose the suitable method for their system. However, it is important to note that the appropriateness and feasibility of these methods depend on the system being *controllable*. The controllability property guarantees that the controller can always keep the plant within a set of boundaries around the reference. In other words, if the controller is not provided with proper means (actuators or configuration knobs), it would be unable to reach the desired reference. Figures 5.15 and 5.16 show the system identification, and controller deployment phases of the

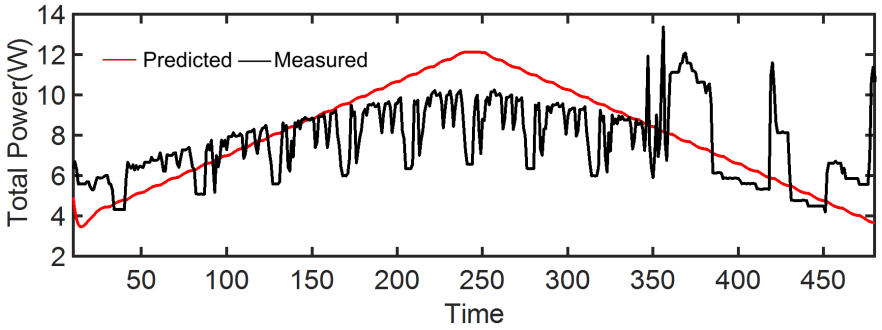


Figure 5.15: Ocean Non-Contiguous workload. System identification of uncontrollable workloads

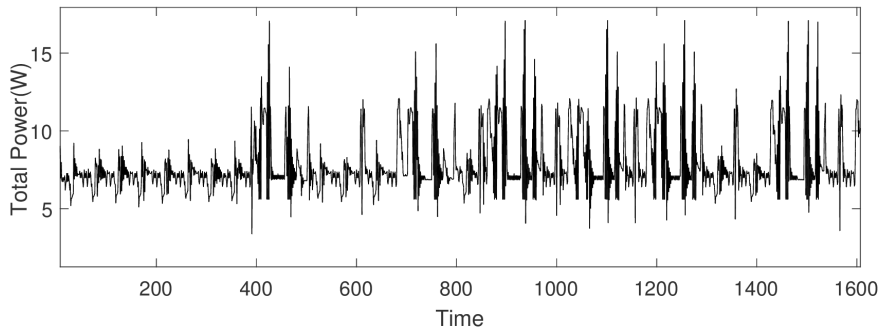


Figure 5.16: Ocean Non-Contiguous workload. Performance analysis of uncontrollable workloads while trying to track 7 Watts reference

system running *Ocean* benchmark. Both implementations of *Ocean* benchmark (contiguous and non-contiguous) show similar behavior. These applications are not controllable using solely DVFS actuation. In the following section, we analyze these benchmarks in more details to elaborate of the reasons behind their abnormal behavior.

5.1.5 Discussion

In this section, we discuss the reliability and performance of SISO controllers in power capping of different class of workloads based on the evaluations done in the previous section. Performance analysis done on the deployed controllers showed stability for majority of the workloads for single-core and 4-core CMPs. In addition, hand tuned controllers were able to meet the second set of requirements which are maximum 30% overshoot and settling time

less than 150ms. In some cases, controllers using the statistical average were not able to meet the overshoot requirements. The reason for more frequent power overshoots in average case is that it does not have the fine grain tuning that some of the workloads require. Although the worst case configuration was not as rapid as the customized controllers, overall it proved to be a reliable controller. Therefore, for scenarios where the computer system is designed to execute an application with similar computing characteristics, the average case can be a valid candidate; and for systems sensitive to changes in power levels that can tolerate some degree of performance overhead, worst case controllers can be deployed.

In our experiments, we observed few benchmarks that exhibited abnormal behavior in tracking power references with high standard deviation. Figure 5.16 shows the behavior of one of these benchmarks. Our first reasoning behind this behavior was that slow response time of a software controller is longer than the periods of time that these workloads change their application phases. This can cause a late response (change in frequency) to a phase that is already passed which can exacerbate the current power state. In order to check this issue we moved our software SISO controller mechanism to the hardware level with 10× faster sampling and DVFS epochs (from 2.5ms to around 0.25ms). Contrary to our expectations, the experiment showed that a faster controller did not have much improvement on these cases. Although we were able to capture power violations at an earlier stage, the responses of our controllers were not able to mitigate this issue. Figure 5.17 shows controller accuracy improvement when migrating from software controllers to hardware controllers. For most of controllable benchmarks, faster hardware controller shows small (less one percent) increase in accuracy but for the corner cases this faster response causes ripple effect and reduction in accuracy.

Our next solution to this issue was to investigate these benchmarks in more detail. We looked at a few measurable metrics and what stood out was the average power consumption. The results in Table 5.5 shows the average power consumption of each benchmark in SPLASH-

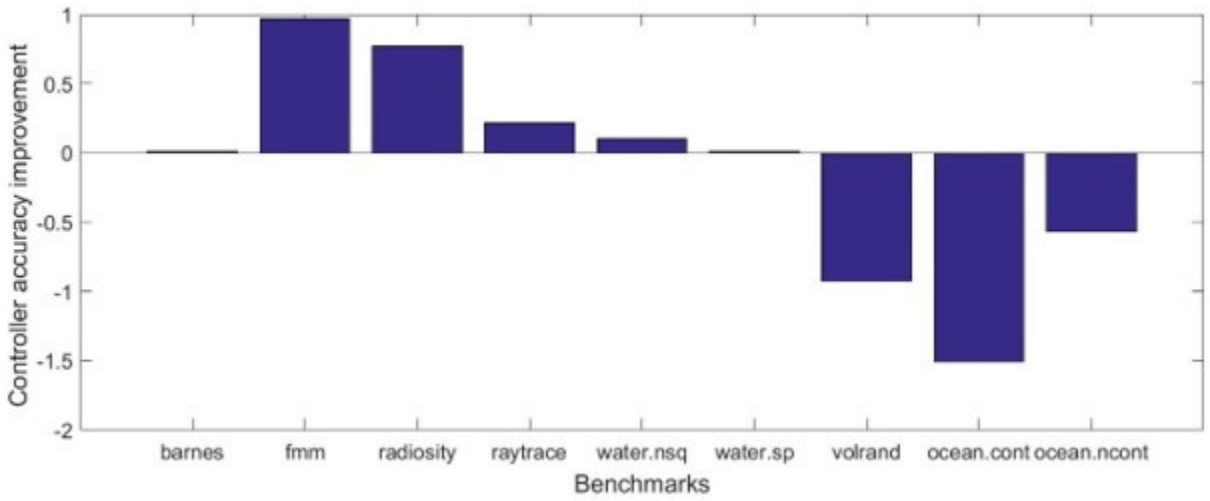


Figure 5.17: Accuracy improvement from software controller to a faster hardware controller

2 benchmark suite while tracking 7-Watt power reference. As we can see, only the two irregular benchmarks (Ocean-Contiguous and Ocean Non-contiguous) have the average power consumption higher than 7 Watts which results in many power violations. Taking into account the inability to track the power reference and the high average power indicated that there might be a barrier that prevents the application behavior to rapidly follow changes in the CPU frequency. At this stage, these two benchmarks were suspected to be memory-bound compared to the rest of the workloads that are CPU-bound. In order to verify this hypothesis, we measured the instruction per second (IPS) of all similar high performance workloads in our benchmark set and tailored the two microbenchmarks that stress CPU and memory modules. The average IPS gathered from each workload is reported in Table 5.5. We could clearly observe that IPS for irregular workloads were far less than the other workloads in the SPLASH2 benchmark suite. Memory-bound microbenchmarks exhibited similar behavior with an average power higher than reference power and an IPS less than one half of other benchmarks' average IPS. This experiment validated the theory that the abnormal behavior of the two Ocean benchmarks is due to their high volume of memory accesses which prevents the changes in CPU frequency to have a direct effect on power reference. In order to enable controller to respond better to memory-bound applications, we have increased the order of

the controller three times. Our evaluations show, compared to first order controllers, second, third and fourth order controllers had [-2, 2] percent difference in controller performance which is neither sufficient nor computationally effective. Such cases either require more advanced controllers (e.g., MIMO adaptive, self-tuning) or different/more configuration knobs such as memory bandwidth that can effect the system's power more efficiently.

Workload	Average power (Watts)	IPS
Barnes	6.3289	2.21E+09
Ocean-Contiguous	7.8306	1.07E+09
Ocean Non-contiguous	7.5408	1.36E+09
FMM	6.9943	3.55E+09
Radiosity	6.7472	2.82E+09
Raytrace	6.4023	2.71E+09
Water-NSQ	6.9931	3.14E+09
Water-SP	6.9846	2.97E+09
Volrend	6.7491	3.30E+09
Compute-bound ubench	6.7207	4.18E+09
Memory-bound ubench	7.1668	1.18E+09

Table 5.5: Comparison of average power and IPS

5.2 Gain scheduled controller

In this section, we outline our process for designing gain scheduled nonlinear controllers for a CMP.[1]

In addition, we outline our process for designing gain scheduled nonlinear controllers for a CMP. As a demonstrative case study, we target the ODROID-XU3 platform [68] 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

[1]Details to confirm and formalize popular notions regarding gain scheduled design can be found in [197].

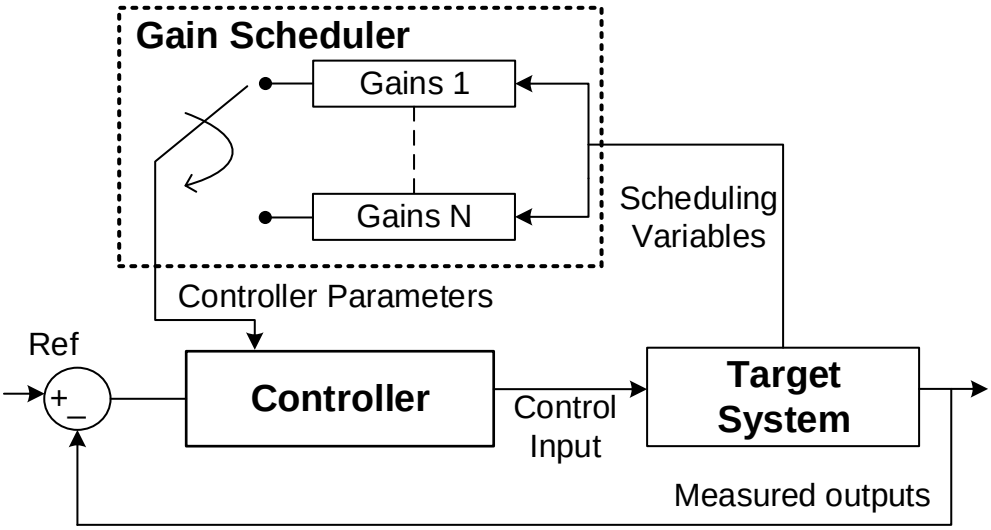


Figure 5.18: Block diagram of GSC.

(due to its linear behavior) and use only the big cores to emulate a uniform nonlinear CMP[2].

5.2.1 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 designed for an operating region is defined by its control parameters K_P and K_I which are stored in the gain scheduler (Figure 5.18). In the gain scheduler, we incorporate logic to determine which gains to provide the controller when invoked.

[2]We refer to this as the Exynos CMP or CMP throughout.

5.2.2 Implementing Gain Scheduling

The gain scheduler enables us to adapt to nonlinear behavior (Figure 5.18) 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.

Algorithm 2 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
```

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 2 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 2 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.

5.2.3 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.[3] We consider a typical mobile scenario in which one or more multi-threaded applications execute concurrently across the CMP.

Controller Configurations: We designed two DVFS controllers for power management of the CMP: 1) **a linear controller** that estimates the transfer function similarly to [75, 146]; and our proposed 2) **GSC**. The GSC contains three operating regions (Table 5.6). 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 whole system. The control input is frequency, and the measured output is power, applied to the entire CMP.

Implementation: 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 $50ms$.

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

[3]Ubuntu 16.04.2 LTS and Linux kernel 3.10.105

	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

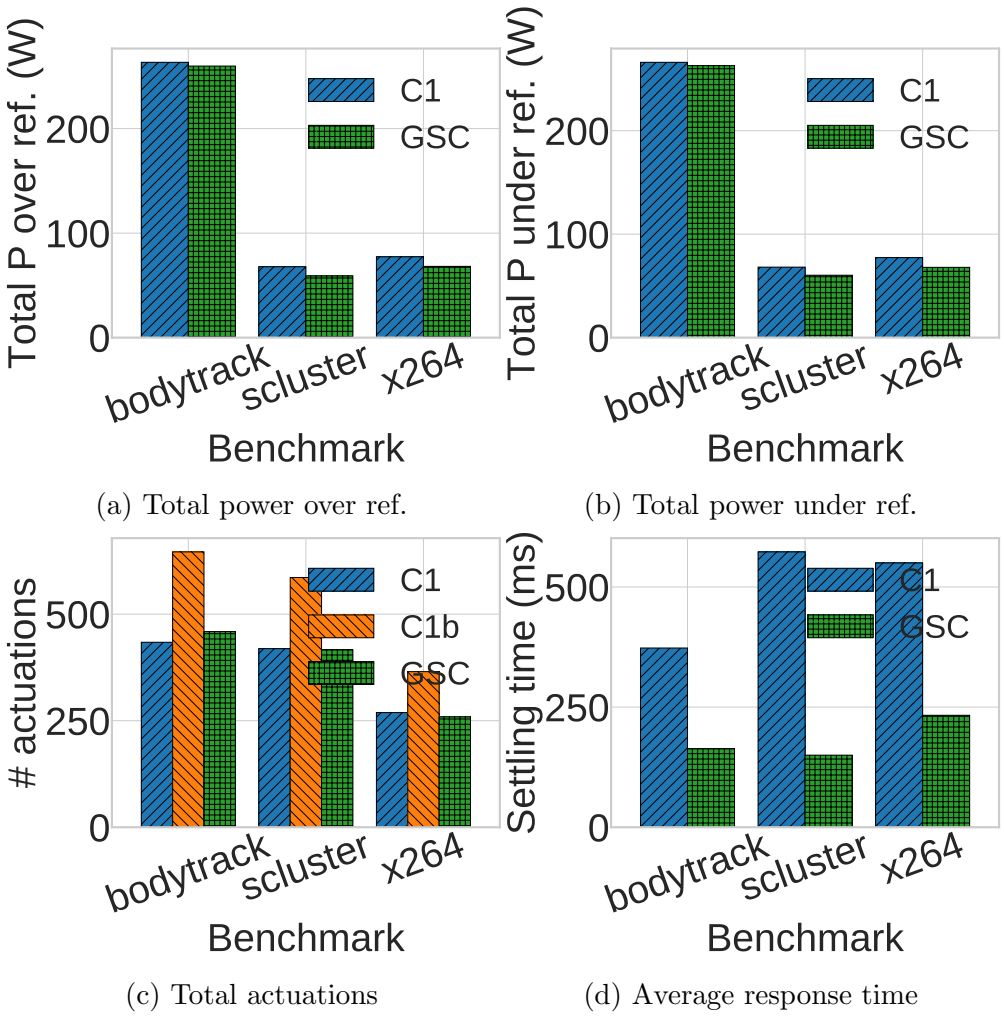
Table 5.6: Accuracy of the full- (Ctrl 1) and sub-range (Ctrl 2.x) controllers.

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 3.5W, 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 0.5W, the lowest reference and represents a reduced budget in response to a thermal event. ref_3 is 1.5W, 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 $65s$. After the first $5s$ (warm-up period) the controllers are set to ref_1 for $20s$, then changed to ref_2 for $20s$, and to ref_3 for the remaining $20s$.

5.2.4 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



(a) Total power over ref.

(b) Total power under ref.

(c) Total actuations

(d) Average response time

Figure 5.19: Comparison of GSC with Controller 1.

2.1-2.3 achieve an order of magnitude better accuracy than Controller 1 (Table 5.6). 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.

5.2.5 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 5.19.

The *power over target* is the total amount of measured power exceeding the reference power throughout execution (Fig. 5.19a). 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. 5.19b). 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. 5.19c). 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. 5.19d). Figure 5.19d 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 2 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.

5.3 HESSLE-FREE a Fuzzy Controller for Heterogeneous Systems

Figure 5.20 shows an overview of our case study. We use the NVIDIA Jetson TX2 development board [34], which contains an HMP and a NVIDIA GPU. HMP contains a quad-core ARM Cortex A57 cluster and a dual-core NVIDIA Denver cluster. Similar to Cortex A57 cores,

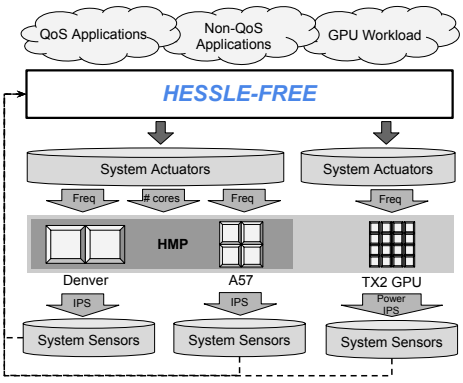


Figure 5.20: HESSLE-FREE experimental setup

Denver cores implement ARMv8 instruction set and are designed as a processor with 7-way superscalar execution pipeline. The GPU is powered by NVIDIA Pascal CUDA cores. We consider multiple scenarios that are common in mobile devices where CPU runs multiple tasks (possibly one foreground with QoS requirements and others in background) and in full system scenarios GPU is executing a highly parallel kernel concurrently.

We use the following two scenarios to demonstrate how HESSLE-FREE can handle different system goals: i) Optimize Energy consumption under dynamic application behavior. Here we execute workloads on CPU cores to demonstrate HESSLE-FREE's ability to dynamically optimize energy. ii) In a full system scenario, the CPU and GPU simultaneously execute their workloads, while HESSLE-FREE optimizes the user metric *frames per second* delivered by the GPU or QoS metric delivered by the CPU, as well as the *power consumption* of the entire system.

5.3.1 Experimental Setup

As described in Section 5.3 we use the JetsonTX2 platform for our evaluations. The controllers used in our experiments are implemented as Linux userspace daemons that execute in the background with the applications. CPU and GPU runtime power are separately measured on-board alongside current and voltage using sensors present on the JetsonTX2 development board. Power measurements are made at the same time increments as performance metrics are gathered. Controller invocation is performed periodically every 200ms. In terms of overhead, the framework runtime on average adds 1.8% to the execution time for accessing PMU registers. A lightweight kernel module is used to collect instruction and cycle counters from ARM's Performance Monitor Unit (PMU) on each CPU. For GPU performance metric measurements, NVIDIA provides CUDA Profiling Tools Interface (CUPTI) library which includes API for attaching callback functions to GPU kernels. The callback functions enable

measuring GPU metrics in application run-time. To avoid modifying target applications, we put necessary CUPTI functions into a shared library which is pre-loaded (LD_PRELOAD) to attach the callbacks when the application begins. This non-intrusive GPU profiling is hooked to our runtime resource management framework to capture the kernel information with low-overhead on the execution of the workload which in average adds 2.2% to GPU kernel executions. This delay is a fraction of imposed overhead by NVIDIA's native profiler (nvprof).

Evaluated Workloads

We use the PARSEC benchmark suite [20] to evaluate the performance of the resource managers. To better represent a real-world scenario where every element of a CPU/GPU system is involved in the computation, we select a face detection algorithm for the GPU workload. we used the implementation in [240] as a standalone application, which is easily portable to an embedded environment, based on the Viola-Jones face detection framework [225] with three GPU kernels for compute-intensive part and some CPU computation for pre- and post-processing of the frame. This application has a frame-per-second (FPS) requirement which can be an objective for the controller. Our target platform is a modern heterogeneous platform that can execute various multi-threaded application simultaneously on HMP while concurrently running massively parallel kernels on the GPU. Depending on user preference and system state on any point of time, priorities of runtime system might change which will update the objective of the resource management mechanism. We will demonstrate three scenarios in Section 5.2.3 to show the ease and efficiency of HESSLE-FREE in adapting to various objectives.

Manager Configurations

HESSLE-FREE provides a framework to efficiently design, implement, deploy and tune fuzzy controllers. This framework comprises of three main components: (1) Design and initial evaluation for the controller, (2) Mapping and optimization of the controllers for portable deployment, (3) Middleware that accommodates the controller and provides APIs for monitoring and actuating configuration settings of the system in various software and hardware layers. In our evaluations, we use Matlab for design and initial test of the resource manager approaches. Fuzzy controllers can be designed using Matlab's fuzzy designer as Mamdani (linguistic) or with neuro fuzzy designer as Sugeno-type controllers with singleton consequents which leads to a simple yet efficient controller with low computational overhead. For defuzzification, we use the centriod method. For each of our experiments with unique objectives, number of membership functions for each input and output range from 3 to 7 and deployed number of rules in the rule base did not exceed one hundred. The design process in HESSLE-FREE starts with the designer defining inputs and outputs of the system and their corresponding number of membership functions. Next, structure of the rule base is defined using linguistic variables. From here, the designer has the option to check the sanity of the inference system, simulate the system based on experimental data gathered from the target platform or use adaptive neuro-fuzzy toolbox to tune rule base and membership functions parameters. Subtractive clustering [192] provided by this toolbox may be used as a rapid one-pass algorithm in this process for estimating the number of rule clusters and the cluster centers in the rule base. The cluster estimates obtained from this function can be used to reduce the size of the rule base and consequently the runtime overhead of the matching and inference process. After initial test and evaluation, each controller is ported into fuzzylite library [176] for an optimized implementation. The resulting controller is integrated with our middleware. We have devised this component to be easily portable to platforms running Linux operating system while providing customizable APIs for accessing system metrics (IPS,

power, utilization, etc.) and changing the configuration knobs (e.g., DVFS settings, number of cores, task mapping, etc.).

In order to design a valid MIMO controller for each scenario, a system model is created using Matlab System Identification Toolbox [141] by generating test waveforms from training applications. A common practice to build a model for complex systems is to use black-box methods based on System Identification Theory [125] for isolating the deterministic and stochastic components. Then, these controllers are tuned using Matlab's Simulink toolbox. Afterwards, they are deployed on the target system for experiments using test workloads. A detailed report on MIMO design can be found in [171]. Considering the large design space of configuration settings and impacting factors in our evaluations that involve both CPU and GPU applications, system identification requires fine tuning and repeated evaluations to ensure the extracted model can appropriately reflect the target platform.

MIMO's complexity: To depict a picture regarding exponential growth of MIMO controllers runtime complexity we use the number of required matrix multiplications depending on *order* and *size* of the system. For a simple second order MIMO controller with two inputs and outputs approximately 300 operations are needed. Increase in order of the same controller to 4th and 8th order controller will result in 1500 and 10000 operations. If instead of the order, size of the system grows to 4 inputs and outputs, number of required runtime operations will reach 15000. For a bigger 8x8 system this number exceeds 200000 operations per decision making. In some cases, runtime overhead of MIMO controllers for such large complex systems can put a burden on the runtime framework. NVIDIA Performance modes and resource management governs are pre-loaded into the operating system that is ported specifically for jetson platform. Based on the target evaluation some of which are used in the appropriate scenario experiments.

5.3.2 Evaluation Results

We now demonstrate the advantages of fuzzy control in runtime resource management with optimization objectives along side system goals that might require certain guarantees. Our goal is to evaluate our HESSLE-FREE with respect to the state-of-the-art control theory controllers in terms of both ability to capture the system's dynamics and achieve the system's objectives. In order to make a fair comparison, we start with a simple multi-core system and work our way towards a more complex heterogeneous system. In this manner we are able to highlight the challenges faced by MIMO controller design for complex system dynamics, while showing the ease of using HESSLE-FREE. In addition, the efficacy of HESSLE-FREE is compared against a correspondent MIMO controller and state-of-the-art algorithms towards a certain objective. We show the capability of HESSLE-FREE towards achieving various system's objectives in order to demonstrate ease in design and flexibility of fuzzy controllers.

Uniform Multi-core

In this part, we demonstrate the inherent optimization of fuzzy controller compared to conventional controllers by evaluating the efficacy of our approach for system energy minimization of a quad-core CPU. We use million instructions per second (MIPS) per Watt to represent energy consumption of this ARM A57 multi-core at runtime. MIMO has shown to be effective in tracking power and performance references for such systems with low-level of heterogeneity [157]. However, when the optimization happens at run-time in reaction to the dynamic behavior of the application, traditional MIMO faces challenges. Fuzzy control enables the embedding of optimization algorithms inside the control mechanism, which in turn naturally allows the system to react to this dynamic behavior.

Figure 5.21 shows the normalized MIPS per Watt for CPU workloads. We evaluate HESSLE-FREE in comparison to a MIMO controller and a NVIDIA performance model designed for

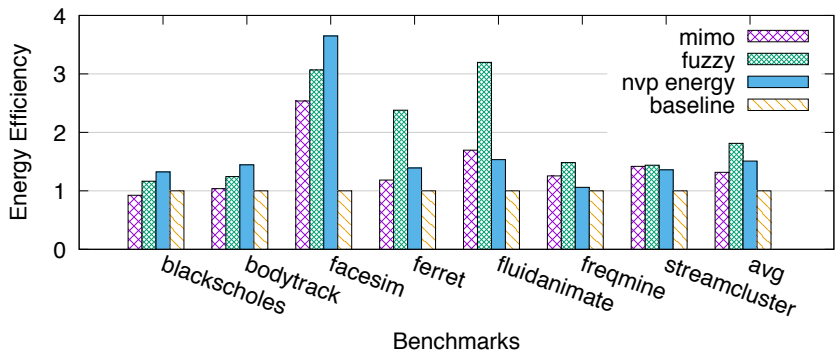


Figure 5.21: MIPS per Watt for CPU workloads. This value is normalized to default linux values.

energy efficient execution on Jetson TX2 platform. The actuations in the system are the number of active cores and core frequency. The intuition behind the designer's expertise used in design of this fuzzy controller is to adjust the computation power of the system to the dynamic behavior of the application. This will allow the fuzzy controller to increase the frequency and active cores as long as it adds to performance of the system in a meaningful manner and reduce the computation power to avoid energy waste while the performance is bounded either by memory access or disturbance of background applications. Alteration of frequency settings has a much smaller threshold compared to change in number of active cores. This threshold is extracted through experimental evaluations done in the initial phase of the fuzzy controller design. Overall, HESSLE-FREE demonstrates efficiency in managing the system's objectives. On average, fuzzy controller shows improvements of 81.3% over the Linux governor, 37.7% over the MIMO controller, and 20.0% over NVIDIA's state-of-the-art energy efficient governor.

CPU-GPU Resource Management

To evaluate the efficacy of HESSLE-FREE with respect to MIMO solutions, we perform two experiments on a full system exercising both the CPU clusters (executing PARSEC benchmarks) and the GPU (executing the Face detection application). We perform our full system evaluations comparing HESSLE-FREE with MIMO controller and a variety

of Linux governors (*interactive*, *ondemand*, *performance oriented*, *power saving*). Each of these governors targets a certain objective in the system like maximizing performance or minimizing power consumption. For each of the experiments we perform system identification and control design process for MIMO controller. Target references are obtained through Sensors and actuators are fixed for each experiment. MATLAB System Identification toolbox also recommends a suitable order for the system. If possible, we pick the order with the best accuracy. However, the *order* of a controller model determines how observed output history is stored in the model, and directly impacts both the controller size and complexity. We begin with smaller size of the system (2 inputs x 2 outputs) where efficient MIMO controller is feasible and show the decrease in system identification accuracy and controller design efficiency when the number of the inputs (e.g, frequency knobs) and target outputs (e.g, QoS metric, power consumption) grows. During our full system experiments, We evaluated that for higher numbers of systems inputs (e.g, frequency and number of active cores in each cluster) and outputs (e.g, simultaneous FPS and QoS metrics with each units power consumption) with heterogeneous compute units the complexity of the system grows to the point that the system dynamics cannot be captured in the system identification phase with acceptable accuracy and also Matlab can only provide very large orders for MIMO controllers that cannot be computed at runtime. However, HESSLE-FREE without the need for system model was able to achieve system goals using expertise knowledge and tuning. Here, we report on the cases that MIMO control design was possible.

QoS focused

In this experiment, we use a QoS metric to evaluate the progress of the CPU foreground application. This is done by the Heartbeats API [53] 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. Figure 5.22 shows tracking

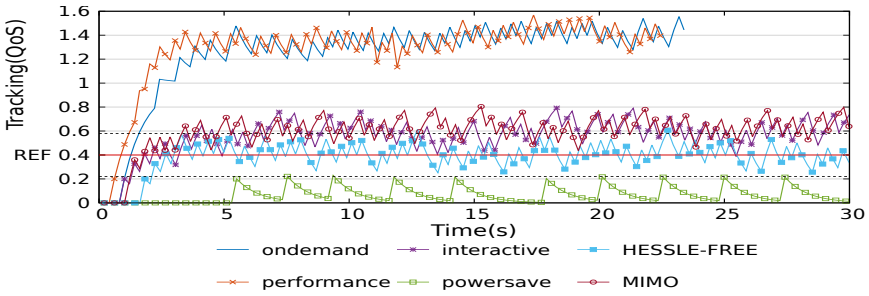


Figure 5.22: Tracking QoS metric (ref = 0.4) for fluidanimate benchmark with different resource managers

of QoS value by different resource management mechanisms for fluidanimate benchmark and target reference of 0.4. In this scenario, one foreground QoS application is running while there are many non-QoS applications are running in the background both on CPU and GPU. The goal of the resource manager is to keep the heart beats (QoS metric) in the specified range (0.2 - 0.6) by the application while consuming minimum power. Because of the heterogeneity in the CPU clusters MIMO controller has a hard time following the target reference while fuzzy controller is able to meet the QoS in a steady manner using minimum energy compared to other resource managers. Expertise used in this experiment for fuzzy controller was to not only consider the error from reference QoS but also the speed of change in measured Heartbeats. Meanwhile, as we reach and pass the target reference, we reduce the frequency of the compute unit incrementally to the extent that QoS drops to half point of target reference and lower boundary. Moving forward, CPU frequency is increased to the point that we exceed target reference again. Figure 5.23 shows the comparison of energy consumption of each resource manager. HESSLE-FREE achieves and tracks the QoS metric while being in average 26%, 46%, 43%, 66% and 65% more energy efficient than MIMO controller, interactive ondemand, performance and power oriented governors, respectively. This is due to the designer expertise incorporated in the fuzzy controllers that favors energy efficient cores actuations and only increases frequency for high-performance cores when higher QoS is needed.

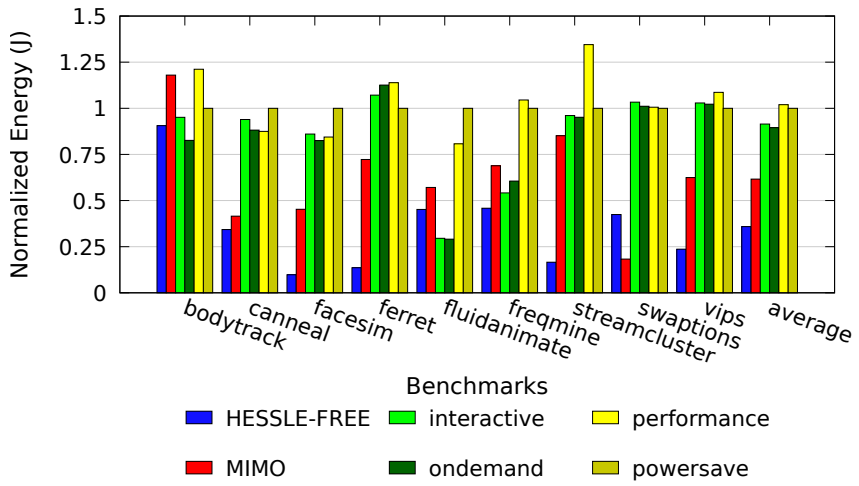


Figure 5.23: Total Energy consumption for CPU+GPU for tracking QoS metric (normalized to power saver energy)

FPS focused

In this experiment our target metric is to meet our desired FPS while consuming the minimum power required. Target FPS is defined as 30 frames per second for our platform with threshold of ±5 frames. We capture the number of frames processed in each measuring window. In order to stress test the management policy, we execute the PARSEC benchmarks on the CPU cluster in parallel with the face detection GPU workloads. The rationale for this mix is that the CPU workloads can demonstrate dynamic phasic behavior (e.g., compute-bound, memory-bound) that can affect the GPU performance and consequently the FPS of the system. Figure 5.24 shows the FPS tracking for each resource manager for the facesim CPU benchmark. The rest of workloads follow a similar trend where: i) the performance oriented governor provides high FPS with no regard to power consumption, ii) the ondemand and interactive governor provides moderate performance based on application demand which sometimes results in high power consumption of CPU units, iii) the power saving governor executes in the lowest configuration of each compute unit without any regard to the system status, iv) MIMO generally tracks FPS but abrupt changes to power consumption can cause deviation from the target reference, and v) our HESSLE-FREE fuzzy controller is able to

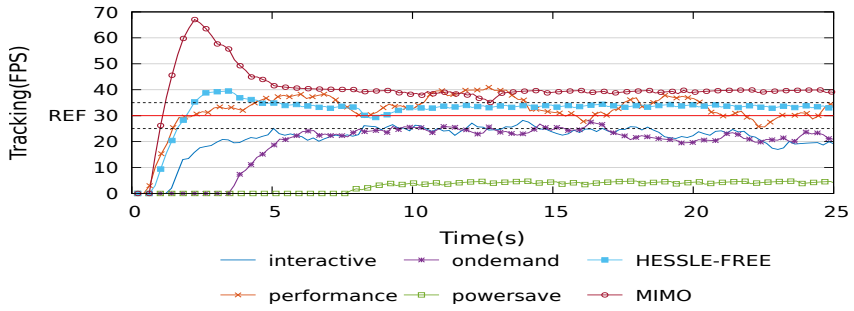


Figure 5.24: Delivered FPS for Face detection

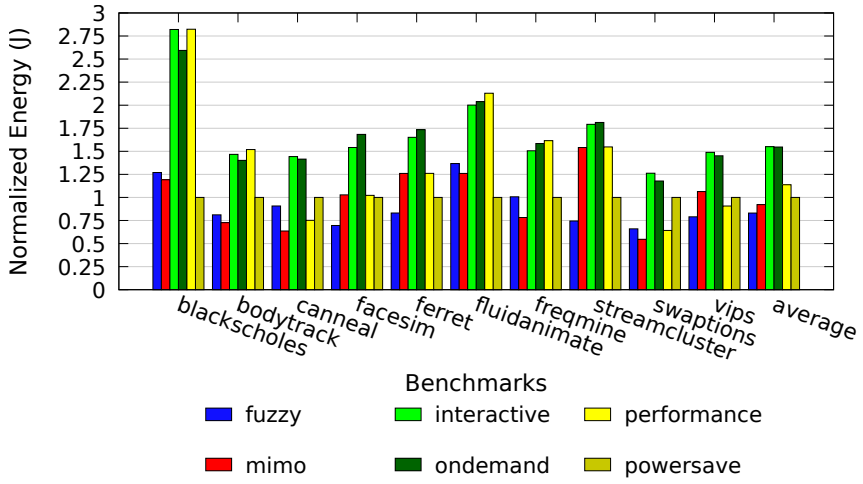


Figure 5.25: Total Energy consumption for CPU (PARSEC) plus GPU (face detection) for FPS metric

follow the reference FPS while minimizing energy consumption. The intuition behind fuzzy rules is to observe the error in FPS and take actions according to value of this error to set the configuration knobs. As the error gets closer to the target FPS, GPU frequency change slows down. Meanwhile, we try to minimize the energy consumption of the entire platform by reducing the energy consumption of the CPU cores while avoiding any drops in FPS measurement. Also, In the case that increase in GPU frequency shows no improvement in a consecutive windows, we increase the CPU cores frequency. This is done to reduce the frame pre-processing bottleneck. Figure 5.25 shows the energy consumption of CPU clusters executing PARSEC benchmarks plus GPU cores executing face detection through this experiment. HESSLE-FREE's fuzzy governor is able to achieve the desired objective tracking the system's FPS while in average preserving 9% more energy than MIMO controllers.

5.4 SPECTR: On-chip Resource Management

We present SPECTR's supervisory control architecture (Section 5.4.1), describe an experimental case study demonstrating the design and verification of SPECTR on the Exynos HMP platform (Section 5.4.2), and outline SPECTR's control synthesis process (Section 5.4.3).

5.4.1 Hierarchical System Architecture

Figure 5.26 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 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 [188][74][153]) 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.

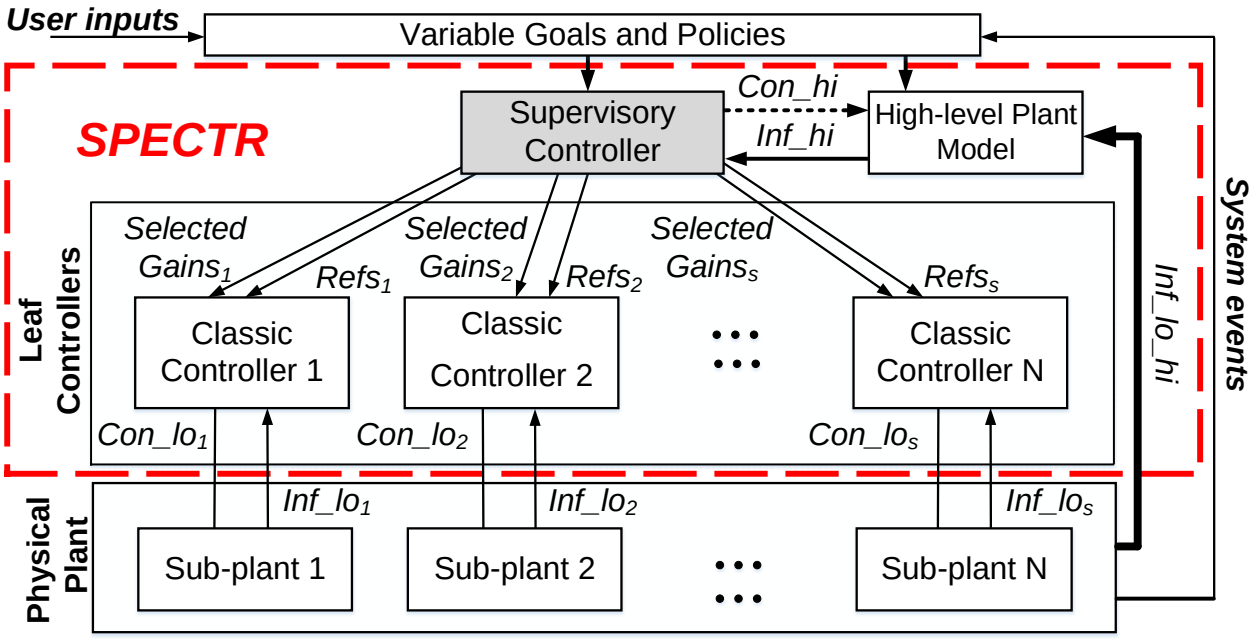


Figure 5.26: SPECTR overview.

5.4.2 Experimental Case Study

Figure 5.27 shows an overview of our experimental setup. We target the Exynos platform [68], 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 application*) 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).

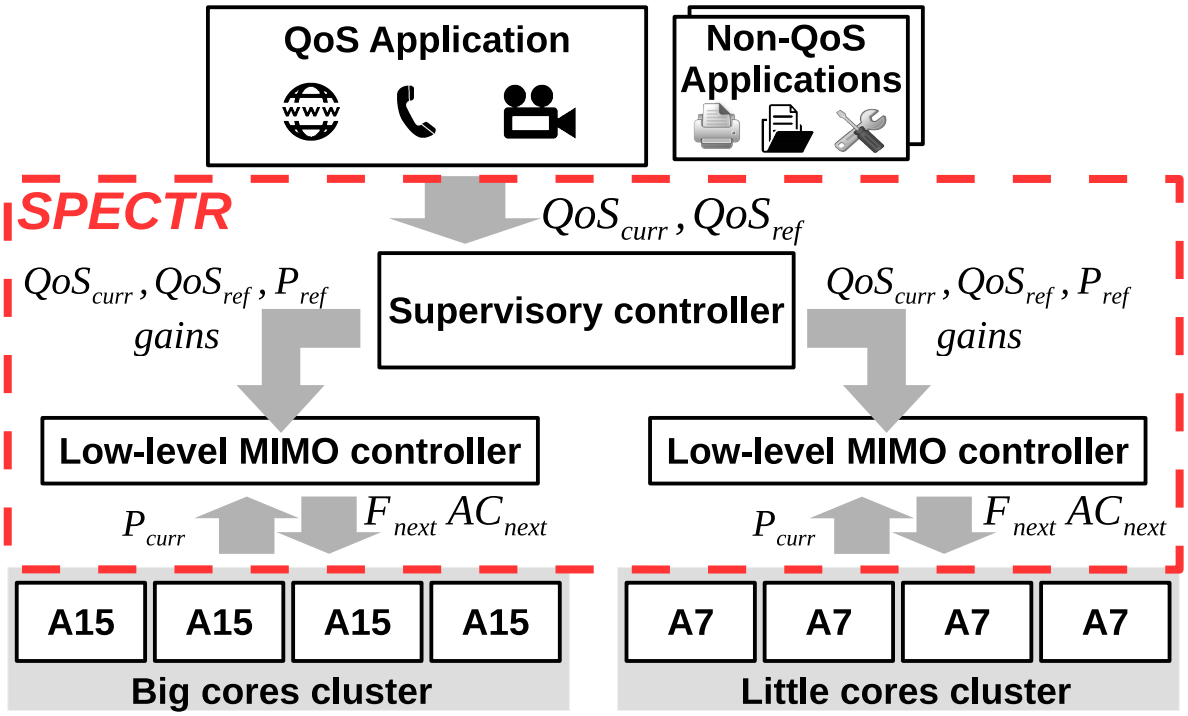


Figure 5.27: SPECTR implementation on the Exynos HMP with two heterogeneous quad-core clusters. Representing a typical mobile scenario with a single foreground application running concurrently with many background applications.

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}).[4]

Supervisory control commands guide the **low-level MIMO controllers** in Figure 5.27 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

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

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.

5.4.3 Supervisor Synthesis Process

The **supervisory controller** is responsible for coordinating the low-level controllers shown in Figure 5.27. The supervisory control synthesis, illustrated on Figure 5.28, follows five steps [16]:

1. Develop high-level *Plant Model* (P) as a discrete-event dynamic system.
2. Develop *Intended Behavior Specification* of the plant (S_P) (i.e., desired control behavior).
3. Perform *Synthesis* of the Supervisor (S) from the plant model and behavioral specifications.
4. Perform *Nonblocking Property Checks* to remove any logical/blocking conflicts.
5. Perform *Controllability Property Checks* to ensure that the supervisor meets controllability properties.

In this part, we discuss each step of modeling, specification, synthesis and verification of the supervisory controller. All steps are automated by the Supremica SCT tool-set [3]. For ease of visualization, we show the automaton generated by Supremica in each step. We integrate the two goals described in Section 5.4.2 for the system in Figure 5.27. We ensure autonomy

of the system to meet the QoS requirements while the total power remains within the defined boundaries by using gain scheduling.

Plant Model

Any physical plant G can be described using an infinite number of attributes, while the plant model P can capture only a finite number of attributes. Therefore, we begin by capturing the platform's most relevant characteristics (*power consumption* and *QoS* in our study) to build a plant model. Given the formal underpinnings of SCT, we exploit automata theory [76] to automatically generate the plant model from simpler models of its constituent subsystems (i.e., sub-plants).

Now, consider an automaton A defined as a 5-tuple
$A = \langle Q_A, \Sigma_A, \delta_A, i_A, M_A \rangle$, where Q_A is the set of states, Σ_A is the set of events consumed by A, $\delta_A : Q_A \times \Sigma_A \rightarrow Q_A$ is the state transition function, i_A is the initial state and M_A is the set of final states. The *synchronous composition* of two automata A and B, A||B, is then defined as [139]:

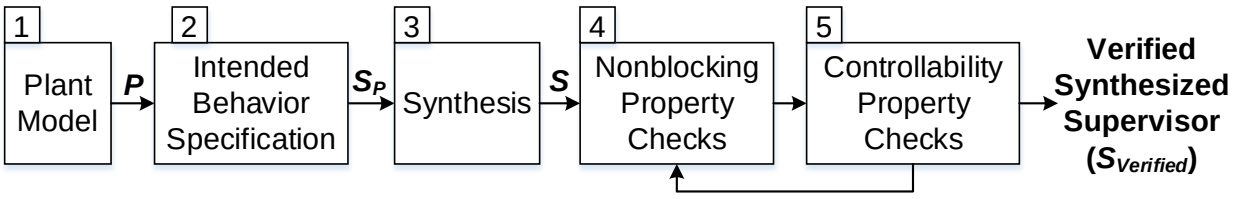
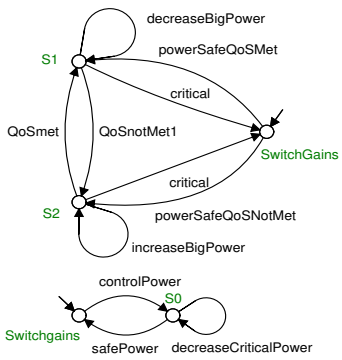
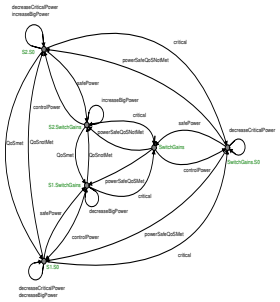


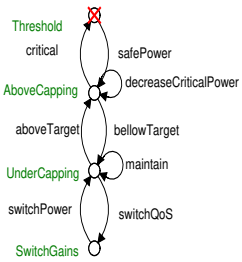
Figure 5.28: Synthesis process for a Supervisory Controller



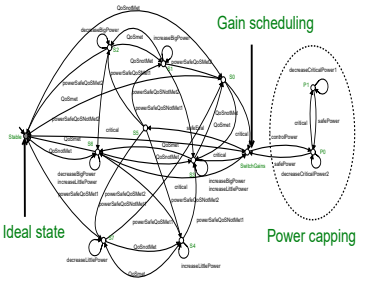
(a) Models for Big cluster: Top: QoS management, Bottom: Power Capping.



(b) Automatically generated Big cluster plant P using the || operator on two models described in Fig. 5.29a.



(c) Example of intended behavior Specification S_P. Red cross indicates forbidden state.



(d) Synthesized supervisor S generated from plant model and specification using the Supremica tool [3]. Checked for non-blocking and controllability properties.

Figure 5.29: Supervisor Synthesis Process. Figures 5.29b and 5.29d are automatically generated by the SCT tool, and the state details are *not* important.

$$A \parallel B = \langle Q_A \times Q_B, \Sigma_A \cup \Sigma_B, \delta, i_A \cdot i_B, M_A \times M_B \rangle, \text{ with}$$

$$Q_A \times Q_B = \{q_A \cdot q_B | q_A \in Q_a, q_B \in Q_B\}$$

$$\delta(q_A \cdot q_B, e) = \begin{cases} \delta_A(q_A, e) \cdot \delta_B(q_B, e) & \text{if } \delta_A(q_A, e) \text{ and } \delta_B(q_B, e) \text{defined} \\ \delta_A(q_A, e) \cdot q_B & \text{if} \delta_A(q_A, e) \text{ defined and } e \notin \Sigma_B \\ q_A \cdot \delta_B(q_B, e) & \text{if } e \notin \Sigma_A \text{ and } \delta_B(q_B, e) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Synchronous composition (operator ||) synchronizes the operations of two automata such that common events are synchronized but private events are **not** affected by the other automaton. This preserves the main characteristics of each automaton while including their interactions that affect the whole plant.

Figure 5.29a shows two simple examples for the Big cluster automata (examplifying two of many possible ways to define our systems control solution). In states *S1* and *S2* of the top automaton, we prioritize QoS: the power reference is updated to meet the QoS reference in a power-efficient manner. Upon detection of a power budget violation, a *critical* signal is

generated. The signal results in a transition to the *SwitchGains* state where power-driven gains replace the performance-driven gains. This updates the low-level controller's priority objective from QoS to power. The supervisor also has the opportunity to enforce a reduced power reference have depending on the severity of the situation (*S0* in bottom automaton of Figure 5.29a). Once the power of the Big cluster returns to a safe region, gains are switched back to prioritize QoS. We can make suitable plant models in a similar manner for the Little cluster and its interaction with the whole system. Figure 5.29b shows the synchronous composition of the two Big cluster plant models and specifies all possible interactions for these two automata. In this model, all states are accessible and all events are accepted.[5] However, such complete freedom might not be desirable for the system. We now describe the *specification* that restricts this model to fit the intended behaviour of the system.

Intended Behavior Specification

While the plant model sets the physical boundaries for *all* possible actions, the specification defines the accepted (i.e., ideal) and forbidden states through *restrictions* on the behavior of the plant model. These restrictions are then transformed into a formal description for the synthesis process.

Figure 5.29c shows a sample specification for the Big cluster in our case study. The plant model shown in Figure 5.29b has no limitations on exceeding the power budget; our specification prevents exceeding the power budget for no more than three control intervals (i.e., *Threshold* state is a forbidden state[6]). Similarly, we can limit the chip power consumption using a specification that restricts the sum of the power budgets of both clusters to be below a safe threshold defined by thermal design power (TDP). In our case study, we use a three-band (i.e., uncapping threshold, capping target and above capping threshold) algorithm similar to

[5]Accepted states are shown with solid dark circles.
[6]A red cross identifies a forbidden state.

[54] for making power capping decisions. While we are below the first threshold (uncapping threshold), controllers focus on meeting their QoS requirements. When we exceed this threshold, gain scheduling ensures that we remain in the capping target region.

Synthesis

Once we have a plant model and a formal specification of intended behavior, a synthesis algorithm is guaranteed to generate a correct controller [55]. Hence, a correct plant P and specification S_P are crucial to synthesize a supervisory controller S such that the closed-loop system fulfills the specification S_P. Figure 5.29d shows an example supervisor that was automatically synthesized for the Exynos platform using the Supremica tool, given as input the plant model and the intended behavioral specification capturing desired outcomes and restricting undesired behavior (e.g., Figure 5.29c). Note that the models built for plant P and the intended behavior specification S_P are design artifacts, and only the final synthesized and verified supervisor $S_{verified}$ is implemented in the system. We now describe the verification of additional properties for ensuring correctness of the entire supervisory controller.

Non-blocking and Controllability Property Checks

We must ensure that the synthesized supervisor is both **non-blocking** and **controllable**. The non-blocking property guarantees that some accepted states (e.g., the ideal state shown in Figure 5.29d) can *always* be reached, so that at least one of the tasks can always be completed. On the other hand, the controllability property guarantees that the supervisor can always keep the plant within the boundaries set by the specification. In our example, there is one accepted (i.e., ideal) state that satisfies the QoS requirement while maintaining the power consumption under the limit. The SCT tool ensures that in the generated supervisor (Figure 5.29d) there is a path to this accepted state from *every* other *valid* state. In addition,

the plant model is pruned by the specification to make it adhere to desired behavior. The closed-loop system will *never* reach a state such that an uncontrollable event causes it to violate the specification. These two properties are provided by two different algorithms: the trimming algorithm [55] provides the non-blocking property, and the extension algorithm [74] provides the controllability property. However, these two algorithms interfere with each other, with trimming possibly impairing controllability, and vice versa. Therefore, the two algorithms must be run successively and iteratively, until they return the same result.

Uncontrollable states. The search for the largest controllable sub-automaton of the specification begins with identifying the uncontrollable states. Subsequently, any state that reaches an uncontrollable state via an uncontrollable event is identified. This forms the basis for the algorithms that construct a controller given a specification and a plant.

Non-blocking. The supervisory controller is non-blocking if the closed-loop system is always able to reach some **marked** state (i.e., *Ideal* state shown in Figure 5.29d). In order to find a lean non-blocking supervisor, we must find the set of accessible states. It is desirable to find the largest possible sub-automaton that has this property.

5.4.4 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 [172], 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 [242], 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 [141].[7] 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

[7]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 [172].

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.

Experimental setup. We perform our evaluations on the ARM big.LITTLE [5] based Exynos SoC (ODROID-XU3 board [68]) as described in our case study (Figure 5.27). We implement a Linux userspace daemon process that invokes the low-level controllers every $50ms$. When evaluating SPECTR, the daemon invokes the supervisor every $100ms$. 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 [53] 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 [17] 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 $3.2X$ (`streamcluster`) to $4.5X$ (`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.

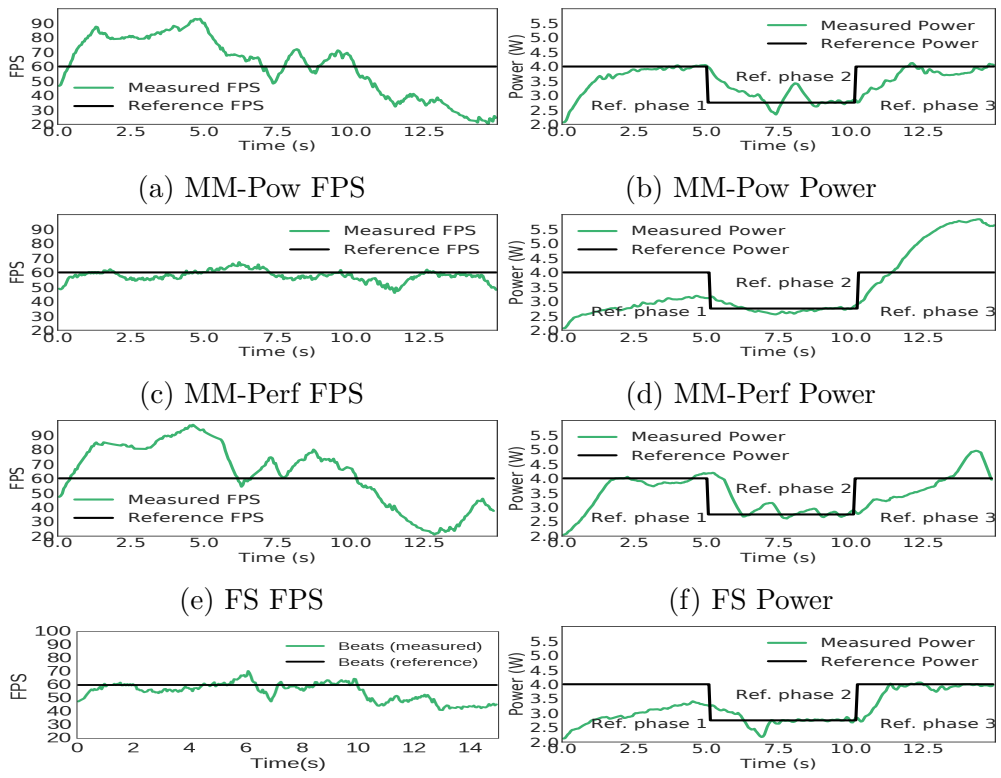
5.4.5 Comparison of Resource Managers

For brevity, 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 5.30 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 seconds 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 5.31a and 5.31b 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 [71]. 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. 5.31b) while maintaining FPS within 10% (Fig. 5.31a) 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



(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 5.30: Measured FPS and Power of all four resource managers for three Phases of 5 seconds each, for the `x264` benchmark.

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-second period of execution in Figure 5.30. 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 5.30f) 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 of feedback control systems [71]. *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 seconds, while SPECTR has an average settling time of 1.28 seconds. The larger size of the state-space ($x(t)$ matrix in Equation 3.2 and 3.3) 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

(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 5.31: 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).

slow settling time of FS. This is also the reason why SISO controllers are generally faster that MIMOs [71].

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 5.30. 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 5.31e and 5.31f. 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. 5.31e), while obeying the TDP limit (Fig. 5.31f). 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. 5.31e) while exceeding the TDP by more than 30% (Fig. 5.31f), 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 5.31a, 5.31c, and 5.31e show the average steady-state error (%) of QoS for Phases 1, 2, and 3 respectively. Figures 5.31b, 5.31d, and 5.31f 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 5.31a and 5.31b). `canneal` follows the same pattern with respect to power as all other benchmarks (Fig. 5.31b). `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. 5.31a). 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

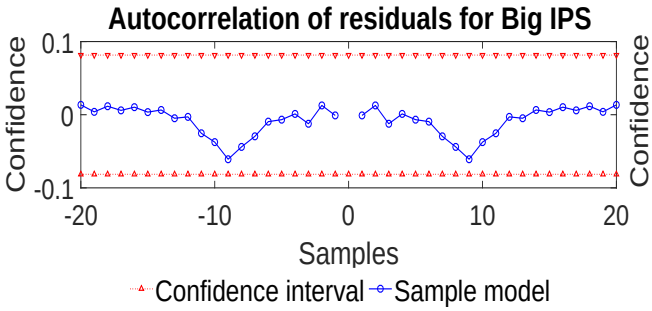
error (Fig. 5.31d). The only exceptions are `canneal` and `k-means`: the MM-Perf manager is unable to react to change in TDP for `canneal` and `k-means`. The MM-Perf manager lacks a supervisory coordinator and prioritizes performance, and was unable to find a configuration for `canneal` and `k-means` that satisfied the QoS reference value within TDP. In the Workload Disturbance Phase, SPECTR, FS, and MM-Pow all achieve near-reference power (Fig. 5.31f). As expected, MM-Perf violates the TDP in all cases, but always achieves the highest QoS (Fig. 5.31e).

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.

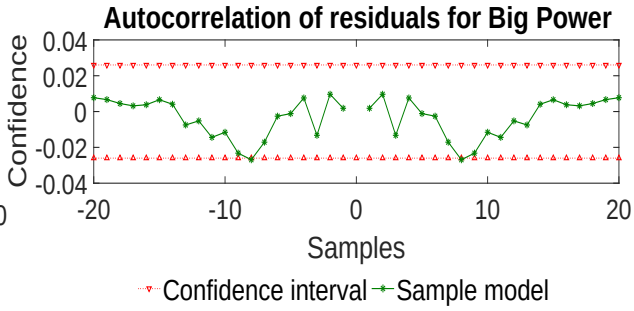
5.4.6 Scalability Evaluation

To evaluate the scalability of SPECTR with respect to single or nested MIMO solutions, we compare the identified models for controlled systems of different sizes. After an estimated system dynamics is produced using system identification techniques, it is cross-validated using different data sets. The common practice is to assess the model by analyzing *residual* auto-correlation [171]. Residual is the stochastic component (e.g., disturbance, noise, etc.) of the system output, which is not supposed to be included in the model. When validating the model, the model output is compared to noisy system outputs. Therefore we expect the residual to be pure noise. To verify this, the residual is analyzed for correlation. If there is *no* correlation between the residual and itself or any inputs, the model is accurate enough. *Confidence* can be used to specify a range. A confidence level is the probability with which the true output will fall into a range called a *confidence interval*. The confidence interval provides a range of values that is likely to contain the population parameter of interest [161].

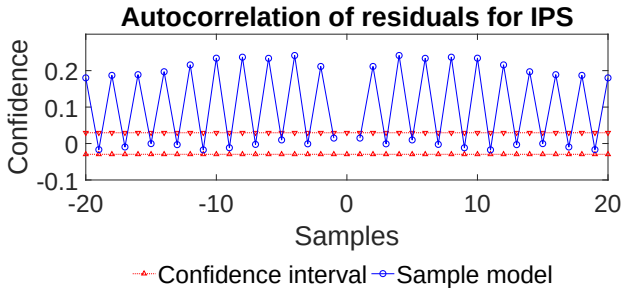
A confidence level of 99% results in a confidence interval that spans three standard deviations. In our case, a higher confidence level means more confidence in where the true output will lie, and a model output within the confidence interval indicates that the deterministic component of the model output will be near the true output.



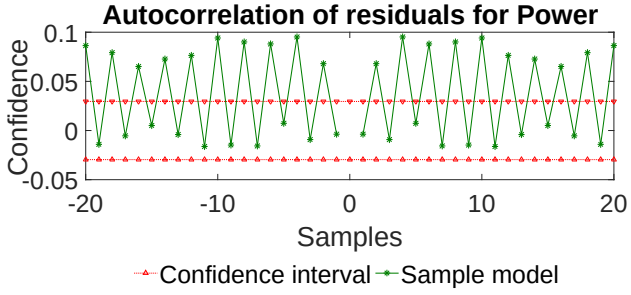
(a) 2×2 system model for the Big cluster controller of SPECTR, total IPS output.



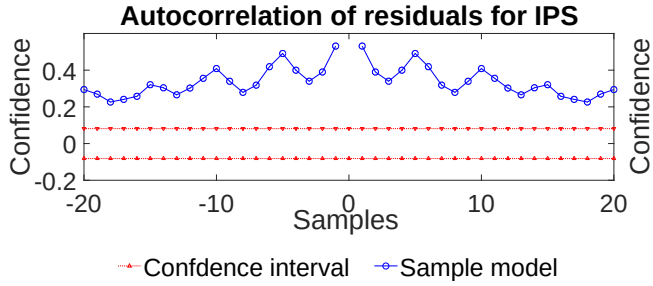
(b) 2×2 system model for the Big cluster controller of SPECTR, total power output.



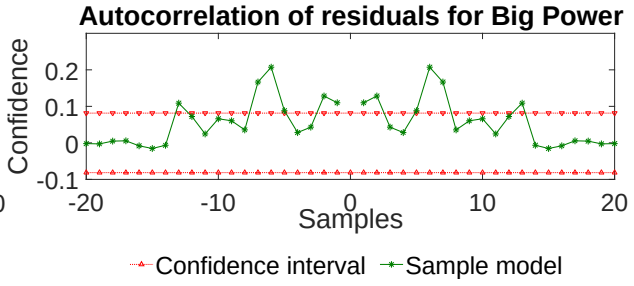
(c) 4×2 system model for the FS controller, total IPS output.



(d) 4×2 system model for the FS controller, total power output.



(e) 10×10 system model for a large-system controller , single-core IPS output.



(f) 10×10 system model for a large-system controller, Big cluster power output.

Figure 5.32: Autocorrelation of residuals for identified system models of different sized MIMO controllers. We show a single performance and power output for each modeled system across multiple sample inputs.

Figure 5.32 compares the autocorrelation of residuals for instructions per second (IPS) and power of three systems: 1) 2×2 Big cluster MIMO used in SPECTR, 2) 4×2 MM-Pow, and 3) 10×10 controller that represents a large system. The 2×2 controller for the Little cluster shows similar behavior to the 2×2 controller for the Big cluster in Figure 5.32a. MM-Perf controller shows similar behavior to MM-Pow in Figure 5.32c.

The two main properties desired while checking the autocorrelation of residuals are for the controller to: 1) stay inside the confidence interval, and 2) avoid sharp peaks and drops. While the 2×2 controller stays within the confidence interval for IPS and power (Figure 5.32a,5.32b), the 4×2 controller exhibits sharp peaks that *violate* the confidence interval for multiple sample inputs (Figure 5.32c,5.32d). The controller for the large 10×10 system has difficulty staying within the confidence interval, especially for IPS (Figure 5.32e,5.32f). Controllers for large MIMO systems with more complex behavior are not only slower in terms of settling time, but also often infeasible to design due to the lack of a sufficiently accurate system dynamics model.

We conclude that SPECTR supports scalability for resource management that classical controllers do not. Classical controllers cannot accurately model large systems. SPECTR solves this issue by deploying many simple controllers for decomposed subsystems, and coordinating them with a high-level supervisor.

5.4.7 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 $2.5ms$, on average, over 30 seconds. The MIMO controller is invoked every $50ms$ resulting in a 5% overhead, which is experienced by all evaluated controllers. We measure the runtime of the supervisor to be $30\mu s$, 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[8] 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.

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

Chapter 6

Conclusions and Future Directions

This dissertation addresses runtime policies for dynamic resource management in heterogeneous systems. First, we proposed a classification for dynamic resource management based on allocation choices and control decisions. We surveyed heuristics, machine learning and control theoretic methods used in tuning architectural parameters in computer systems.

Our early work demonstrates using SISO for power capping presents an initial case study for using control theory managers in order to manage runtime metrics in computer systems and pitfalls in regards to abrupt changes in application dynamics. Based on these findings we focus on adding adaptive policies that can respond to system dynamics in a rapid and response fashion.

The HESSLE-FREE project presents a novel method for benefiting from Fuzzy Control Theory in simplifying complex system identification of heterogeneous systems and increasing the efficiency of runtime management mechanism by utilizing designer's expertise. HESSLE-FREE leverages fuzzy control theory to combine heuristic approaches with the strengths of classic control theory to efficiently manage complex heterogeneous systems with a variety of system objectives.

In the SPECTR project, we develop a hierarchical supervisory control mechanism for resource management in heterogeneous many-core systems. SPECTR combines the strengths of classic control theory with state-of-the-art heuristic approaches to efficiently manage complex systems with multiple goals in a hierarchical manner. SPECTR leverages formal Supervisory Control Theoretic techniques, such as gain scheduling, to achieve autonomy for individual distributed controllers and scalability for the entire system, while satisfying higher-level system goals.

6.1 Directions for Future Work

Contributions of this thesis can only cover a certain area in resource management and can be further expanded by future work. Some can be immediate extensions to this dissertation that are easily reachable and some may require additional effort for future researchers. Hopefully, following ideas will give some insight to next generation of PhD students.

6.1.1 Immediate extensions of this dissertation

In order to highlight the possible challenges for emerging systems, a representative real world application is required that has interaction with majority of the system components. To this effort an autonomous driving framework is under implementation. This framework includes object detection and lane detection. Further analysis of power, temperature, performance (end to end delay) and QoS (frame per second) and possible optimizations may be required. Additionally, more software components such as localization, steering and control modules can be added into this framework.

6.1.2 Novel work that could build on top of this thesis

With advancements in modern systems, using online machine learning methods alongside other workloads has become possible. Therefore, exploring hybrid management using machine learning and control theory is now a promising avenue. Namely, neuro-fuzzy can facilitate parameter tuning in fuzzy management that can enable online adaptation of fuzzy management or even fuzzy supervisors. In addition, SCT can be proven useful managing light-weight online learning methods such as Q-learning.

Finally, future efforts can be focused towards expanding the reach of management mechanism by adding sensing/actuation mechanisms in components such as domain specific accelerators and memory subsystem. This may require novel resource management methods that involve understanding of the behaviour of the system based on memory access patterns and optimization of the overall system.

Bibliography

[1] A. Acquaviva, L. Benini, and B. Riccó. Energy characterization of embedded real-time operating systems. *SIGARCH Comput. Archit. News*, pages 13–18, 2001.

[2] A. H. Ajami, K. Banerjee, and M. Pedram. Analysis of substrate thermal gradient effects on optimal buffer insertion. In *ICCAD*, ICCAD '01, 2001.

[3] K. Akesson, M. Fabian, H. Flordal, and R. Malik. Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. In *International Workshop on Discrete Event Systems*, 2006.

[4] J. Aracil and F. Gordillo. *Stability Issues in Fuzzy Control*. Studies in Fuzziness and Soft Computing. Physica-Verlag HD, 2000.

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

[6] K. J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley, 1995.

[7] K. M. Attia, M. A. El-Hosseini, and H. A. Ali. Dynamic power management techniques in multi-core architectures: A survey study. *Ain Shams Engineering Journal*, pages 445–456, 2017.

[8] R. Ayoub, K. R. Indukuri, and T. S. Rosing. Energy efficient proactive thermal management in memory subsystem. In *ISLPED*, 2010.

[9] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 168–175, 2002.

[10] W. Baek and T. M. Chilimbi. Green : A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. pages 198–209, 2010.

[11] A. Baldassari, C. Bolchini, and A. Miele. A dynamic reliability management framework for heterogeneous multicore systems. In *Proc. of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2017.

[12] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers*, pages 1454–1469, 2003.

[13] N. Beckmann, P.-A. Tsai, and D. Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 538–550. IEEE, 2015.

[14] A. Beloglazov and R. Buyya. Energy efficient resource management in virtualized cloud data centers. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831, 2010.

[15] L. Benini, A. Bogliolo, G. A. Paleologo, and G. D. Micheli. Policy optimization for dynamic power management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1999.

[16] 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.*

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

[18] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. pages 47 – 56, 10 2008.

[19] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct 2008.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite. In *PACT*, page 72, New York, New York, USA, 2008. ACM Press.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[22] R. Bitirgen and et al. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO*, 2008.

[23] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[24] P. Bogdan, R. Marculescu, and S. Jain. Dynamic power management for multidomain system-on-chip platforms: An optimal control approach. *ACM Trans. Des. Autom. Electron. Syst.*, pages 46:1–46:20, 2013.

[25] C. Bolchini, M. Carminati, and A. Miele. Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing*, 29(2):159–175, Apr 2013.

[26] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *HPCA*, 2001.

[27] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang. Run-time technique for simultaneous aging and power optimization in GPGPUs. In *Proc. Design Automation Conference (DAC)*, pages 1–6, 2014.

[28] Z. Chen and D. Marculescu. Distributed reinforcement learning for power limited many-core system performance optimization. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, 2015.

[29] S.-T. Cheng, C.-M. Chen, and J.-W. Hwang. Low-power design for real-time systems. In *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat.*, pages 1746–1750 vol.3, 1997.

[30] C. T. Chou, Y. P. Lin, K. Y. Chiang, and K. C. Chen. Dynamic buffer allocation for thermal-aware 3d network-on-chip systems. In *ICCE-TW*, 2017.

[31] I. Christoforakis, O. Tomoutzoglou, D. Bakoyiannis, and G. Kornaros. Dithering-based power and thermal management on fpga-based multi-core embedded systems. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 173–177, 2015.

[32] E.-Y. Chung, L. Benini, A. Bogliolo, and G. D. Micheli. Dynamic power management for nonstationary service requests. In *DATE*, 1999.

[33] R. Cochran and et al. Pack & cap: Adaptive dvfs and thread packing under power caps. In *MICRO*, 2011.

[34] N. Corporation. Nvidia jetson tx2 embedded module. 2017.

[35] A. Coskun, R. Strong, D. Tullsen, and T. S. Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proc. Int. Conf. Measurement and Modeling of Computer Systems*, pages 169–180, 2009.

[36] A. K. Coskun, T. S. Rosing, and K. Whisnant. Temperature aware task scheduling in mpsocs. In *2007 Design, Automation Test in Europe Conference Exhibition*, 2007.

[37] 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 *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.

[38] A. Das, M. J. Walker, A. Hansson, B. M. Al-Hashimi, and G. V. Merrett. Hardware-software interaction for run-time power optimization: A case study of embedded linux on multicore smartphones. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

[39] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[40] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[41] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In *Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '10, pages 155–162, Washington, DC, USA, 2010. IEEE Computer Society.

[42] J. Diemer, R. Ernst, and M. Kauschke. Efficient throughput-guarantees for latency-sensitive networks-on-chip. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 529–534. IEEE, 2010.

[43] Y. Ding, P. Yedlapalli, and M. Kandemir. Qos aware dynamic time-slice tuning. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014.

[44] J. Donald and M. Martonosi. Leveraging simultaneous multithreading for adaptive thermal control. 2005.

[45] Dong Hwa Kim. Parameter tuning of fuzzy neural networks by immune algorithm. In *2002 IEEE World Congress on Computational Intelligence. 2002 IEEE International Conference on Fuzzy Systems. FUZZ-IEEE'02. Proceedings (Cat. No.02CH37291)*, 2002.

[46] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. Sparta: Runtime task allocation for energy efficient heterogeneous many-cores. In *CODES, 2016*.

[47] B. Donyanavard, A. M. Rahmani, T. Muck, K. Moazemmi, and N. Dutt. Gain scheduled control for nonlinear power management in cmps. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 921–924, 2018.

[48] C. Dubach, T. M. Jones, and E. V. Bonilla. Dynamic Microarchitectural Adaptation Using Machine Learning. In *TACO, 2013*.

[49] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O'Boyle. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *MICRO, 2010*.

[50] N. Dutt, A. Jantsch, and S. Sarma. Self-aware cyber-physical systems-on-chip. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 46–50, Nov 2015.

[51] N. Dutt, A. Jantsch, and S. Sarma. Toward Smart Embedded Systems: A Self-aware System-on-Chip (SoC) Perspective. *ACM Transactions on Embedded Computing Systems*, 15, 2016.

[52] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 449–460. Ieee, Dec. 2012.

[53] H. et al. A generalized software framework for accurate and efficient management of performance goals. In *EMSOFT*, 2013.

[54] Q. W. et al. Dynamo: Facebook's data center-wide power management system. In *ISCA*, 2016.

[55] M. Fabian and A. Hellgren. *Desco — a Tool for Education and Control of Discrete Event Systems*. Springer US, 2000.

[56] R. Ge, X. Feng, S. Song, H. C. Chang, D. Li, and K. W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 658–671, 2010.

[57] B. Grigorian, N. Farahpour, and G. Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 615–626. IEEE, 2015.

[58] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-noc: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 401–412, New York, NY, USA, 2011. ACM.

[59] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive virtual clock: A flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 268–279, New York, NY, USA, 2009. ACM.

[60] U. Gupta, M. Babu, R. Ayoub, M. Kishinevsky, F. Paterna, and U. Y. Ogras. Staff: online learning with stabilized adaptive forgetting factor and feature selection algorithm. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[61] U. Gupta, J. Campbell, U. Y. Ogras, R. Ayoub, M. Kishinevsky, F. Paterna, and S. Gumussoy. Adaptive performance prediction for integrated GPUs. In *ICCAD, 2016*.

[62] M.-H. Haghbayan, A. Kanduri, A.-M. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen. MapPro: Proactive Runtime Mapping for Dynamic Workloads by Quantifying Ripple Effect of Applications on Networks-on-Chip. In *NOCS*, 2015.

[63] M. H. Haghbayan, A. Miele, A. M. Rahmani, P. Liljeberg, and H. Tenhunen. A lifetime-aware runtime mapping approach for many-core systems in the dark silicon era. In *Proc. Conf. on Design, Automation & Test in Europe (DATE)*, pages 854–857, 2016.

[64] 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. *IEEE Trans. on Computers*, 66(9):1599–1612, Sept 2017.

[65] M. H. Haghbayan, A. M. Rahmani, A. Miele, M. Fattah, J. Plosila, P. Liljeberg, and H. Tenhunen. A power-aware approach for online test scheduling in many-core architectures. *IEEE Transactions on Computers*, 65(3):730–743, March 2016.

[66] H. Hajimiri, M. A. Qathrady, and P. Mishra. Proactive thermal management using memory based computing. In *NANOARCH*, 2013.

[67] J. Hamers and L. Eeckhout. Scenario-based resource prediction for qos-aware media processing. *Computer*, 2010.

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

[69] W. Heirman, A. Isaev, and I. Hur. Sniper: Simulation-based instruction-level statistics for optimizing software on future architectures. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, EASC '15, page 29–31, GBR, 2015. University of Edinburgh.

[70] J. Heißwolf, R. König, and J. Becker. A scalable noc router design providing qos support using weighted round robin scheduling. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 625–632. IEEE, 2012.

[71] J. L. Hellerstein and et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[72] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, 2007.

[73] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 657–668. IEEE, 2016.

[74] J. P. Hespanha. Tutorial on supervisory control. Lecture Notes for the workshop *Control using Logic and Switching* for the 40th Conf. on Decision and Contr., Orlando, Florida.

[75] H. Hoffmann and et al. Dynamic Knobs for Responsive Power-aware Computing. In *ASPLOS*, 2011.

[76] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*.

[77] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, pages 8–11, 1994.

[78] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2005.

[79] J. Hu, W. Peng, and C. Chung. Reinforcement learning for hevc/h.265 intra-frame rate control. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

[80] K. Huang, L. Santinelli, J. J. Chen, L. Thiele, and G. C. Buttazzo. Adaptive dynamic power management for hard real-time systems. In *RTSS*, 2009.

[81] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 250–261, 2001.

[82] C. Imes, S. Hofmeyr, and H. Hoffmann. Energy-efficient application resource scheduling using machine learning classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, 2018.

[83] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA, 2008*.

[84] 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 *MICRO, 2006*.

[85] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO*, 2006.

[86] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings. 1998 International Symposium on Low Power Electronics and Design (IEEE Cat. No.98TH8379)*, pages 197–202, 1998.

[87] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[88] J. R. Jang. Anfis: adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man, and Cybernetics*, 1993.

[89] A. Jantsch, N. Dutt, and A. M. Rahmani. Self-awareness in systems on chip– a survey. *IEEE Design Test*, 34(6):8–26, Dec 2017.

[90] JEDEC Solid State Tech. Ass. Failure mechanisms and models for semiconductor devices. *JEDEC Publication JEP122G*, 2010.

[91] B. K. Joardar, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu. Learning-based application-agnostic 3d noc design for heterogeneous manycore systems. *IEEE Transactions on Computers*, 2018.

[92] H. Jung and M. Pedram. Stochastic dynamic thermal management: A markovian decision-based approach. In *2006 International Conference on Computer Design*, pages 452–457, 2006.

[93] Kaggle Inc. The State of Data Science and Machine Learning. `https://www.kaggle.com/surveys/2017`, 2017. Accessed: 2018-08-08.

[94] A. Kanduri, M. H. Haghbayan, A. M. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen. Dark silicon aware runtime mapping for many-core systems: A patterning approach. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015.

[95] A. Kanduri, M. H. Haghbayan, A. M. Rahmani, M. Shafique, P. Liljeberg, and A. Jantsch. dBoost: Thermal Aware Performance Boosting through Dark Silicon Patterning. *IEEE Transactions on Computers*, 2018.

[96] N. Kapadia and S. Pasricha. Varsha: Variation and reliability-aware application scheduling with adaptive parallelism in the dark-silicon era. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1060–1065, March 2015.

[97] C. Karamanolis and et al. Designing Controllable Computer Systems. In *HOTOS*, 2005.

[98] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Multi-Mechanism Reliability Modeling and Management in Dynamic Systems. *Trans. on VLSI Systems*, 16(4):476–487, 2008.

[99] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM.

[100] U. A. Khan and B. Rinner. Online learning of timeout policies for dynamic power management. *ACM Trans. Embed. Comput. Syst.*, 13(4), Mar. 2014.

[101] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An online quality management system for approximate computing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 554–566. IEEE, 2015.

[102] R. G. Kim, W. Choi, Z. Chen, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu. Imitation learning for dynamic vfi control in large-scale manycore systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[103] T. Kim, X. Huang, H. B. Chen, V. Sukharev, and S. X. D. Tan. Learning-based dynamic reliability management for dark silicon processor considering EM effects. In *Proc. Conf. on Design, Automation & Test in Europe (DATE)*, pages 463–468, 2016.

[104] D. Kirovski and M. Potkonjak. System-level synthesis of low-power hard real-time systems. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 697–702, 1997.

[105] G. J. Klir and T. A. Folge. *Fuzzy Sets, Uncertainty and Information*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1988.

[106] G. J. Klir and B. Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.

[107] S. Kounev, P. Lewis, K. L. Bellman, N. Bencomo, J. Camara, A. Diaconescu, L. Esterle, K. Geihs, H. Giese, S. Götz, P. Inverardi, J. O. Kephart, and A. Zisman. *The Notion of Self-aware Computing*. Springer International Publishing, Cham, 2017.

[108] C. M. Krishna and Y. H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings Sixth IEEE Real-Time Technology and Applications Symposium. RTAS 2000*, pages 156–165, 2000.

[109] C. Kulkarni, F. Catthoor, and H. D. Man. Code transformations for low power caching in embedded multimedia processors. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 292–297, 1998.

[110] D. E. Lackey, P. S. Zuchowski, T. R. Bednar, D. W. Stout, S. W. Gould, and J. M. Cohn. Managing power and performance for system-on-chip designs using voltage islands. In *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, 2002.

[111] R. Laddaga. Active Software. In *Proceedings of the First International Workshop on Self-adaptive Software*, IWSAS' 2000, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[112] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

[113] H. Lee, M. Shafique, and M. A. A. Faruque. Aging-aware Workload Management on Embedded GPU Under Process Variation. *IEEE Trans. on Computers*, PP(99):1–1, 2018.

[114] J. W. Lee, M. C. Ng, and K. Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 89–100, Washington, DC, USA, 2008. IEEE Computer Society.

[115] S. Lee, K. Kang, and C. M. Kyung. Runtime thermal management for 3-d chip-multiprocessors with hybrid sram/mram l2 cache. *TVLSI*, 2015.

[116] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings 37th Design Automation Conference*, pages 806–809, 2000.

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

[118] B. Li, L.-S. Peh, L. Zhao, and R. Iyer. Dynamic qos management for chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 9(3):17:1–17:29, Oct. 2012.

[119] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell. Coqos: Coordinating qos-aware shared resources in noc-based socs. *Journal of Parallel and Distributed Computing*, 71(5):700–713, 2011.

[120] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.

[121] Y. Li and W. Wolf. A task-level hierarchical memory model for system synthesis of multiprocessors. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 153–156, 1997.

[122] W. Liao, L. He, and K. M. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *TCAD*, 2005.

[123] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 840–845, 2001.

[124] L. Ljung. Black-box models from input-output measurements. In *I2MTC, 2001.*

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

[126] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ISCA*, Piscataway, NJ, USA, 2014.

[127] W. H. Lo, K. z. Liang, and T. Hwang. Thermal-aware dynamic page allocation policy by future access patterns for hybrid memory cube (hmc). In *DATE*, 2016.

[128] J. Luo and N. K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 444–449, 2001.

[129] K. Ma and X. Wang. PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22, 2012.

[130] T. C. L. Ma and K. G. Shin. A user-customizable energy-adaptive combined static/dynamic scheduler for mobile applications. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 227–236, 2000.

[131] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu. Improving System-Level Lifetime Reliability of Multicore Soft Real-Time Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1895–1905, June 2017.

[132] Y. Ma, T. Chantem, R. P. Dick, S. Wang, and X. S. Hu. An on-line framework for improving reliability of real-time systems on x201c;big-little x201d; type mpsocs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 446–451, March 2017.

[133] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *CDC, 2010*.

[134] D. Mahajan and et al. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *ISCA*, 2016.

[135] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh. Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 66–77, Piscataway, NJ, USA, 2016. IEEE Press.

[136] Man-Lap Li, R. Sasanka, S. V. Adve, Yen-Kuang Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 34–45, Oct 2005.

[137] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, pages 3239–3242 vol.6, 2000.

[138] A. Manzak and C. Chakrabarti. Variable voltage task scheduling algorithms for minimizing energy. In *Low Power Electronics and Design, International Symposium on, 2001.*, pages 279–282, 2001.

[139] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, 1992.

[140] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

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

[142] MathWorks. System PID tuner Toolbox. Technical report, 2017.

[143] M. Meterelliyoz, H. Mahmoodi, and K. Roy. A leakage control system for thermal stability during burn-in test. In *ITC*, 2005.

[144] A. Miele, A. Kanduri, K. Moazzemi, D. Juhász, A. R. Rahmani, N. Dutt, P. Liljeberg, and A. Jantsch. On-chip dynamic resource management. *Foundations and Trends® in Electronic Design Automation*, 13(1-2):1–144, 2019.

[145] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. volume 1, pages 25–34, 2010.

[146] A. K. Mishra and et al. CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors. In *SC*, 2010.

[147] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann. Caloree: Learning control for predictable latency and low energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[148] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[149] K. Moazzemi, C. Hsieh, and N. Dutt. Hamex: heterogeneous architecture and memory exploration framework. In *2016 International Symposium on Rapid System Prototyping (RSP)*, pages 1–7, 2016.

[150] K. Moazzemi, A. Kanduri, D. Juhász, A. Miele, A. M. Rahmani, P. Liljeberg, A. Jantsch, and N. Dutt. Trends in on-chip dynamic resource management. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 62–69, 2018.

[151] K. Moazzemi, B. Maity, S. Yi, A. M. Rahmani, and N. Dutt. Hessle-free: ¡u¿he¡/u¿terogeneou¡u¿s¡/u¿ ¡u¿s¡/u¿ystems ¡u¿le¡/u¿veraging ¡u¿f¡/u¿uzzy control for ¡u¿r¡/u¿untim¡u¿e¡/u¿ resourc¡u¿e¡/u¿ management. *ACM Trans. Embed. Comput. Syst.*, 18(5s), Oct. 2019.

[152] T. Moreau, F. Augusto, P. Howe, A. Alaghi, and L. Ceze. Exploiting quality-energy tradeoffs with arbitrary quantization: special session paper. In *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*, page 30. ACM, 2017.

[153] A. S. Morse. *Control using logic-based switching*. Springer London, 1997.

[154] T. Mück, B. Donyanavard, and N. Dutt. Poiicym: Rapid prototyping of resource management policies for hmps. In *Proceedings of the 28th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, RSP '17, pages 23–29, New York, NY, USA, 2017. ACM.

[155] T. R. Mück. *Reflective On-Chip Resource Management Policies for Energy-Efficient Heterogeneous Multiprocessors*. PhD thesis, University of California, Irvine, USA, 2018.

[156] J. Murray, R. Kim, P. Wettin, P. P. Pande, and B. Shirazi. Performance evaluation of congestion-aware routing with dvfs on a millimeter-wave small-world wireless noc. *J. Emerg. Technol. Comput. Syst.*, 11(2), Nov. 2014.

[157] T. Mück, B. Donyanavard, K. Moazzemi, A. M. Rahmani, A. Jantsch, and N. Dutt. Design methodology for responsive and rrobust mimo control of heterogeneous multicores. *IEEE TMSCS*, 2018.

[158] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*, pages 182–188, Oct 2006.

[159] L. S. Nielsen, C. Niessen, J. Sparso, and K. van Berkel. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *TVLSI*, 1994.

[160] M. Niknafs, I. Ukhov, P. Eles, and Z. Peng. Runtime resource management with workload prediction. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, 2019.

[161] NIST. Engineering Statistics Handbook. Technical report, 2012.

[162] U. Y. Ogras, R. Marculescu, and D. Marculescu. Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In *2008 45th ACM/IEEE Design Automation Conference*, 2008.

[163] U. Y. Ogras, R. Marculescu, D. Marculescu, and E. G. Jung. Design and management of voltage-frequency island partitioned networks-on-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2009.

[164] J. Ouyang and Y. Xie. Loft: A high performance network-on-chip providing quality-of-service support. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 409–420. IEEE, 2010.

[165] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Commun. ACM*, 46(10):24–28, Oct. 2003.

[166] R. Parloff. Why deep learning is suddenly changing your life. *Fortune*, 2016.

[167] K. M. Passino and S. Yurkovich. *Fuzzy Control*. Addison-Wesley, 1997.

[168] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *HPCA*, 2015.

[169] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(2):412–423, June 2007.

[170] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 89–102, 2001.

[171] R. Pothukuchi and et al. A Guide to Design MIMO Controllers for Architectures. *http://iacoma.cs.uiuc.edu/iacoma-papers/mimoTR.pdf*, 2016.

[172] R. P. Pothukuchi and et al. Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures. In *ISCA*, 2016.

[173] Q. Wu, P. Juang, M. Martonosi, D. W. Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *ASPLOS, 2004.*

[174] Q. Qiu and M. Pedram. Dynamic power management based on continuous-time markov decision processes. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 555–561, 1999.

[175] G. Qu, D. Kirovski, M. Potkonjak, and M. B. Srivastava. Energy minimization of system pipelines using multiple voltages. In *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, pages 362–365 vol.1, 1999.

[176] J. Rada-Vilela. The fuzzylite libraries for fuzzy logic control, 2018.

[177] A. Rahmani, P. Liljeberg, A. Hemani, A. Jantsch, and H. Tenhunen. *The Dark Side of Silicon.* Springer, 2016.

[178] 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 *ASPLOS*, 2018.

[179] 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 *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015.

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

[181] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 1989.

[182] E. A. Rambo, T. Kadeed, R. Ernst, M. Seo, F. Kurdahi, B. Donyanavard, C. B. de Melo, B. Maity, K. Moazzemi, K. Stewart, S. Yi, A. M. Rahmani, N. Dutt, F. Maurer, N. A. Vu Doan, A. Surhonne, T. Wild, and A. Herkersdorf. The information processing factory: A paradigm for life cycle management of dependable systems. In *2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2019.

[183] P. Ranganathan and N. Jouppi. Enterprise it trends and implications for architecture research. In *HPCA*, pages 253–256. IEEE, 2005.

[184] S. Rezaei, E. Bozorgzadeh, and K. Kim. Ultrashare: Fpga-based dynamic accelerator sharing and allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019.

[185] S. Rezaei, K. Kim, and E. B. Scalable multi-queue data transfer scheme for fpga-based multi-accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 374–380. IEEE, 2018.

[186] C. Rusu, A. Ferreira, C. Scordino, and A. Watson. Energy-efficient real-time heterogeneous server clusters. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 418–428, 2006.

[187] A. Sadighi, B. Donyanavard, T. Kadeed, K. Moazzemi, T. Mück, A. Nassar, A. M. Rahmani, T. Wild, N. Dutt, R. Ernst, A. Herkersdorf, and F. Kurdahi. Design methodologies for enabling self-awareness in autonomous systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1532–1537, 2018.

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

[189] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4, 2009.

[190] M. Samadi, J. Lee, and D. Jamshidi. Sage: Self-tuning approximation for graphics engines. 2013.

[191] S. Sarma, N. Dutt, P. Gupta, A. Nicolau, and N. Venkatasubramanian. On-chip self-awareness using cyberphysical-systems-on-chip (cpsoc). In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, CODES '14, pages 22:1–22:3, New York, NY, USA, 2014. ACM.

[192] Seema Chopra, R. Mitra, and Vijay Kumar. Identification of rules using subtractive clustering with application to fuzzy controllers. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826)*, 2004.

[193] Semiconductor Industry Association et al. International Technology Roadmap for Semiconductors. http://www.itrs2.net/, 2011.

[194] M. Shafique, B. Vogel, and J. Henkel. Self-adaptive hybrid dynamic power management for many-core systems. In *DATE*, 2013.

[195] S. Shahhosseini, K. Moazzemi, A. M. Rahmani, and N. Dutt. On the feasibility of siso control-theoretic dvfs for power capping in cmps. *Microprocessors and Microsystems*, 63:249 – 258, 2018.

[196] S. Shahosseini, K. Moazzemi, A. M. Rahmani, and N. Dutt. Dependability evaluation of siso control-theoretic power managers for processor architectures. In *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2017.

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

[198] E. Shamsa, A. Kanduri, A. M. Rahmani, P. Liljeberg, A. Jantsch, and N. Dutt. Goal formulation: Abstracting dynamic objectives for efficient on-chip resource allocation. In *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–4. IEEE, 2018.

[199] 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 *Proc. of Conf. on Design, Automation Test in Europe (DATE)*. IEEE, 2019.

[200] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: Meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[201] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 134–139, 1999.

[202] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. pages 124–134, 2011.

[203] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 2–13, 2003.

[204] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael. Exploring system availability during software-based self-testing of multi-core cpus. *Journal of Electronic Testing*, 34(1):67–81, Feb 2018.

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

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

[207] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili. Managing performance-reliability tradeoffs in multicore processors. In *IEEE International Reliability Physics Symposium*, pages 3C.1.1–3C.1.7, 2015.

[208] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[209] J. Srinivasan, S. Adve, P. Bose, and J.A.Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proc. Int. Symp. on Computer Architecture*, pages 276–287, 2004.

[210] J. T. Starczewski. *Defuzzification of Uncertain Fuzzy Sets*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[211] 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 *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 62–75, New York, NY, USA, 2015. ACM.

[212] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali. Proactive control of approximate programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 607–621, New York, NY, USA, 2016. ACM.

[213] J. Sun, R. Lysecky, K. Shankar, A. Kodi, A. Louri, and J. Roveda. Workload Assignment Considering NBTI Degradation in Multicore Systems. *Journal Emerg. Technol. Comput. Syst.*, 10(1):4:1–4:22, Jan. 2014.

[214] H. Sung, J. Min, S. Ha, and H. Eom. Ombm: Optimized memory bandwidth management for ensuring qos and high server utilization. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 269–276. IEEE, 2017.

[215] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, New York, NY, USA, 2012. ACM.

[216] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 283–294. ACM, 2011.

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

[218] A. Tilli, A. Bartolini, M. Cacciari, and L. Benini. Guaranteed computational resprinting via model-predictive control. *ACM Trans. Embed. Comput. Syst.*, 2015.

[219] M. Torabzadehkashi, S. Rezaei, A. HeydariGorji, H. Bobarshad, V. Alves, and N. Bagherzadeh. Computational storage: an efficient and scalable platform for big data and hpc applications. *Journal of Big Data*, 6(1):100, 2019.

[220] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 335–344, New York, NY, USA, 2012. Association for Computing Machinery.

[221] O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-cache for hot multimedia. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 274–283, 2001.

[222] O. S. Unsal and I. Koren. System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, pages 1055–1069, 2003.

[223] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. The minimax cache: an energy-efficient framework for media processors. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 131–140, 2002.

[224] A. Vassighi and M. Sachdev. Thermal runaway in integrated circuits. *IEEE Transactions on Device and Materials Reliability*, 2006.

[225] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, Dec 2001.

[226] T. Wang, Q. Zhang, and Q. Xu. Approxqa: a unified quality assurance framework for approximate computing. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 254–257. European Design and Automation Association, 2017.

[227] Y. Wang, K. Ma, and X. Wang. Temperature-constrained Power Control for Chip Multiprocessors with Online Model Estimation. In *ISCA*, 2009.

[228] C. Weis, A. Mutaal, O. Naji, M. Jung, A. Hansson, and N. Wehn. Dramspec: A high-level dram timing, power and area exploration tool. *Int. J. Parallel Program.*, 45(6):1566–1591, Dec. 2017.

[229] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[230] M. Wolf, S. Bhattacharyya, J. Florence, and A. E. Sapio. Power and thermal modeling for communication systems. In *SiPS*, 2016.

[231] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 24–36, New York, NY, USA, 1995. Association for Computing Machinery.

[232] Q. Wu and et al. Formal control techniques for power-performance management. *IEEE Micro*, 2005.

[233] Y. Xiang and S. Pasricha. Soft and hard reliability-aware scheduling for multicore embedded systems with energy harvesting. *IEEE Transactions on Multi-Scale Computing Systems*, 1(4):220–235, Oct 2015.

[234] C. Xu, X. Wu, W. Yin, Q. Xu, N. Jing, X. Liang, and L. Jiang. On quality trade-off control for approximate computing using iterative training. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 52:1–52:6, New York, NY, USA, 2017. ACM.

[235] C. Xu, X. Wu, W. Yin, Q. Xu, N. Jing, X. Liang, and L. Jiang. On quality trade-off control for approximate computing using iterative training. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 52:1–52:6, New York, NY, USA, 2017. ACM.

[236] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*.

[237] Z. Yang, L. Li, and B. Liu. Auto-tuning method of fuzzy pid controller parameter based on self-learning system. In *2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2014.

[238] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 374–382, 1995.

[239] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 381–392, New York, NY, USA, 2014. ACM.

[240] S. Yi, I. Yoon, C. Oh, and Y. Yi. Real-time integrated face detection and recognition on embedded GPGPUs. In *ESTIMedia*, 2014.

[241] K. Yu. big.LITTLE Switchers. In *2012 Korea Linux Forum*, 2012.

[242] H. Zhang and et al. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS*, 2016.

[243] T. Zhang, J. L. Abellán, A. Joshi, and A. K. Coskun. Thermal management of manycore systems with silicon-photonic networks. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014.

[244] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[245] Y. Zhang, J. Yao, and H. Guan. Intelligent cloud resource management with deep reinforcement learning. *IEEE Cloud Computing*, 2017.

[246] Y. Zhou, H. Hoffmann, and D. Wentzlaff. Cash: Supporting iaas customers with a sub-core configurable architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 682–694, Piscataway, NJ, USA, 2016. IEEE Press.

[247] H. J. Zimmerman. *Fuzzy Set Theory–and Its Applications*. Kluwer Academic Press, Boston, MA, USA, 1991.