INTELLIGENT SYSTEMS FOR EFFICIENCY AND SECURITY

BY

RAGHAVENDRA PRADYUMNA POTHUKUCHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

ABSTRACT

As computing becomes ubiquitous and personalized, resources like energy, storage and time are becoming increasingly scarce and, at the same time, computing systems must deliver in multiple dimensions, such as high performance, quality of service, reliability, security and low power. Building such computers is hard, particularly when the operating environment is becoming more dynamic, and systems are becoming heterogeneous and distributed.

Unfortunately, computers today manage resources with many ad hoc heuristics that are suboptimal, unsafe, and cannot be composed across the computer's subsystems. Continuing this approach has severe consequences: underperforming systems, resource waste, information loss, and even life endangerment.

This dissertation research develops computing systems which, through intelligent adaptation, deliver efficiency along multiple dimensions. The key idea is to manage computers with principled methods from formal control. It is with these methods that the multiple subsystems of a computer sense their environment and configure themselves to meet system-wide goals.

To achieve the goal of intelligent systems, this dissertation makes a series of contributions, each building on the previous. First, it introduces the use of formal MIMO (Multiple Input Multiple Output) control for processors, to simultaneously optimize many goals like performance, power, and temperature. Second, it develops the *Yukta* control system, which uses coordinated formal controllers in different layers of the stack (hardware and operating system). Third, it uses robust control to develop a fast, globally coordinated and decentralized control framework called *Tangram*, for heterogeneous computers. Finally, it presents *Maya*, a defense against power side-channel attacks that uses formal control to reshape the power dissipated by a computer, confusing the attacker. The ideas in the dissertation have been demonstrated successfully with several prototypes, including one built along with AMD (Advanced Micro Devices, Inc.) engineers. These designs significantly outperformed the state of the art.

The research in this dissertation brought formal control closer to computer architecture and has been well-received in both domains. It has the first application of full-fledged MIMO control for processors, the first use of robust control in computer systems, and the first application of formal control for side-channel defense. It makes a significant stride towards intelligent systems that are efficient, secure and reliable.

Kṛṣṇārpaṇamastu
dedicated to Krishna

ACKNOWLEDGMENTS

There are many individuals who helped me pursue the work leading to this dissertation. The support from each of them is invaluable and irreplaceable.

This interdisciplinary work would not have been possible without the freedom of thought that I was given by my advisor, Professor (Prof.) Josep Torrellas. He has been very supportive and patient in guiding my work. Beyond research, I have learnt many things from him: envisioning a research career, organizing work, choosing important problems, technical writing and collaborative work. Thanks to him I feel prepared and confident as I graduate from studenthood.

My collaborators, some of whom are in my dissertation committee, have contributed immensely to this work. Prof. Petros Voulgaris has been like a second advisor who has helped me understand how formal control could be applied to computers. Thanks to Dr. Joseph Greathouse, I had the opportunity to work with Advanced Micro Devices, Inc. (AMD) and learn about perspectives from industry. This experience has added great value to my research. Discussions with Prof. Laxmikant Kale enabled me to understand how my work could scale to large systems. The feedback from Prof. Tarek Abdelzaher on using control for computing has been very helpful. I also thank him for generously allowing me to run my experiments on his cloud computing testbed. My discussions with Prof. Wen-Mei Hwu and Prof. Nam Sung Kim were valuable in structuring my work. The Chai applications from Prof. Wen-Mei's group have been a strong factor in shaping my research on heterogeneous systems. Dr. Amin Ansari who was a postdoc at Illinois when I started, introduced me to the idea of using formal control for computer architectures. Indeed, this was the spark that ignited the path of my dissertation.

Prof. Alexander Schwing helped me design the Machine Learning attacks for my security work. At AMD, Karthik Rao has been a strong collaborator and I have also gained a good friend in him. Leonardo Piga and Christopher Erb at AMD have significantly supported me during my work there. Kevin James Colravy, from the Department of Electrical & Computer Engineering (ECE) at the University of Illinois at Urbana-Champaign (UIUC), setup the measurement infrastructure for my security experiments on electrical power outlets. His help was vital in evaluating my security research ideas. Dipanjan Das, also from ECE, provided the scripts for data acquisition for these experiments. I also acknowledge funding agencies like NSF (National Science Foundation) and DARPA (Defense Advanced Research Projects Agency) that made my research possible.

My wife, Sweta Yamini Pothukuchi, contributed a lot to this dissertation. We co-authored papers, wrote code, analyzed experiments and in fact, her work made it possible for us make it in time for several conference deadlines.

I think one place where bureaucracy has been successful in helping people is my department at UIUC, at least from my experience as a student. Dana Garard, Madeleine Garvey, Viveka Kudaligama, Maggie Metzger Chappell, Sherry Unkraut and other administrative staff have been very accommodating and helpful. Be it conferences, travel, meetings or paperwork, they made it very easy for me and always received me with a smile. I also thank the Engineering IT Staff for their help in setting up my experimental platforms.

I am fortunate to have the company of my groupmates, Bhargava Gopireddy, Jiho Choi, Thomas Shull, Aditya Agrawal, Azin Heidershenas, Wooil Kim, Yasser Shalabi, Tanmay Gangwani, Mengjia Yan, Houxiang Ji, Hyuongwook Nam, Apostolos Kokolis, Dimitrios Skarlatos and other iacoma group members, for not only their valuable feedback on my work but also for all the time we spent together—it has been a great experience. I have also found good friends in Ashutosh Dhar, Tarun Prabhu, Radha Venkatagiri and Matthew Sinclair.

It is my fortune to have a large group of friends who bore with me through thick and thin, and I have plenty of good things to fondly remember for years to come. Mahesh Akella, Srujana and Raghavendhar, Sriraam Chandrasekaran, Anand Deshmukh, Vijayalakshmi Devarakonda, Prasanna Giridhar, Ramsai Gorugantu, Siva Theja Maguluri, Krishna Kalyan Medarametla, Sree Kalyan Patiballa, Sasidhar Potukuchi, Snehita Srivarma, Vinay Muttineni, Shilpa and Charan, Anusha Reddy, Kartik Reddy, Srikanthan Sridharan—in this limited space here, I cannot list them all, but my heartfelt thanks go to each one of them.

It would be silly of me to try to use a few words to acknowledge what my family has given me. They have seen me at my best and worst all these years and my well-being has always been their priority. All the love, care, patience and support that they have been unconditionally showering on me makes me feel very lucky. My parents and parents in law have come to the United States multiple times, taking a leave from their busy work, solely to help me and Sweta in taking care of our kids, so that we could complete our research on time. Our siblings and siblings-in-law have helped us significantly at every opportunity. I owe this dissertation to my family.

During my graduate school, I experienced the death of my maternal grandparents who were very dear to me. I regret that I wasn't around them in their final moments. It is also during my Ph.D. that my two children were born and as luck would have it, I had a publication acceptance each time—I hope childbirth does not become a prerequisite for my future publications.

Speaking of luck, I thank the anonymous reviewers of my papers who helped improve my research, and also to those reviewers, without whose unreasonable rejections, my degree would have been completed much earlier.

I am forever in debt to my teachers at school, Birla Institute of Technology & Science (BITS), Pilani, UIUC and elsewhere who transformed me to what I am now. There are many others who touched my life through its course, gently nudging, guiding, helping and channelling it to give it the form and momentum that it has today. I am grateful to all of them.

I am aware of the serendipity with which I have found ideas and people, and how important events transpired in my life. I attribute this to *Mahāmāya*, the Queen of the Absolute Truth, who revolves the wheel of this world -"*Mahāmāyā viśvam bhramayasi parabrahma mahiṣī̄*" (*Sauṃdaryalahari, Ādi Śaṃkarācārya*).

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

As computing becomes ubiquitous and personalized, resources like energy, storage and time become increasingly scarce and, at the same time, computing systems must deliver in multiple dimensions, such as high performance, quality of service, reliability, security and low power. This has prompted computers to include controllers that manage multiple resources during execution [1,2,3,4,5,6]. Processors too, are being built to be more adaptive, with extensive performance, power and thermal management [3,4,7,8,9], and there is a myriad of reconfiguration capabilities under research (e.g., [10,11,12,13,14]). In this environment, intelligent resource control is of paramount importance for computers to be efficient and secure—and the design of such intelligent systems is the focus of this dissertation.

1.1 CHALLENGES IN BUILDING RESOURCE EFFICIENT COMPUTERS

Building resource efficient computers that deliver many goals is hard because of several reasons. First, as a program runs, we increasingly want to enforce *multiple* requirements *at the same time* — e.g., that the power be no higher than a target limit, the frame rate of the application be no lower than a target rate, and the average busy core utilization be no lower than a target value. It is difficult to quickly identify the best configuration from the many choices that achieves all these goals.

Second, modern computing systems are organized in multiple layers, each with its own resources and with partial information about the current execution such as the hardware, OS (Operating System), and networking layers. To meet the multiple resource constraints in an execution, each layer must use only the information available to it to manage its layer, and communicate with other layers for overall efficiency. Designing controllers in this environment is challenging because each layer's controller is modularly designed, despite which, it should coordinate with other unknown controllers for overall efficiency.

Third, the emerging trend in today's computing systems is to integrate subsystems built by different vendors into heterogeneous computers [15,16,17,18,19,20,21]. Such subsystems can be CPUs, GPUs, or various accelerators. This approach is attractive because the individual components are often easier and cheaper to develop separately, and they can be reused across multiple products. For example, the same GPU design is used in AMD's Ryzen mobile processors [22] and in Intel's multi-chip Core i7-8809G [23]. Building resource controllers in computers with multiple subsystems is challenging. There is a tension between the need to generate *local* decisions in each subsystem quickly for timely response and the desire to coordinate the different subsystems for *global* optimization. Global coordination is especially challenging in heterogeneous computers with multi-vendor subsystems, as one needs to compose logic from different vendors that was designed without knowing the full system configuration.

Finally, resource control must also consider security. The more efficient a computer becomes, the easier it is for attackers to capture information through physical side channels like the computer's power consumption, temperature, and electromagnetic (EM) signals. Conventionally, efficiency has been the primary goal of resource control and there is not much exploration into utilizing resource control for security.

Unfortunately, physical signals have been increasingly exploited as side and covert channels. Through these physical channels, attackers have been able to exfiltrate a variety of information about the applications running, including keystrokes and passwords [24], location, browser and camera activity [25, 26, 27], and encryption keys [28]. Many types of platforms have been successfully attacked using these physical channels, including smartphones, personal computers, cloud servers, multi-tenant datacenters, and home appliances [24, 25, 26, 27, 29, 30, 31, 32, 33]. The methods through which physical signals can be acquired have significantly grown in number and stealth [34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]. Since most of these techniques simply collect measurements, they cause little interference to the target computer and are hard to detect.

1.2 LIMITATIONS OF THE STATE OF THE ART

Unfortunately, computers today have not fully addressed the challenges described earlier. Resource control is suboptimally organized across the system layers and the various heterogeneous subsystems, and is primarily run using ad hoc heuristics.

For heterogeneous hardware, the current approach chosen by industry is to use centralized decision-making [9, 46, 47, 48, 49, 50, 51], despite the availability of per-subsystem sensors and actuators. The reason is the difficulty of composing independently designed controllers for system-wide efficiency [51, 52, 53]. There are many heuristic policies that are difficult for designers to develop even within a single subsystem like a CPU; it is even harder to redesign such fragile logic to make it work across subsystems [51].

Researchers too have examined the joint optimization of multiple hardware subsystems. For example, they have optimized the combination of CPU and GPU [52, 53], GPU and memory [54], CPU and memory [55], multiple cores in a multicore [56, 57, 58, 59, 60], and servers in a datacenter [61]. In many of these works, however, the decision-making is centralized [52, 53, 54, 55, 56, 57, 58, 59]. In addition, many of these systems also rely on heuristics, which cause other problems. To control emerging heterogeneous computers effectively, we need a new approach that is fast, globally coordinated, and modular.

Across the system layers, there are many proposals on managing resources in a multilayer environment (e.g., [61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80]). One approach is to have a single, *monolithic* controller that takes signals from all the layers and controls the resources in all the layers. This solution is complex to develop, difficult to maintain, cannot interoperate across systems, and is not scalable [61,62,63,65,66,69]. Indeed, to build these controllers, designers have to understand the inner details of all the layers. Moreover, when one of the layers needs to be modified, these controllers require a complete re-design. Sometimes, this design is infeasible, as when hardware and OS come from different vendors.

An alternative is to have a controller at each layer, and have them operate in an uncoordinated, *decoupled* way. However, without coordination, the overall behavior can be greatly suboptimal [62,65,67,68,69,76]. The different controllers may interfere destructively.

An example of such interference is described by Vega et al. [78] in an IBM POWER7. In this system, there is a per-core hardware DVFS (Dynamic Voltage and Frequency Scaling) controller that changes the core's frequency to maintain high utilization and meet power requirements. In the OS, there is a task scheduler that tries to consolidate threads onto cores and power-gate the resulting unused cores to save power. When the multicore's load goes down, it is expected that the scheduler will consolidate threads to reduce power without hurting performance. Unfortunately, the DVFS controller immediately reduces the frequency to increase utilization, preventing the scheduler from ever consolidating threads and power-gating cores.

There is a need to use modular controllers in each layer of a multilayer system that collaborate through a mutually agreed interface, without a monolithic or decoupled design. This is the position taken by industry, where hardware and software companies such as ARM and Linaro are working on coordinated hardware-software approaches [6]. Other examples where hardware and software power management modules coordinate with each other include IBM [81] and Intel systems [1,5,82]. In these designs, each layer performs its own resource management, and interacts with the other layers through well-defined interfaces.

Unfortunately, most existing coordinated, multilevel designs also rely substantially on heuristics [63,68,69,81]. Using heuristics, it is not obvious how to build algorithms that meet multiple resource management goals. For example, suppose that, to accomplish certain goals, we can move jobs across cores and change each core's frequency, issue width, and load/store queue size. It is unclear by how much and in what order we need to change each of these parameters. Second, this approach requires developing complex algorithms, which may end up being buggy. As the algorithm executes, there is the danger that unexpected corner cases cause large deviations from the targets. Third, heuristic designs are highly specific to particular choices and do not address the general problem. Their algorithms may become unusable when a different hardware or software platform is used. Moreover, even highly tuned heuristics can perform poorly on application corner cases [67, 83].

As a result, despite their high overheads during design and verification, heuristics are still suboptimal, unsafe, and cannot be composed across the computer's subsystems. Continuing this approach has severe consequences: underperforming systems, resource waste, information loss, and even life endangerment.

On the security front, research on defenses against power side channels has limitations. One of them is that most prior research on defenses has focused on encryption circuits (e.g., [28, 84, 85, 86, 87, 88, 89]). In practice, there are many attacks that are easier to mount, and which use system- or chip-level power measurements to steal sensitive information not related to encryption, like application activity, passwords, and browsing data [24, 25, 26, 27, 29, 30, 31, 32].

Another limitation of many proposed defense techniques is that they require new hardware and, hence, leave existing computers in the field vulnerable. Obvious mechanisms such as maintaining constant power, inserting noise, or simply randomizing DVFS levels have been proven unsuccessful because they do not completely mask application activity [25, 29, 42, 90].

An alternative approach is to modify *each* application individually, so that its activity is not visible through physical side channels [41]. However, this is a very costly proposition.

There is an urgent need to develop effective defenses against power side channels that do not rely on special hardware, and which can be implemented as firmware or privileged software in an application-transparent manner. It is relevant to note that many common attacks that steal personal data like keystrokes or browser activity, analyze signals by sampling at intervals of several milliseconds or longer — suggesting that a firmware- or software-level defense is a good choice.

1.3 GOAL AND OVERVIEW OF THE DISSERTATION

The goal of this dissertation is to develop computing systems which, through intelligent adaptation, deliver efficiency along multiple dimensions. The approach that we propose is to use formal methodologies from control theory, whose properties are well studied [91]. By using principled methods from formal control, the multiple subsystems of a computer sense their environment and configure themselves to meet system-wide goals.

This dissertation makes a series of contributions, each building on the previous, to realize the target of building intelligent computers with formal control. First, it focuses on processors, and introduces the idea of formal MIMO (Multiple Input Multiple Output) control, with which a processor dynamically changes many parameters like frequency and cache size, to simultaneously optimize many goals like performance, power and temperature [92,93]. Second, it considers the need to coordinate the multiple layers in which computers are organized. Each layer, like the hardware, OS, and networking layers, is built independently and has its own functionality. Hence, to manage the computer scalably and portably, this work introduces the *Yukta* control system, which uses modular formal controllers in multiple layers [94,95,96]. With Yukta, the different layers exchange coordination signals, and tolerate any interference between each other to optimize full-system efficiency.

As computers become large and heterogeneous, they are integrated with CPUs, GPUs, accelerators and other subsystems provided by many vendors. To manage these computers effectively, this dissertation proposes the *Tangram* framework, where the different controllers are connected hierarchically and exchange standardized coordination signals [97]. Tangram enables fast, globally coordinated, and decentralized control of computers.

Finally, the more efficient a computer becomes, the easier it is for attackers to capture information through physical side channels like the computer's power. Therefore, this work introduces *Maya*, an application-transparent defense where firmware or privileged software uses formal control to reshape the power dissipated by the computer [98]. The resulting power signature appears to carry activity information which, in fact, is unrelated to the application, and confuses the attacker.

1.4 SPECIFIC CONTRIBUTIONS AND NOVELTY

This dissertation makes the following contributions to the field of computer architecture and systems:

1. This dissertation introduces full-fledged MIMO control for processors.
   (a) It applies MIMO control theory techniques to develop controllers that dynamically tune architectural parameters in processors. To our knowledge, this is the first work in this area.
   (b) It gives architectural intuition for the different procedures involved in the design of MIMO controllers. We develop a methodology to guide architects in designing MIMO controllers.
   (c) It discusses three general ways in which a MIMO controller can be used in computer systems.

2. This dissertation introduces modular coordinated controllers for the uncertain, multi-layered environment of modern computers.
    (a) It introduces *Yukta*, the first approach for independent teams to design coordinated multilayer controllers.
    (b) It presents the first application of Structured Singular Value (SSV) control from robust control theory to computer systems.
    (c) It develops a prototype of Yukta on a big.LITTLE multicore board, evaluates it, and performs sensitivity studies.
3. This dissertation develops the fundamental insights for building resource control frameworks for heterogeneous computers.
    (a) It presents *Tangram*, a fast, globally coordinated, and modular control framework to manage heterogeneous computers.
    (b) It introduces a novel controller design that combines multiple engines and uses formal control principles.
    (c) It develops a prototype of Tangram in a server that we build using components from three different vendors, and evaluates it.
4. Finally, this dissertation develops *Maya*, a new defense technique against power side channels using formal control to re-shape the power signal.
    (a) It presents the first application of formal control to side-channel defense.
    (b) It shows an implementation of Maya using only privileged software. To our knowledge, this is the first defense against power side channels that is readily-deployable and application-transparent. It operates at millisecond-level sampling, and thwarts power attacks that require no physical access.
    (c) It develops an evaluation of Maya using machine learning-based attacks on three different real machines, in one case tapping an AC electric power outlet.

CHAPTER 2: BACKGROUND

2.1 BASICS OF APPLYING FORMAL CONTROL FOR COMPUTERS

In control theory, a controlled system is represented as a feedback control loop as in Figure 2.1(a). The controller reads the output y of a system in state x, compares it to the target value $y_\circ$ and, based on the difference (or error), generates the input u to be applied to the system to reduce the error.

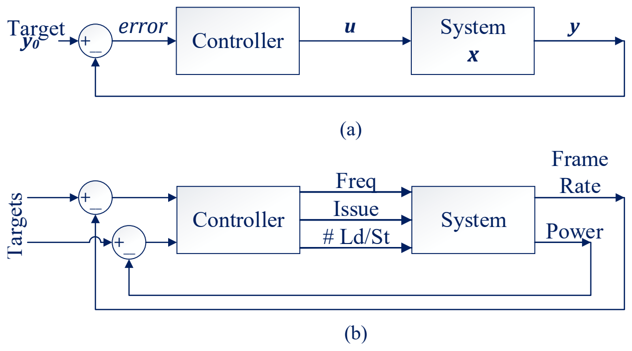


Figure 2.1: Typical feedback control loop.

Control theory has been used to design controllers that tune architectural parameters in processors (e.g., [1, 13, 99, 100, 101, 102, 103, 104]). To the best of our knowledge, in these proposals, a controller only controls a single output (y). The large majority of these proposals use a SISO design: single input to the system (u) and single output (y) [1, 13, 102, 104]. For example, Lu et al. [13] control the frame rate of multimedia applications by changing the frequency. This is a limited approach.

The other designs use a MISO approach: multiple inputs and single output. Some of them combine multiple SISO models to generate a larger MISO controller. For example, Wang et al. [101] control the total power of a multicore by changing the frequencies of all the cores. The multicore's power is the sum of the powers of all the cores, and each core's power only depends on that core's frequency. Changing a core's frequency (input) only impacts the power of that core (output); it does not impact the power of all the other cores — at least, not directly. A similar approach is followed by others [99, 103].

One design that is intrinsically MISO is Fu et al. [100]. The authors control the utilization of a processor by changing its frequency and the size of its L2 cache. They embed this controller inside an outer loop that uses a linear programming solver to minimize power.

These designs do not address the general case where we want to control multiple outputs in a coordinated manner. For example, having three controllers, one for power, one for performance, and one for utilization is not optimal. These controllers may end up working against each other as follows. To keep the average utilization high, the utilization controller consolidates the work into a few cores and power-gates the rest; the resulting workload thrashing in the cache lowers the performance, which causes the performance controller to increase the frequency; then, the power goes over the limit, which causes the power controller to reduce the frequency and spread the workload into more cores. The cycle then repeats. Overall, the system runs inefficiently and may violate constraints.

Figure 2.1(b) shows our goal, MIMO control: multiple inputs and multiple outputs [91]. The example controller senses the frame rate and power of a processor, compares them to their target values, and then actuates multiple inputs (the processor's frequency, issue width, and load/store queue size) to ensure both frame rate and power targets are satisfied. Each of the inputs impacts each of the outputs.

The MIMO approach enables designers to rank the relative importance of the different outputs. For example, they can declare that the power target is more critical, and the controller will ensure that power errors are minimal. Most importantly, this approach ensures the coordinated control of the multiple outputs. The result is more effective control in a resource-constrained era.

2.2 OVERVIEW OF MIMO CONTROL THEORY FOR PROCESSORS

We take a processor as our system, and abstract it as a controlled system as in Figure 2.1(a). The system is characterized by state x, inputs u, and outputs y. They are all a function of time T. The system state x is given as an N-dimensional vector. We assume we have I inputs (e.g., frequency, issue width, and ld/st queue size) and O outputs (e.g., power and frame rate). These measures are related as follows [91]:

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

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

where A, B, C, and D are matrices that characterize the processor. They are obtained from an analytical model of the processor or from measurements with programs running on the processor (system identification). A is the evolution matrix, and is $N{\times}N$; B is the impact of inputs on the state, and is $N{\times}I$; C is state-to-output conversion, and is $O{\times}N$; finally, D denotes the feed-through of inputs to outputs, and is $O{\times}I$.

The unpredictability component of the system is represented as two matrices [91]. One encapsulates the non-determinism of the system, possibly caused by effects such as interrupts and unexpected program behavior changes. The other matrix encapsulates the noise in reading the outputs — e.g., due to inaccuracies in the power sensor. These two effects are shown in the augmented feedback control loop of Figure 2.2 (together with an uncertainty effect that we discuss later).

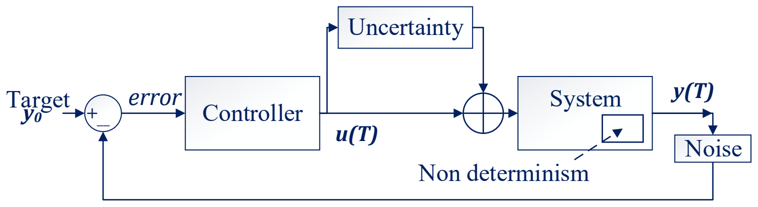


Figure 2.2: Augmented feedback control loop.

To control this system, we can use a type of MIMO controller called Linear Quadratic Gaussian (LQG) controller [91,105]. The LQG controller generates the system inputs, $u(T)$, based on the state of the system, $x(T)$, and the difference in the outputs of the system from their target (i.e., reference) values. However, as the system's true state is not known, the controller begins with a state estimate and generates the system inputs based on this estimate. The controller refines the estimate and learns the true state by comparing the output predicted using the state estimate and the true output. Both estimation and system input generation happen simultaneously and their accuracy increases with time. The design of the LQG controller guarantees that the estimated state converges to the unknown true state soon and, therefore, the appropriate input values are generated to stabilize the system's outputs at their target values quickly.

Specifically, an LQG controller tries to minimize the sum of the squares of a set of costs (also called errors). Such errors are the differences between each output and its target value, and between each input and the proposed new value of that input — the controller minimizes input changes to avoid quick jerks from steady state. These errors can be given architectural meanings. Moreover, the designer can add weights to each of these errors: the higher the weight, the more important is for that error to be small. For example, the designer can give a high weight to power errors. These weights are given by the designer in two positive diagonal matrices [91]: the Tracking Error Cost matrix (Q) for the outputs, and the Control Effort Cost matrix (R) for the inputs. We give the architecture insights in Chapter 3.

LQG control also allows a system to be characterized as the combination of a deterministic part and an unknown part that follows a Gaussian distribution. As indicated above, this also matches architectural environments, which include unpredictable effects. Finally, LQG control is simple and intuitive. It has a low overhead because it only performs simple matrix vector multiplications at runtime.

Since a processor is a very complex system, even models generated by experimenting with many applications will be incorrect in some cases. Hence, we add an uncertainty factor to the model. In practice, this means adding an additional guardband to the parameters generated by the control theory calculations. This is shown in Figure 2.2 as an extra path that perturbs the system in a random way.

Then, we perform robustness analysis [91] to ensure that the controller will work correctly with this level of uncertainty. Robustness analysis is a type of mathematical analysis that analyzes every type of uncertainty that is possible in the system (e.g., non-linearity or time variance) and, for a given bound on the size of this uncertainty, determines whether the system will be stable.

While LQG controllers are good for standalone systems, they are not optimized to work in multilayer or distributed environments. The main reason is that the LQG controllers are not natively optimized to work with only partial models of the system. Unfortunately, in multilayer or heterogeneous environments, a designer has a model only of their own subsystem, and which could be inaccurate because of the unknown interference from other layers. Therefore, we next describe Robust control, which can address this limitation.

2.3 ROBUST CONTROL FOR UNCERTAIN ENVIRONMENTS

The MIMO approach is the most applicable to computer architecture, since multiple goals (performance, power) are typically coupled with each other, and depend on each of the inputs. However, computers are complex, and program behavior is determined by many factors. As a result, controlling computer environments intrinsically involves dealing with uncertain dynamics and approximate models.

A branch of control theory that focuses on hard-to-predict environments is Robust control theory [91]. In this field, variability and uncertainty of the system dynamics at runtime is an integral part of the controller synthesis process. Among the robust controller methodologies, one of the most mature and better understood, with standard packages and tools, is *Structured Singular Value (SSV)* control [91, 106, 107, 108].

SSV controllers have four traits that make them desirable in uncertain environments such as computers. One is the ability to take-in external signals at runtime. The other three are the ability to accept designer-specified (i) bounds on the deviations of the outputs from their targets, (ii) uncertainty guardbands, and (iii) descriptions of the allowed discrete settings for the inputs. We consider each in turn.

First, SSV controllers can read at runtime an additional type of signals called *External Signals*, unlike other formal controllers. These signals provide information on measures that the controller cannot directly change, but this information helps the controller make better decisions. For example, a DVFS hardware controller may take, as an external signal, the number of active application threads from the OS.

Second, designers can specify bounds on the allowed deviation of the outputs from their targets or goals. The controller guarantees that the output values will be within these bounds — subject to the existence of inputs that generate such output values. This is in contrast to non-robust controllers, which generally try to keep the outputs close to the targets, but cannot guarantee any bounds.

Third, when building robust controllers, the designer specifies model *uncertainty guardbands*. They are typically expressed in percentages. For example, a 20% uncertainty means that, due to limitations of the model or other unanticipated effects, the values of the outputs can possibly be ±20% different than predicted by the model.

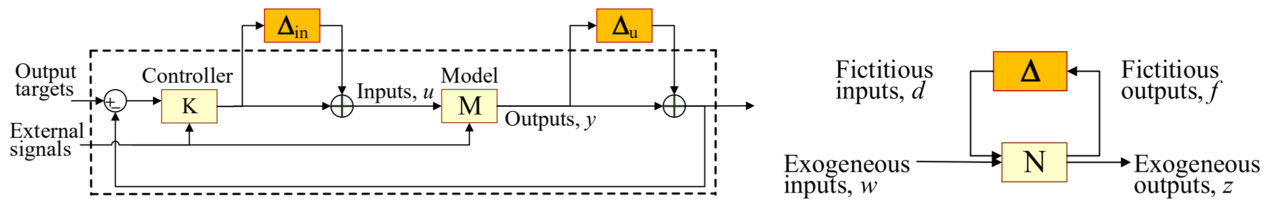
The designer sets the uncertainty guardband based on a combination of suggestions from theory, system insight, and actual experimentation. Guardbands enable the controller to work correctly in scenarios that are very different from modeled executions. Unlike non-robust controllers, SSV controllers do not become slow unless significantly large guardbands (e.g., over 400%) are used [107]. On the other hand, if the guardband is not large enough and is exhausted at runtime, the controller detects it dynamically, and may no longer provide all the guarantees expected.

Finally, robust controllers accept a description of the allowed input settings. The designer can specify the range of values taken by the inputs and their discrete values (saturation and quantization). This is in contrast to typical non-robust controllers, such as PID controllers [91], where each input is assumed to take values that are continuous and unbounded. This makes SSV controllers natively applicable to computing systems, which have discrete resources.

2.3.1 Mathematical Theory of SSV Controllers

Figure 2.3a shows the representation of a system when designing an SSV controller. M is the model of the system that we want to control. M describes how the inputs (u) and external signals generate the outputs (y). K is the controller we have to design. In practice, there are real world inaccuracies shown as Δ in the figure. One is due to the true system behavior deviating from the model (Δ_u), caused by any behavior of the system not captured by the model. This is the model uncertainty for which we specify guardbands. Another constraint is due to the inputs taking only a discrete (or quantized) and limited (or saturated) set of allowed values (Δ_{in}) instead of unlimited values. This is the input discretization. The external signals may also have such effects, but we omit them in this discussion.

The system inside the dotted line boundary in Figure 2.3a is called the nominal closed loop because it contains the components without any imprecisions. Consolidating the individual Δ components into an overall Δ, and denoting the nominal closed loop of Figure 2.3a as N, we can represent the system as Figure 2.3b. In this figure, signals generated from elsewhere (i.e., external signals and output targets or references) are called exogenous inputs (w). The outputs of the system that can actually be measured outside are called exogenous outputs (z). The Δ block interacts with the system through fictitious signals called perturbation inputs (d) and perturbation outputs (f) that capture the effects of model uncertainty and discrete inputs.



(a) System representation when designing an SSV controller. (b) The Δ-N representation of the problem.

Figure 2.3: Mathematical SSV controller design

The controller K in the closed loop N is robust if it: (i) keeps N stable, (ii) generates optimal inputs according to designer-specified input weights W, and (iii) keeps all visible outputs z within bounds B of the targets – for all possible model inaccuracies smaller than the specified Δ. Robust control theory [91] uses the Structural Singular Value (SSV) defined as follows to assess a controller's robustness:

$$SSV(N, \Delta, B, W) = \frac{1}{\min\{s \,|\, det(I - s \times N \times [\Delta; B^{-1}; W^{-1}]) = 0\}} \tag{2.3}$$

where $[\Delta; B^{-1}; W^{-1}]$ is a diagonal matrix with the inaccuracies (Δ), the inverse of the bounds (B), and the inverse of the input weights (W) in the diagonal; N is the closed-loop matrix that gives the outputs (z, f) as a function of the inputs (w, d); and I is the identity matrix. Finally, s is a factor that makes the determinant (det) of $I - s \times N \times [\Delta; B^{-1}; W^{-1}]$ equal to zero.

Physically, s is a scaling factor that multiplies the Δ, $1/B$, and $1/W$ given by the designer. The minimum scaling factor $min(s)$ gives the worst-case inaccuracy ($min(s) \times \Delta$) that the controller tolerates, the worst-case bounds ($1/min(s) \times B$) that it provides, and the worst-case weights ($1/min(s) \times W$) that it supports. So, if $min(s)$ is larger than 1, it means that the controller can handle the Δ, B, and W requested by the designer. On the other hand, if *min(s)* is smaller than 1, the controller is not robust; the specified Δ is too large, the specified B is too small, and/or the specified W is too small.

To design an SSV controller (K), the designer specifies the model of the system (M), the set of Δ values to tolerate, and the desired B and W values. Then, MATLAB selects an initial controller and solves Equation 2.3 to find its *min(s)*. If the *min(s)* value is smaller than 1, MATLAB changes the controller, and then looks for the new *min(s)* for the new controller. MATLAB continues this search until it finds a controller with a *min(s)* value that is as close as possible to (and higher than) 1, which will make $SSV(N, \Delta, B, W)$ as close as possible to (and lower than) 1. If MATLAB cannot find a controller with such a *min(s)* value, the designer selects lower Δ, $1/B$, and $1/W$ values, and restarts.

Compare this approach to the design of a non-SSV controller such as a PID controller [91]. In such case, the designer can only specify the model M and obtains a controller K. There is no way to specify inaccuracies Δ, bounds B, and weights W in the controller design. As a result, such controllers are less useful in complex multilayer environments like computing systems.

The operation of the SSV controller is listed in the equations below. It is a simple state machine, characterized by the dimensionality of its state (N), and the number of inputs (I), outputs (O), and external signals (E).

$$x(T+1) = A \times x(T) + B \times \Delta y(T) \tag{2.4}$$

$$u(T) = C \times x(T) + D \times \Delta y(T) \tag{2.5}$$

where x is the state of the controller (N-entry vector), Δy is the external signals and the deviation of outputs from their targets (vector of $O{+}E$ entries), u is the new inputs (I-entry vector), A is the controller evolution matrix ($N{\times}N$), B is the matrix of impact of output deviations on the state ($N{\times}(O{+}E)$), C is the state-to-input conversion matrix ($I{\times}N$), and D is the matrix of feed-through of output deviations to inputs ($I{\times}(O{+}E)$).

2.4 SYSTEM IDENTIFICATION

To design formal controllers, we need models of the systems that the controllers should regulate. We use the black box system identification approach to obtain our models [109]. This black-box methodology involves running a training set of applications on the system we want to control, while setting the signals that would be actuated by the controller (i.e., the inputs) in a variety of ways, and recording the changes in the signals that would be observed by the controller (i.e., the outputs). When we want to design an SSV controller that reads external signals, we also change the external signals along with the inputs.

Then, the input and output data is passed to MATLAB to obtain a dynamic model of the system of the following form:

$$\begin{aligned} y(T) = a_1 \times y(T-1) + \ldots + a_m \times y(T-m) + \\ b_1 \times u(T) + \ldots + b_n \times u(T-n+1) \end{aligned} \tag{2.6}$$

In this equation, $y(T)$ and $u(T)$ denote the outputs and inputs at time T. This model describes outputs at any time T as a function of the m past outputs, and the current and *n-1* past inputs (and the external signals, if present). The constants a_i and b_i are obtained by least squares minimization from the experimental data [109].

The system identification methodology is widely used, and captures many subtleties of the input-output dependencies using targeted tests.

2.5 TAXONOMY OF CONTROLLERS

Table 2.1 presents our taxonomy of designs available from control theory.

Table 2.1: Space of control-theoretic design choices.

Modeling	White Box (Analytical), Black Box (Data Driven), Gray Box
Mode	SISO, MISO, SIMO, MIMO
Organization	Decoupled, Centralized, Cascaded, Collaborative, Hierarchical
Approach	Classical, Robust, Gain Scheduling, Adaptive
Type	PID, LQG, MPC, SSV

Modeling: A model of the system can be obtained with analytical principles (white box), experimental data (black box), or a combination of both (gray box). Black box models are best when the system internals are unknown or too complicated to describe, as in computers.
Mode: We have four modes depending on the number of inputs that a controller actuates and the number of outputs it monitors. These are: SISO (Single Input Single Output), MISO (Multiple Input Single Output), SIMO (Single Input Multiple Output) and MIMO (Multiple Input Multiple Output). MIMO controllers are most relevant for computers because we target multiple tightly-coupled goals that depend on multiple inputs.
Organization: For multilayer or distributed systems, Decoupled or Centralized designs cannot achieve modularity and coordination simultaneously. In a Cascaded design [61],

controllers are organized as a nested loop, where each controller sets the targets for the immediately inner one. Only the innermost controller changes the system inputs. This method is also suboptimal. In a Collaborative architecture, independent controllers communicate to attain coordination and it is best suited for multilayer control.

Within a layer such as the hardware, there is a natural hierarchy in the system organization such as the core, chip and the compute node. Here, the best organization is Hierarchical where local controllers manage each component and constraints are propagated across the hierarchy.

Approach: There are several approaches used to ensure that the controller works correctly under uncertainty or highly-changing conditions. The Classical one is to design controllers with additional stability margins [110]. This works for simple systems. Robust control explicitly optimizes controllers for large uncertainty, and is applicable to computer environments [91]. The controllers have low complexity and low overheads. In Gain Scheduling, multiple controllers are used — each suited for a particular type of execution [60]. At runtime, some logic chooses when each of the controllers is active, based on the execution. This approach requires additional modeling efforts and expensive selection logic at runtime. Lastly, Adaptive control synthesizes a new controller online whenever changing conditions are detected [67]. It has higher runtime overhead.

Type: PID controllers are popularly used for their simplicity, but are not useful to control MIMO systems. For standalone MIMO systems, LQG controllers (Section 2.2) and Model Predictive Controllers (MPC) [111] are applicable. However, these controllers are not suitable for multilayer or decentralized environments. This is because they are not natively optimized for uncertainty and, instead, trade-off optimality and fast response time for robustness. Moreover, they do not have channels to communicate among controllers. In multi-controller environments, controllers need to share information between themselves for coordination. SSV controllers are optimized to work in uncertain environments and can read communication signals from other controllers (Section 2.3). This makes them suitable for systems that require multiple controllers to operate simultaneously.

2.6 COMPARING APPROACHES FOR ADAPTATION

Computer architects use several approaches to perform architecture parameter tuning. We broadly classify them into optimization, machine learning, control theory, model-based heuristics, and rule-based heuristics. Table 2.2 compares these approaches, outlining their problem formulation, design and tuning method, advantages, and shortcomings.

Most of the entries in the table are self explanatory. From the advantages and shortcomings

Table 2.2: Comparing approaches for architecture tuning.

Approach	Problem formulation	Example	Design and tuning	Advantages	Shortcomings
Static Optimization [112, 113]	Minimize an objective subject to constraints. The objective is a f(inputs).	Objective: power. Constraint: IPC $>$ k.	Obtain a model of f(). Use solvers at runtime to obtain the inputs.	1) Natural choice for architecture. 2) Expressive.	1) Model needs to be close to reality and convex. 2) No feedback.
Machine learning [59, 114]	$\text{input}_i = \max([\text{weights}]_{v \times o} \times [\text{features}]_o)$, where v is the # of values for input_i, and o is the # of features.	Input: frequency. Features: power, misses/Kinstr.	Tune weights by specifying the best set of input values and associated feature values.	1) Data driven identification of relationships. 2) Formal reasoning and methodology.	1) Hard to add feedback. 2) No guarantees. 3) Requires exhaustive enumeration during training.
Control theory [99, 101]	Change inputs to make outputs approach target values, where outputs[T] = f(inputs[T,T-1,...],outputs[T-1,...])	Input: frequency. Output: IPC. Target value: QoS_o.	Obtain the f() for outputs[T]. Specify several design requirements.	1) Provides guarantees. 2) Learns from feedback. 3) Formal reasoning and methodology.	1) Hard to obtain model. 2) Specifying target values is not obvious.
Model-based heuristics [55, 115]	Use a model to guide decisions. The model relates outputs, inputs and auxiliary outputs.	Output: power. Aux output: misses/Kinstr. Input: frequency.	Find the model. Use insight to develop rules on top of the model. Train rules to set thresholds.	1) Model simplifies decision making.	1) No guarantees. 2) No formal methodology. 3) Hard to add learning. 4) Prone to errors. 5) Hard to deal with multiple inputs and/or outputs.
Rule-based heuristics [116, 117]	Encode in an algorithm decisions to choose inputs based on outputs and auxiliary outputs.		Select rules. Train rules to set thresholds.	1) Lightweight.	

columns, we see that the control theory approach is the only one that: 1) uses feedback to learn automatically at runtime, and 2) provides three guarantees. The guarantees are *Convergence*, *Stability*, and *Optimality* [91]. Informally, these guarantees mean the following. Convergence means that, if it is possible for the outputs to take the target values, then they will eventually reach them. Stability means that, once the outputs converge to their targets or to the closest they can get to them, then they exhibit no oscillatory behavior. Optimality means that the final state of the system is optimal according to the cost function specified by the architect.

In spite of control theory's advantages, at least two issues have limited its use in architecture. First, it needs a model of the system that gives the current output values as a function of the current input values and some history of input and output values. This is hard to obtain analytically for computers. Second, control theory approaches assume that the target values for the outputs are specified by a higher entity. This might be easy in some cases, such as when targeting a Quality of Service (QoS) requirement, but it is not so when trying to optimize a metric such as the product of energy and delay ($E \times D$, or EDP—energy delay product).

We also note two limitations specific to the control theory choices we use here. First, for MIMO to be effective, the number of outputs cannot be more than the number of inputs. Second, it is not straightforward to specify a target as "be less/more than this value"; it is easier to specify it as "attain this value".

Finally, a general limitation of all of these approaches that use models or history (e.g., control theory and machine learning) is that they are reactive. This means that they respond only after observing changes in the outputs they monitor. Hence, when the system conditions change abruptly, the monitored outputs may move away from their expected levels before the controllers actuate.

CHAPTER 3: USING MULTIPLE INPUT, MULTIPLE OUTPUT FORMAL CONTROL TO MAXIMIZE RESOURCE EFFICIENCY IN ARCHITECTURES

As processors seek higher efficiency, they increasingly need to target multiple goals at the same time, such as certain levels of performance, power consumption, and average utilization. Unfortunately, most systems today use heuristic control, which is suboptimal, non-robust, and unsafe. Recent processors like Intel's Skylake [1], AMD's Zeppelin [118] and IBM's Power9 [46] have begun using simple formal controllers that manage a single outcome, like power, by changing a single parameter, like frequency. However, processors must meet multiple interrelated goals, and can best do so by changing multiple parameters at the same time.

3.1 MIMO CONTROLLERS FOR PROCESSORS

We now propose how to design MIMO controllers for processors. We first outline the steps, and then explain in detail the steps that require architectural insight.

3.1.1 Steps in Controller Design

Figure 3.1 shows the proposed process to build a MIMO controller, with hexagons showing steps that need architectural insight. First, we select the outputs to be controlled and the inputs that can be manipulated. Then, using architectural insights, we decide on the relative importance of the outputs (to generate the Q matrix), the relative overheads of the inputs (to generate the R matrix), and the strategy for modeling the system. The latter mainly involves choosing how to model the system (analytically or experimentally) and the number of dimensions of the system state x.

We model the system experimentally, performing the experiments for black-box system identification [109], described in Section 2.4. We pass the experimental data to a least square solver for a dynamic environment (run with MATLAB) and generate the A, B, C, D, and two unpredictability matrices. These matrices constitute the model. We pass this model plus the Q and R matrices to a constraint optimization solver (also run with MATLAB) to generate the controller. The controller is encoded as a set of matrices that can produce the changes in the manipulated inputs on observing tracking errors in the controlled outputs.

Next, we validate the model by running additional programs on both the model and the real system. Based on the observed differences, we estimate the model error and, using

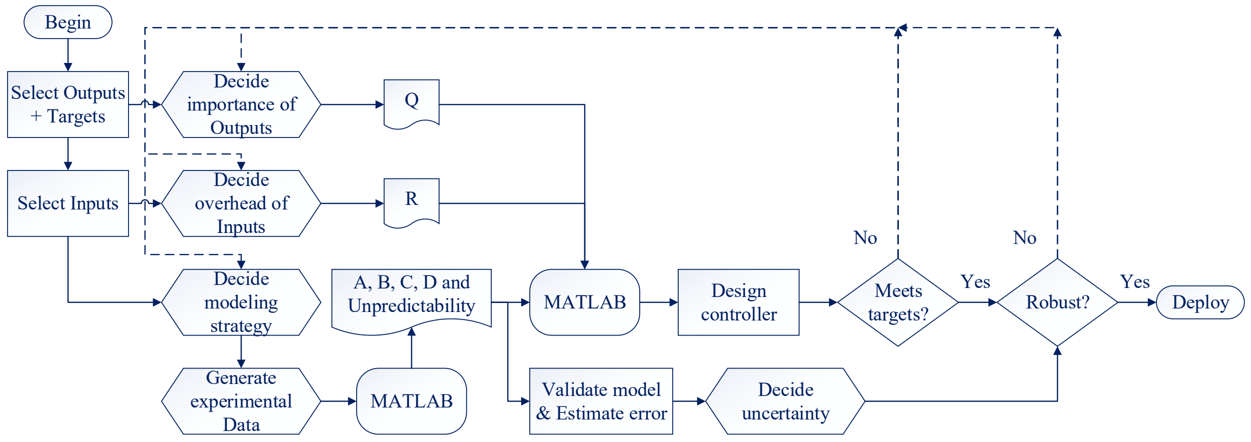


Figure 3.1: Proposed flowchart of the process of building a MIMO controller. Architecture-level insight is especially needed in the hexagonal steps.

architectural insights, we set the uncertainty of this model. The uncertainty will be used in the validation step.

Finally, we proceed to validating the controller in two steps. First, we check if the controller meets the targets (i.e., it brings the outputs to the targets and does it fast enough). Then, we use Robust stability analysis [91] to see if, for the worst case of estimated uncertainty, the system is still stable. We describe this process later. If any of these two checks fail, we change the initial decisions and repeat the process. Otherwise, we have the final controller.

3.1.2 Detailed Controller Design

Modeling the System

While there are analytical models of processor performance and power [119, 120, 121, 122], they do not capture the dynamics of the system at the level we need, and are not amenable for formal controller design. To control the system, we need models that describe the relation between inputs and outputs as a function of time. Hence, we build an experimental model using the black-box identification technique of system identification theory [109, 123] (Section 2.4).

Building the Cost Matrices Q and R

The Tracking Error Cost matrix (Q) contains a positive weight for each output, and the Control Effort Cost matrix (R) a positive weight for each input. Intuitively, the output weights represent how important it is for a given output not to deviate from that output's target. The input weights represent how reluctant the controller should be to change a given input from its current value (due to its overhead). These two matrices are set by the designer. Let us consider the architectural implications.

Consider Q first. For outputs that have a high weight, the controller is more reluctant to actuate inputs in a manner that changes those outputs away from their target values. Consequently, we assign the highest output weights to architecture measures that are critical to correctness. On the other hand, we assign lower weights to architecture measures that determine result quality or performance. This is because if these measures veer off their target values, the system still functions acceptably.

Row 2 of Table 3.1 shows a sample of architectural measures used as outputs, with a possible weight order from higher to lower. Outputs such as voltage guardbands and temperature limits have the highest weight. Intermediate weights can be assigned to power, utilization, or energy. Lower weights can go to various measures of performance, as long as the performance is acceptable.

Table 3.1: Qualitative weights of architectural measures. Input weights only consider change overheads.

Type of Weight	Qualitative Weight Ranking (From High to Low)
System Outputs	Voltage guardband, temperature, power, core utilization, energy, frame rate, instructions per second (IPS)
System Inputs	Cache power gating, core power gating, frequency, issue width, ld/st queue entries

Consider R now. The controller is more reluctant to change inputs that have a high weight. There are two reasons to be reluctant to change an input. The first one is if changing it has a high energy or performance overhead. For example, power gating a cache has higher overhead than changing the number of load/store queue entries.

A second reason results from the fact that inputs often take discrete values rather than continuous ones — e.g., we only change the frequency in 100 MHz steps. Then, consider an input that can take a large number of discrete values. If we assign a small weight to this input, the controller will generate frequent and large changes in the input's value, jumping over many possible settings, and not utilizing the range of values available for this input. On the other hand, if we assign a higher weight, the controller will be more likely to use smaller steps, utilizing more of the available settings and, hence, using the input for more effective control.

Row 3 of Table 3.1 shows a sample of architectural measures used as inputs with a possible weight order (from higher to lower), taking into account only change overhead. Power gating a component has a high overhead, especially for components with substantial state such as caches. Frequency changes often take a few microseconds. Pipeline changes may require only a pipeline flush or not even that, which is why they have a low weight.

Finally, consider the relative weights of the outputs in Q and the inputs in R. They strongly determine the behavior of the system. Specifically, if the input weights are low relative to the output weights, the controller is willing to change the inputs at the minimum change of outputs — e.g., due to noise. This will create a ripply system. If, instead, the input weights are high relative to the output ones, the system will have inertia: when the output is perturbed, the system will react sluggishly, only after a while.

Figures 3.2(a) and (b) illustrate the two cases. Each figure shows how the output (top) and input (bottom) change with time, relative to the initial conditions. The time starts when the output suffers some positive noise. Figure 3.2(a) shows a ripply system: the input immediately reacts, causing the output to change course and get into negative values. After a few waves, the system stabilizes again. Figure 3.2(b) shows a system with inertia: the input does not react until much later, and with lower intensity. After a while, both output and input return to the stable values. Note that the figures are not drawn to scale.

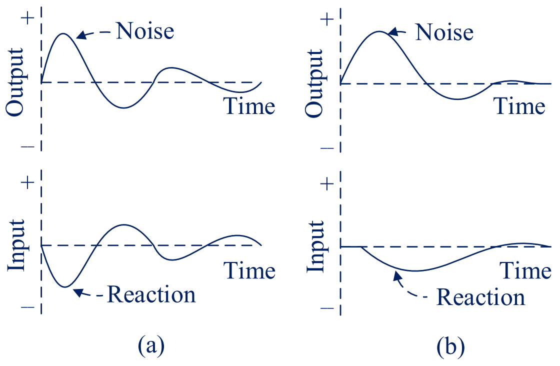


Figure 3.2: System behavior when input weights are low (a) or high (b) relative to output weights.

We avoid systems that are too ripply or too sluggish: they take too long to stabilize or may never do it. Besides this, from an architecture perspective, we note the following. First, when dealing with critical output measures, such as voltage guardbands and temperature, we want the output weights to be relatively high. This will ensure that the system reacts immediately to changing conditions. On the other hand, when we have high-overhead inputs such as power gating large memory structures or process migration, we want the input weights to be relatively high. This will avoid continuous input changes due to noise.

The absolute values of the weights are unimportant; only their relative value matters. The designer uses offline experimentation and his architectural intuition to set them appropriately. As an example, consider a weight of $100\times$ for one output (o_1) over the other (o_2). The relative quadratic cost of a tracking error Δ in these outputs becomes $100\Delta o_1^2$ and Δo_2^2. If they are to matter equally, then we have $100\Delta o_1^2 = \Delta o_2^2$, or $10\Delta o_1 = \Delta o_2$. This means that the controller will not deviate from the reference value for o_1 by 1% unless it can reduce the deviation for o_2 by 10%. Applying a similar analysis for inputs, a relative input weight of 100 means that the controller will change the lower-weight input by up to 10% before changing the higher-weight input by 1%.

Weight selection is an offline procedure used during the design of the controller. It requires an understanding of the underlying system. The LQG methodology ensures that the resulting controller is stable for any choice of weights.

Unpredictability Matrices

When MATLAB's least square solver takes the input and output waveform data and generates the A, B, C, and D matrices, it also generates two unpredictability matrices. These unpredictability matrices encapsulate two effects: one that impacts the state x and one that impacts the outputs y.

The architectural insights are as follows. The first set of effects represents non-deterministic events such as branches, interrupts, page faults and other probabilistic events; the second set of effects represents sensor noise, such as that resulting from inaccurate or coarse-grained sensors.

Uncertainty

Once we have the system model (i.e., the A, B, C, D, Q, R, and unpredictability matrices), we proceed to design the controller. However, as architects, we know that processors are complicated, and unusual applications may exercise corner cases. Hence, we validate the model by running additional, highly compute- and highly memory-intensive applications on both the model and on the real system, and compare the results. Based on the difference, we roughly estimate the uncertainty of the model. For example, we may estimate that, under an unusual application, the model's predictions may be *consistently* (i.e., on average) 20% off from what the real system exhibits.

We want to ensure that the controller, based on a model with this uncertainty, is still stable. Therefore, in the step labeled *Robust* in Figure 3.1, we perform Robust stability analysis

(RSA) [91]. RSA checks whether a perturbation equal in magnitude to the uncertainty, if coming at the worst time and in the worst manner, can make the system unstable — e.g., prevent the output ripples in Figure 3.2(a) from dying down. If the system is unstable, we have to go back and change the inputs to the controller design. In particular, we can use more challenging applications in the initial design, or use lower Q weights relative to R weights, thereby making the system less ripply and more cautious to changes.

Note that, for heuristic algorithms, it is not possible to perform a similar stability analysis. Hence, there is a risk that a heuristic algorithm can fail when it encounters an unusual application that it has not been trained on.

3.1.3 Adding Additional Inputs and/or Outputs

The MIMO methodology allows an easy procedure to re-design the controller when new inputs and/or outputs need to be added. First, the process of system identification is repeated with the new inputs/outputs. Then, weights are chosen for each of the new inputs/outputs. It is not necessary to modify the weights for the existing inputs/outputs. If more outputs are being added, then their uncertainty bounds are measured and specified. The LQG design process automatically generates and tunes the new controller.

3.2 USES OF THE CONTROLLER

There are multiple ways in which a MIMO hardware controller can be used. In this section, we discuss three.

Tracking Multiple Targets

In the simplest use, a high-level agent specifies the target value for each of the multiple outputs. In addition, it can specify the relative importance of each of the outputs and each of the inputs.

Time-Varying Tracking

A more advanced use is when a high-level agent monitors real-time conditions and, based on those, changes the target values that it wants the outputs to track [124]. A typical example is a battery-powered mobile platform that runs a program as the battery is being depleted. The best tradeoff in performance (or quality of service) versus power consumed changes as the battery energy level decreases. For example, while the battery is above a certain level, the output targets are high performance and a tolerably-high power consumption. As the battery level decreases, the OS sets successively lower pairs of performance and power targets to conserve battery life [1,124,125,126,127].

Fast Optimization Leveraging Tracking

A final use is when the high-level agent does not want certain target values for its outputs, but that the value of a combination of the output values is minimized or maximized. For example, given power (P) and performance in instructions per second (IPS), it wants to maximize IPS^2/P, which is to minimize Energy×Delay (EDP). In this case, the controller needs to do some search, but the search is at a high level and very efficient. This is in contrast to a heuristic-based controller, which typically needs to perform a very costly and inefficient low-level search, apart from requiring heavy tuning.

Figure 3.3(a) shows the envisioned system. The outputs are IPS and P, and we are trying to maximize IPS^2/P. The search is driven by an extension to the original controller that we call *Optimizer*. It can be a part of the runtime system or a hardware module.

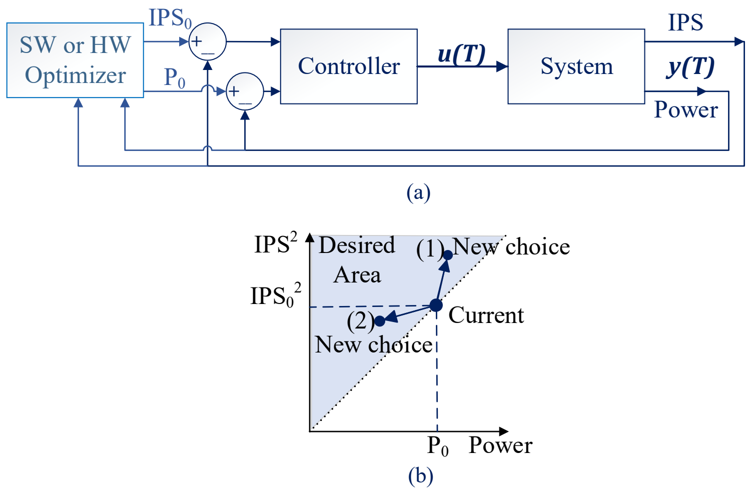


Figure 3.3: Using MIMO to optimize a combination of measures such as $E\times D$.

Initially, the optimizer sets a certain target IPS$_0$ and P$_0$. The base controller then generates the input configuration (e.g., frequency, issue width, and load/store queue size) that attains this target. After the system converges, the optimizer changes the target outputs so that IPS^2/P increases. Specifically, it either increases P a little and increases IPS much more, or decreases IPS a little and decreases P much more. This is shown in Figure 3.3(b), where the original point is called *Current*, and the two possible changes are (1) and (2), respectively. Based on the new (IPS_1, P_1), the base controller regenerates the input configuration.

This process is repeated a few times. Note that, given an original point, the desirable points are those to the left of the line that connects the point to the (0, 0) coordinate. Of course, at any given step, the system may not reach the desired (IPS_i, P_i), and we may end up in a less desirable (IPS, P) point — especially since the inputs and outputs are discrete. In this case, the optimizer does not choose the new point and moves on. Eventually, the optimizer settles into a good $\mathrm{IPS}^2/\mathrm{P}$ point. A new search will start only when the controller detects that the application changes phases.

Overall, we see that the optimizer's search is very efficient. In contrast, a heuristic-based algorithm has to figure out how to change the inputs (frequency, issue width, and load/store queue size) to increase $\mathrm{IPS}^2/\mathrm{P}$. The algorithm is likely to be complicated and non-robust.

3.3 EXAMPLE OF MIMO CONTROL SYSTEM

We describe the design of a MIMO control system for an out-of-order processor using the design flow described earlier. Our system is a processor with, initially, two configurable inputs: (1) the frequency of the core plus L1 cache, and (2) the size of the L1 and L2 caches. The input settings are shown in Table 3.2. The frequency is changed with DVFS. It has 16 different settings, changing from 0.5GHz to 2GHz in 0.1GHz steps. The cache size is changed by power gating one or more ways of the two caches. The associativities of the L2 and L1 caches can be (8,4), (6,3), (4,2), and (2,1). We later add an additional configurable input, namely the reorder buffer (ROB) size. We are interested in two outputs: (1) the power of the processor plus caches, and (2) the performance in billions of instructions committed per second (IPS).

3.3.1 Controller Design

Choosing Input/Output Weights

To assign the input and output weights, we proceed based on the discussion of Section 3.1.2. Specifically, among the outputs, we assign to power a higher weight than to IPS, to minimize power tracking errors and power budget violations. As shown in Table 3.2, we use weights of 1,000:1 for power:IPS, which makes power $\sqrt{1,000}\times$ (or $\approx 30\times$) more important than IPS. In other words, we are willing to trade 1% deviation from the power reference for 30% deviation from the IPS reference.

Table 3.2: Control and architecture parameters.

Controller Parameters	
Input configurations	Frequency: 16 settings 0.5GHz to 2GHz in 0.1GHz steps Cache size: 4 settings L2,L1 associativity: (8,4),(6,3),(4,2),(2,1) ROB size: 8 settings 16 to 128 entries in 16-entry steps
Input/output weights	10,000 for power, 10 for IPS, 0.01 for frequency, 0.0005 for cache size, 0.001 for ROB size
Dimensions of system state	4
Uncertainty guardband	50% for IPS, 30% for power
Controller invocation	Every 50μs
Optimizer Parameters	
Optimizer invocation	Every 10 ms or phase change as in [12]
MaxTries	10
Baseline Core Parameters	
Superscalar	3-issue out of order
ROB; Ld/St queue	48 entries (for $E{\times}D$ opt); 32/16 entries
Branch predictor	38Kb hybrid
Frequency	1.3 GHz (for $E{\times}D$ opt)
DVFS latency	5 μs
Baseline Memory System Parameters	
L1 data cache	32KB, 3-way (for $E{\times}D$ opt), 3 cycles latency, 64B line
L1 instr. cache	32KB, 2-way, 2 cycles latency, 64B line
L2 cache	256KB, 6-way (for $E{\times}D$ opt), 18 cycles lat, 64B line
Main memory	125 cycles latency

Among the inputs, we observe that the overhead of power-gating a cache way, and that of adjusting the frequency by one step are both large and perhaps comparable. However, frequency offers more different settings than cache size (16 settings versus 4). Hence, to ensure that the controller uses all the frequency settings and does not bypass many of them in each adaptation, we choose a higher weight for frequency. As shown in Table 3.2, we use weights of 20:1 for frequency:cache, which makes frequency $\approx$4$\times$ less likely to change in large steps, to account for having 4$\times$ more adaptation settings.

We consider that it is more important for the outputs to remain close to their reference values than to minimize the overheads of changing inputs. Hence, we give higher weights to the outputs than to the inputs. As discussed in Section 3.1.2, if the ratio of output to input weights is too high, the system becomes ripply and takes longer to converge; if it is too low, the inputs are sluggish, and any perturbation also takes long to disappear. We need to experiment with MATLAB to ensure that the chosen ratio falls in between the two scenarios, and hence the system converges reasonably fast. To select the output to input weight ratio, we need to consider the less important output (IPS) and the most important input (frequency). As shown in Table 3.2, we use weights of 1000:1 for IPS:frequency, which makes IPS ≈30× more important than frequency.

Section 3.5.1 performs a sensitivity analysis of the choice of weights for the inputs and outputs.

Model Identification & Uncertainty Analysis

It is challenging to build analytical processor models that can accurately relate processor performance and power with cache size and frequency. Hence, we perform experimental system identification [109,123] of a cycle-level simulator that we wrote to model the processor system. We run four profiling applications from SPEC CPU 2006 on the simulator — two integer (*sjeng* and *gobmk*) and two floating-point (*leslie3d* and *namd*). For each program, we apply test waveforms of cache size and frequency changes at runtime. We record the time variation for the inputs and outputs.

System identification tests are designed to extract the most information from the runs of these training-set applications. With this information, we are able to characterize the system and build a model. We find that a model of dimension 4 is a good tradeoff between accuracy and computation cost (Table 3.2). Section 3.5.2 performs a sensitivity analysis of the number of dimensions. Based on this model, we use MATLAB to construct the first version of the controller.

As per Section 3.1.2, we use uncertainty analysis to revise the design (Figure 3.1). We run two additional applications (*h264ref* and *tonto*) on both the simulator and the model obtained with system identification, changing cache size and frequency signals. The outputs are compared. We find that the *maximum* error in the model is 14% for IPS and 10% for power. Then, we conservatively set the uncertainty guardbands to 3× these values, namely to 50% for IPS and 30% for power (Table 3.2).

Recall from Section 3.1.2 that these uncertainty guardbands refer to the *average* prediction errors across the whole application execution that are tolerable. After choosing the uncertainty guardbands, we run robust stability analysis to see if the system converges. If it does not, we use MATLAB to reconstruct the controller with larger input weights, until the system is shown to converge for the desired guardbands.

Section 3.5.3 performs a sensitivity analysis of the uncertainty guardband.

3.3.2 Optimizer Design

As discussed in Section 3.2, the Optimizer performs a high-level search for the optimal operating point, according to Figure 3.3(b). Depending on our goal, the search can be in the $E{\times}D$, $E{\times}D^2$, ... $E{\times}D^{k-1}$ space. To minimize $E{\times}D^{k-1}$, the algorithm tries to maximize $\mathrm{IPS}^k/\mathrm{P}$.

Every time that the algorithm is invoked, it starts by setting the inputs to their midrange values: 1 GHz frequency and (4,2) associativity for (L2,L1) caches. Then, it makes a move in one of the two directions in Figure 3.3, namely "Up" (higher IPS but only slightly higher power) or "Down" (slightly lower IPS and much lower power). If the resulting value of the measure $\mathrm{IPS}^k/\mathrm{P}$ is higher than the previous one, the algorithm continues to explore more points in the same direction. Otherwise, it reverses the search direction. This process repeats for a fixed number of trials (*MaxTries* as shown in Table 3.2). We do not use backtracking in this algorithm.

3.3.3 Overheads of the Design

Both controller and optimizer operation cause very minor overheads. The controller is invoked every 50 μs, and operates entirely in hardware. It reads performance and power counters, and computes the difference between the values and their references. It then performs four floating-point vector-matrix multiplies, and generates the actuations on cache and frequency. The controller only stores less than 100 floating-point numbers. The optimizer is invoked every 10 ms or when there is a phase change as detected in [12]. It also runs in hardware. For this evaluation, we have assumed a system that can change the DVFS level in 5 μs. While aggressive by today's standards, such capability may be reachable in the near future.

3.3.4 Adding an Additional Input

To show the flexibility of MIMO control, in a second set of experiments, we augment the controller by adding a third configurable input: the size of the reorder buffer (ROB). The ROB size is changed by power gating 16 entries at a time, as described in [128]. Since the full ROB has 128 entries, we have 8 different ROB sizes (Table 3.2).

We repeat the system identification process with the same application training set, now including ROB size changes. To set the input weight for the ROB resizing, we note that ROB resizing has less overhead than cache resizing or frequency changes. Hence, it should have a low weight. However, since it has more settings that cache resizing, we give it a slightly higher weight. Hence, we use weights 2:1 for ROB:cache resizing. We place the same uncertainty guardbands as before and do not change the weights for the other inputs/outputs. Since we do not change outputs, we reuse the optimizer.

3.4 EVALUATION METHODOLOGY

3.4.1 Infrastructure

We base our evaluation on simulations of a processor like the ARM Cortex-A15 modeled with the ESESC simulator [129]. The architecture parameters optimized for the *best energy delay product in the baseline architecture* are listed in Table 3.2. We modified ESESC to model the configurable inputs and implement the hardware controller and optimizer. Power estimates are obtained from the McPAT modules that are integrated within ESESC. We use CACTI 6.0 for cache power estimates. DVFS pairs are obtained from interpolating published A15 DVFS values [130]. We use MATLAB's System Identification and Robust Control Toolboxes [131] for system identification, LQG controller design, tuning, and robustness evaluation. We run all the SPEC CPU 2006 applications except *zeusmp*, which our infrastructure does not support. We group the applications into a training set (*sjeng*, *gobmk*, *leslie3d*, and *namd*) and a production set (the remaining ones). Each application is monitored for an average of 50 billion instructions, after fast forwarding 10 billion instructions.

3.4.2 Experiments

Tracking Multiple Targets

The goal of this experiment is for the outputs to track target values. Specifically, we target 2.5 BIPS for IPS and 2 W for power. These values are obtained by performing a design space exploration on the training set applications, and picking output values that minimize the average $E \times D$ for them. This IPS target is infeasible for highly memory-bound applications. Hence, we will show results separately for such applications.

Time-Varying Tracking

The goal of this experiment is for the outputs to track time-changing target values. As indicated before, an example is when a high-level agent throttles performance and limits power consumption based on operating conditions such as battery levels [125,126]. We model such a scenario by changing the IPS and power targets based on the recently-introduced Quality of Experience (QoE) parameter in handheld devices [127]. We use the analytical models for QoE and battery charge consumption in [127] to compute how the targets should be changed. We set the time betweeen changes to 2,000 epochs of 50 μs each, and the total energy supply to 1 J.

Fast Optimization Leveraging Tracking

The goal of this experiment is to generate outputs that minimize energy (E), $E{\times}D$, or $E{\times}D^2$. For the optimization (Figure 3.3(b)), the optimizer can try at most *MaxTries* trials.

3.4.3 Architectures Evaluated

We compare the four architectures of Table 3.3. *Baseline* is a non-configurable architecture where the inputs are fixed and chosen to deliver the best outputs. Specifically, we profile the training set applications and find the cache size, frequency, (and ROB size for the 3-input experiments) that deliver the best output — E, $E{\times}D$, $E{\times}D^2$, etc, depending on the experiment.

Table 3.3: Architectures compared.

Baseline	Not configurable. Inputs fixed and chosen for best output
Heuristic	Configurable with a coordinated-heuristics controller
Decoupled	Configurable with decoupled SISO controllers
MIMO	Configurable with our proposed MIMO controller

The other designs are our configurable architecture with different hardware controller algorithms to drive input adaptations. *Heuristic* uses a sophisticated heuristics-based algorithm similar to [63], which is tuned with the training set applications. The algorithm has two steps. First, it ranks the adaptive features (cache size, frequency, and ROB size) according to their expected impact in this application, like [12]. The second step depends on the experiment performed.

In tracking experiments, the second step involves taking different actions, using the ranked features in order, depending on the difference (magnitude and sign) between each output value and its reference value. These actions are qualified by threshold values experimentally determined. In the optimization experiments, the second step involves searching the space (e.g., $E \times D^2$) using an iterative process, testing a few configurations of each of the adaptive features in rank order. This is similar to earlier schemes [63, 78, 115, 132].

Details of *Heuristic* were released as a technical report [93]. Note that, for *Heuristic*, the algorithms developed and tuned for the two-input system (cache size and frequency) have to be completely redesigned from scratch for the three-input system (cache size, frequency, and ROB size).

Decoupled uses two formally designed SISO controllers. One changes cache size to control IPS, and the other changes frequency to control power. There is no coordination between the two. The optimizer works as the MIMO optimizer. Note that we cannot use *Decoupled* in the three-input experiments.

MIMO uses our MIMO controller and optimizer. The optimizers in all the architectures are limited to trying at most *MaxTries* trials per search.

3.5 RESULTS

This section evaluates our four architectures.

3.5.1 Impact of Input and Output Weights

To see the impact of input and output weights, we run the MIMO controller with the different sets of weights in Table 3.4, tracking 2.5 BIPS for IPS and 2 W for power (P). For the application *namd*, Figure 3.4 shows the epochs taken to achieve steady state (a), and the output tracking errors (b) — i.e., the difference between output values at steady state and their reference values. In all cases, we initialize the system with the same input values, which are 20% and 30% different than the reference IPS and power values, respectively.

Table 3.4: Different sets of weight choices.

Label	Description	[W_{cache} W_{freq} W_{IPS} W_P]
Equal	Same weights inputs & outputs	[1 1 1 1]
Inputs	Lower weights for inputs	[0.01 0.01 1 1]
Power	Higher weight for power	[0.01 0.01 1 100]
Size	Lower weight for cache size	[0.001 0.01 1 100]

In *Equal*, all inputs and outputs have the same weight. In this case, the relatively high input weights make the controller reluctant to change inputs significantly. The controller makes only small input changes, many of which are rounded to zero. As a result, for the

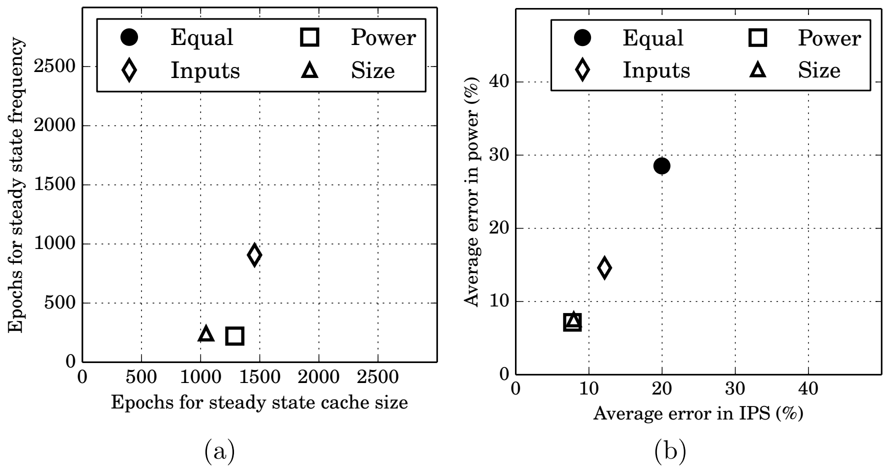


Figure 3.4: Epochs to achieve steady state (a) and output tracking errors (b) for different weight choices.

duration of our experiment, the outputs do not converge yet. Hence, the *Equal* datapoint is missing in Figure 3.4(a), and has not moved from initial conditions in Figure 3.4(b). In *Inputs*, minimizing input changes is less important than meeting the output targets. Hence, the output tracking errors decrease (Figure 3.4(b)) and the system converges within the measured time (Figure 3.4(a)).

In *Power*, P has a higher weight and, hence, tracking P has a higher priority. The resulting controller reduces the P tracking error to less than 10%, and the IPS error also comes down as a side-effect. Fewer epochs are needed for steady state. Finally, in *Size*, by choosing a lower weight for the cache size, cache size changes are favored over frequency changes. Consequently, the steady state cache size is reached faster, without changing the output tracking errors.

3.5.2 Impact of Model Dimension

The number of model dimensions is a tradeoff between accuracy and computation overhead. With more model dimensions, we model the true system more accurately, but the controller requires more computations. In practice, for our small system, computation overhead is not a concern. Still, we would like to use as few dimensions as possible while retaining accuracy. Hence, we compare the IPS and P attained by the true system (i.e., the simulator) and our model with different dimensions. We refer to the difference as the error. Figure 3.5 shows the maximum errors for different dimensions. Based on this result, we use a model dimension of 4.

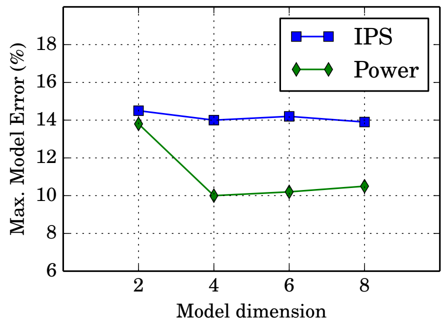


Figure 3.5: Maximum prediction errors for different model dimensions.

3.5.3 Impact of Uncertainty Guardband

The size of the uncertainty guardband is a tradeoff between the time to attain steady state and the risk of system instability. If we bet that production applications will behave more like the training applications and, hence, a smaller uncertainty guardband is acceptable, we can reduce the input weights. Then, the system will reach the steady state faster. However, if a production application deviates more than we expected, the system will become too ripply and not reach steady state. As per Section 3.3.1, to design our controller, we use uncertainty guardbands equal to 50% for IPS and 30% for power. Figure 3.6 shows the resulting number of epochs needed to reach steady state with our controller (*High Uncertainty*). It also shows the number needed if we had used a more aggressive design with lower uncertainty guardbands equal to 30% for IPS and 20% for power (*Low Uncertainty*). From the figure, we see that the more aggressive design is still stable — and hence needs fewer epochs to achieve steady state. Hence, our controller design is conservative.

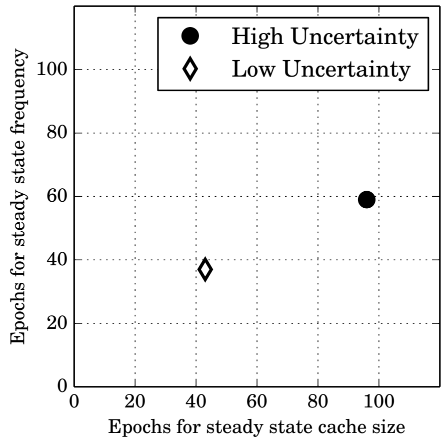
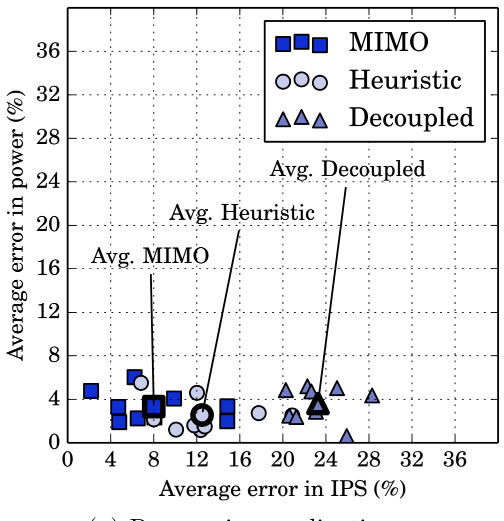


Figure 3.6: Time to achieve steady state for different uncertainty guardbands.

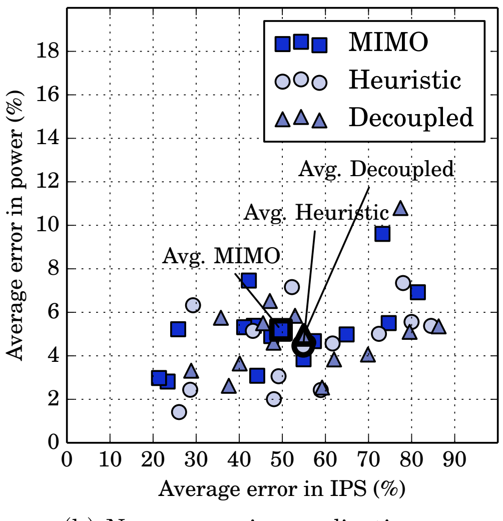
3.5.4 Using MIMO for Tracking Multiple Targets

We compare how effectively *MIMO*, *Heuristic*, and *Decoupled* can track multiple output target values — specifically, 2.5 BIPS for IPS and 2 W for *P*. As indicated in Section 3.4.2, this IPS reference value is high, and several memory-bound applications cannot reach it. For these applications, which we call *Non-responsive*, no amount of control can get IPS and *P* very close to their targets. For the rest, which we call *Responsive* applications, different control architectures have different effectiveness. The non-responsive applications are *bzip2*, *gcc*, *hmmer*, *h264ref*, *libquantum*, *mcf*, *omnetpp*, *perlbench*, *Xalan*, *bwaves*, *dealII*, *GemsFDTD*, *lbm*, and *soplex*.

Figure 3.7(a) and (b) show the average error in IPS and *P* for the responsive and non-responsive applications, respectively, for the *MIMO*, *Heuristic*, and *Decoupled* architectures. For each architecture, the figures show a small data point for each application and a large datapoint for the average of all the applications.



(a) Responsive applications.



(b) Non-responsive applications.

Figure 3.7: Results for tracking multiple targets.

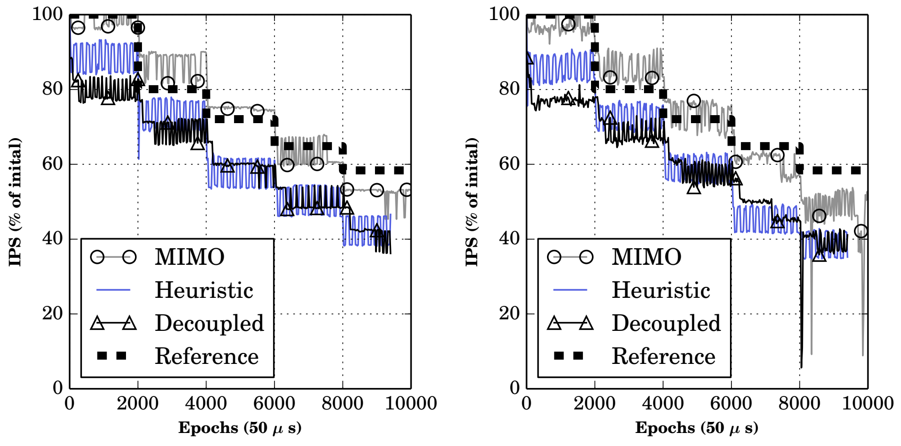
Focusing on the responsive applications, we see that while all three architectures result in good power tracking, they differ in IPS tracking. The average IPS error is 7%, 13%, and 24% for *MIMO*, *Heuristic*, and *Decoupled*, respectively. *MIMO* works best, as it can learn and adapt to the runtime characteristics of the workload. *Decoupled* has a high error because the two SISO controllers sometimes trigger antagonistic actions. In particular, one controller increases cache size to improve IPS, inadvertently increasing *P*, while the other reduces frequency to meet the *P* goal, degrading IPS. The result is a suboptimal working point.

Heuristic is also limited in its capability. Even though it uses metrics such as the memory boundedness of the execution to choose its actions, its thresholds and rules are based on static profiling with the training set. It lacks a learning framework like *MIMO*. Hence, it may not make the choices that align best with the dynamic execution of the production set applications.

For the non-responsive applications, all the architectures perform similarly.

3.5.5 Using MIMO for Time-Varying Tracking

We change the IPS and P target values periodically, to minimize the decrease in quality of experience in handheld devices [127], and observe the outputs using the *MIMO*, *Heuristic*, and *Decoupled* architectures. Figure 3.8 shows the resulting IPS values as a function of time for each architecture and the reference. Figures 3.8(a) and (b) correspond to *astar* and *milc*, respectively. We do not show P values as all the architectures perform similarly well.



(a) astar (b) milc

Figure 3.8: Examples of time-varying tracking.

The figure shows that *MIMO* is able to track the time-varying reference IPS values well, much closer than the other architectures. *Heuristic* and *Decoupled* attain an IPS that is lower than the target. For the lowest IPS at the end of the battery life, *MIMO* performs a bit worse than expected (but still better than the other architectures). This is because we set an IPS target that is too aggressive when combined with the companion P target.

3.5.6 Using MIMO for Fast Optimization Leveraging Tracking

We compare the ability of the different controllers to optimize a combination of outputs. We first consider minimizing $E \times D$. Figure 3.9 shows the $E \times D$ of the different applications under *MIMO*, *Heuristic*, and *Decoupled*. For each application (and the average in the far right), the bars are normalized to the $E{\times}D$ of *Baseline*.

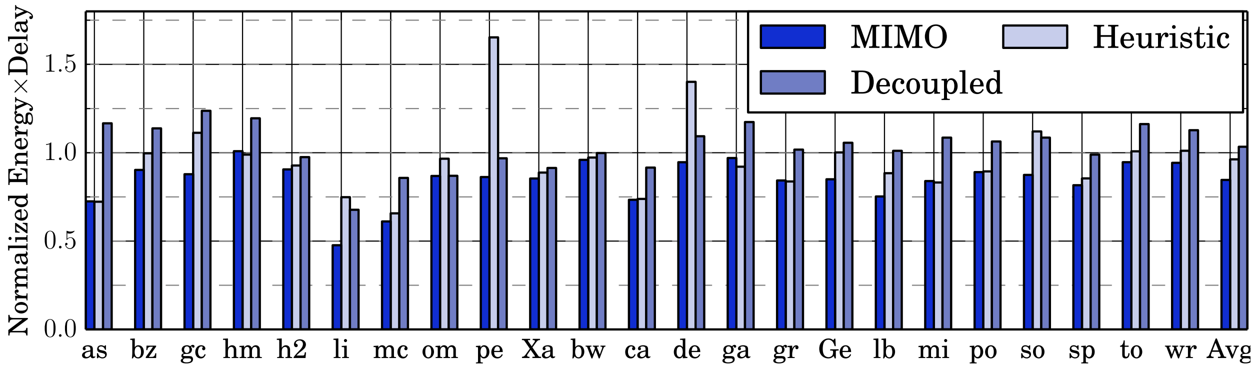


Figure 3.9: Energy×Delay minimization.

On average, *MIMO*, *Heuristic*, and *Decoupled* reduce the $E{\times}D$ of the applications by 16%, 4%, and -3%, respectively. *MIMO* is effective in practically all the applications, even though there is substantial variation across integer and floating-point applications. *Heuristic* does well on some codes, but not on others, such as *perlbench* and *dealII*. This is because some of the heuristics and thresholds from the training set do not work well all the time. In *perlbench*, the application is classified in a way that results in limiting the set of cache sizes explored in the search, resulting in sub-optimal $E{\times}D$. In *dealII*, the code has a relatively low number of memory accesses per operation, but is fairly sensitive to L2 misses. The heuristic assumes that *dealII* is compute intensive and has little sensitivity to cache size, which is incorrect. Finally, *Decoupled* chooses bad values for cache size and frequency because of lack of coordination between the sub-controllers.

We obtain similar results for energy (E) and $E{\times}D^2$. However, we do not show results for brevity. *MIMO*, *Heuristic*, and *Decoupled* reduce the $E{\times}D^2$ by 18%, 7%, and 4%, respectively, and the E by 9%, 1%, and 0%, respectively, over *Baseline*. Most importantly, for these experiments, the *MIMO* and *Decoupled* controllers *remain unmodified*. Even the optimizer search in the IPS^n–P space is parameterized by n and *remains unchanged*. However, the *Heuristic* controller needs to be completely *redesigned and retuned* to optimize $E{\times}D^2$ or E.

3.5.7 Adding a New Input: Configurable ROB Size

We augment the processor with the resizable ROB of Section 3.3.4, and repeat the experiments in the previous section. We cannot use *Decoupled* because the system has 3 inputs and only 2 outputs. Note that, while the controller for *MIMO* is regenerated semiautomatically as explained in Section 3.3.4, the controller for *Heuristic* needs to be redesigned largely *from scratch*.

Figure 3.10 shows the $E{\times}D$ of the different applications under *MIMO* and *Heuristic*. As usual, the bars in each application (and the average) are normalized to the $E{\times}D$ for *Baseline*. On average, *MIMO* and *Heuristic* reduce the $E{\times}D$ of the applications by 25% and 12%, respectively.

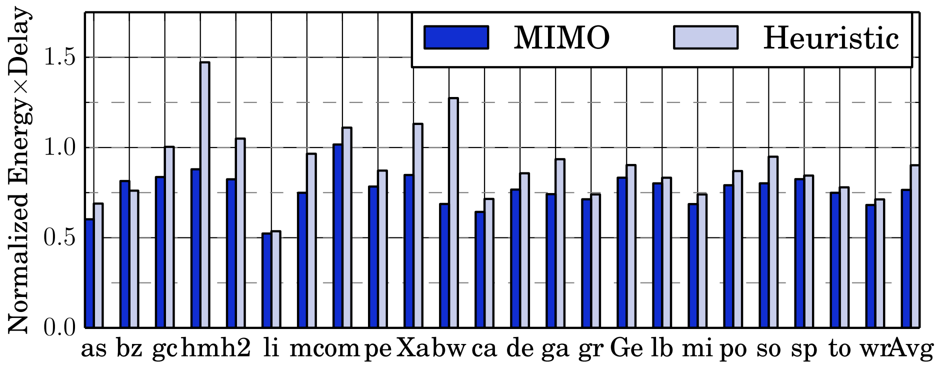


Figure 3.10: Energy×Delay minimization with 3 inputs.

We note that *MIMO* attains a substantial $E{\times}D$ reduction. *Heuristic* does not do as well, and is affected by several outliers. The rules and threshold values have become more complicated with more inputs, and some of the tuning performed based on the training set does not work well all the time. In some cases, finding the best values of each of the inputs in sequence, one by one, produces a configuration that is inferior to the one attained by considering all three inputs simultaneously.

3.6 CHAPTER SUMMARY

Control-theoretic MIMO controllers, which actuate multiple inputs and control multiple outputs in a coordinated manner are likely to be key as future processors become more resource-constrained and adaptive. In this chapter, we applied MIMO control to the development of controllers for dynamic architecture parameter tuning. To our knowledge, this is the first work in this area. We discussed three ways in which a software agent, such as the OS or the runtime, can use a MIMO controller. We developed an example MIMO controller and showed that it is substantially more effective than controllers based on heuristics or built by combining single-output formal controllers.

CHAPTER 4: YUKTA: MULTILAYER RESOURCE CONTROLLERS TO MAXIMIZE EFFICIENCY

After introducing MIMO control for processors, we consider the need to coordinate the multiple layers in which computers are organized. Examples are the hardware, OS, and networking layers. Each layer is built independently and performs different functions. Managing such a system scalably and portably requires a controller in each layer, and that the different controllers coordinate their operation. To achieve this capability, we conceive a new design called *Yukta*.

There have been many works on managing resources from different layers of a computer system (e.g., [61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80]). As indicated in Section 1.2, designs vary depending on whether they manage resources in a monolithic, decoupled, or coordinated manner. In addition, they also vary depending on whether they use heuristics, control-theoretic methods, machine learning, or optimization theory.

There is no prior work on formal control methods to develop coordinated multilayer controllers. Popular formal control designs such as PID controllers [91] and similar SISO proposals [1, 13, 102, 104] can only monitor one goal and change one parameter. Some designs [66,67,79,80] use a collection of separate SISO controllers, but cannot manage the interaction between the goals [111,133]. There are controller designs which are MISO [99,100, 101,103] or MIMO (e.g., MPC controller [111] and the LQG controller from Chapter 3) but all of these controllers are intended for standalone use, and do not have channels for coordination between multiple controllers. Some designs employ heuristics to make up for this deficit [66], but this defeats the purpose of formal control methods. Importantly, existing designs are not natively robust to the large uncertainty that appears in the presence of multiple controllers, each acting with partial system information. To our knowledge, Yukta is the first approach that uses control-theoretic methods to build coordinated, modular multilayer controllers.

4.1 CHALLENGE: CONTROLLING MULTIPLE LAYERS

The operation of a computer involves interactions between multiple layers, including the hardware, OS, and networking layers. In such an environment, designing a single, unified formal controller that senses and actuates on signals from all the layers is both impractical and non-portable. This is because each layer has its own specialized design team, which is intimately familiar with the control signals in that layer, but not with those of other layers. Moreover, any controller designed by this team should be useful even if the other layers' implementation changes — e.g., the same hardware controller should work for different OSs.

Hence, control should be organized in multiple layers, with a per-layer controller. However, a decoupled design with independent controllers is also undesirable.

4.2 SOLVING THE PROBLEM WITH SSV CONTROLLERS

To address the challenge of controlling multiple layers, we propose using Collaborative MIMO SSV controllers. In this solution, there is preferably an SSV controller in each layer. Less desirably, there is an SSV controller at least in the layer that controls outputs requiring accurate control (e.g., temperature or power), and other types of controllers in the other layers. We call this general approach *Yukta*. In the following discussion, we assume an SSV controller in each layer; in the evaluation section, this is relaxed.

SSV control is suitable for multilayer computer control. To see why, compare the traits of SSV control as per Section 2.3 to the needs of computer systems. First, SSV controllers take external signals. Traditionally, these signals were used by the controller to monitor "external disturbances". In computer systems, we use them to pass information from the controller of one layer to that of another layer at runtime. The second controller can use the signals to make better decisions, although it cannot control such signals. For example, an OS controller can pass the number of running threads as an external signal to a hardware DVFS controller.

Second, in SSV controllers, designers can specify bounds for the output value deviations. This ability allows the design of more accurate computer controllers. In addition, if any output is passed as an external signal to another layer's controller, the availability of precise output bounds helps the pair of controllers improve their coordination.

An important case is when two controllers have the same output — e.g., both the hardware and the OS controllers limit the temperature. In non-SSV controllers, this output is liable to large value oscillations, as both controllers attempt to push its value up, overreach the limit, then push its value down, and overreach again. Instead, two SSV controllers can coordinate. If each controller knows the bounds that the other controller has set for the output value, it will take a more measured action based on the expected response of the other controller.

Third, consider the ability to design SSV controllers with uncertainty guardbands. In a multilayer controller, one controller's actions may indirectly affect the outputs that a second controller is supposed to control. This interference can be incorporated in the SSV controller design by increasing the uncertainty guardband of the second controller.

Finally, with SSV controllers, designers can specify realistic inputs, rather than assuming that inputs take continuous, unlimited values. In computer systems, inputs typically take a discrete set of values within a range. For example, core frequency can only take a few discrete values. In our SSV design, we provide, for each input, a notion of allowed discrete values. This information enables more accurate controller design. In addition, if an input is passed as an external signal to another layer's controller, it allows better coordination between controllers.

Figure 4.1 shows the envisioned Yukta control system for a two-layer system. Each controller takes external signals from the other controller.

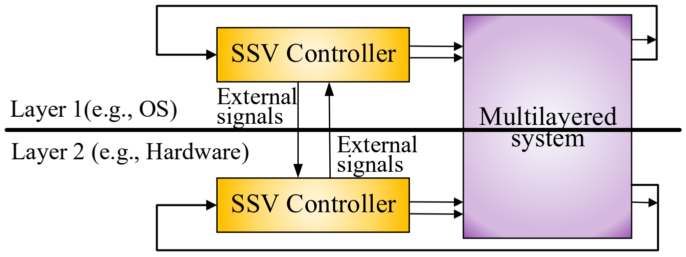


Figure 4.1: Yukta multilayer SSV controller.

4.3 DESIGNING SSV CONTROLLERS

Figure 4.2 shows the process of designing a Yukta multilayer SSV controller. In each layer, a team initiates the design of the layer's controller by selecting the input signals and their discretized values, the output signals and their deviation bounds, and the external signals that the controller takes.

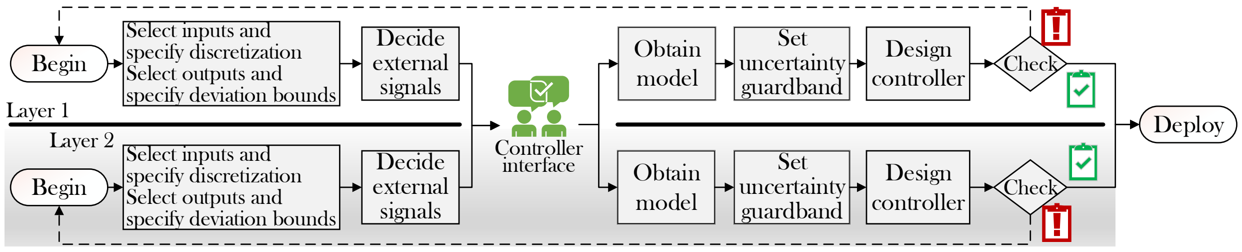

Begin
Select inputs and specify discretization Select outputs and specify deviation bounds
Decide external signals
Layer 1
Layer 2
Begin
Select inputs and specify discretization Select outputs and specify deviation bounds
Decide external signals
Controller interface
Obtain model
Set uncertainty guardband
Design controller
Check
Obtain model
Set uncertainty guardband
Design controller
Check
Deploy

Figure 4.2: Process to design a Yukta multilayer SSV controller.

Then, the teams exchange *Interface* information. This is meta-information about external signals and common outputs. Specifically, for outputs common to both controllers, the teams exchange their layer's deviation bounds; for an external signal to a controller from a second layer, the second layer team passes the allowed discrete values if the signal is an input in the second layer, or the deviation bounds if it is an output in the second layer.

After this communication step, each team develops a model of the system according to their layer's perspective (possibly with the system identification methodology [109]), sets its controller's uncertainty guardband, designs the SSV controller using MATLAB controller synthesis routines [134], and validates it. Finally, the designs of all the layers are combined, validated as a group, and deployed.

This process can work across companies. For example, Intel Skylake [1] includes new hardware control algorithms for which some parameters must be set by the user or the OS. Soon after the processor release, Microsoft announced power management features in the Windows OS to take advantage of these features [5].

If the timelines of the two teams do not overlap, or close communication between teams is not desired, an approach like Figure 4.2 can still work, albeit less effectively. Teams can use historically-available or standard information from the other layer for their external signals. An example is how OS teams use the popular P-state interface of processors [135, 136]. Alternatively, a team can do without any extra information for their external signals. In this case, the team should increase their uncertainty guardband. This works because SSV controllers withstand inaccurate assumptions.

A multilayer SSV controller can be used in two ways. The basic use is when we want each output to meet a certain target value. In this case, the controllers will attain output values within the allowed bounds around the target values.

A second use is when we want some outputs (or combination thereof) to maximize or minimize their value, subject to other outputs to be within certain limits. An example is to minimize the energy delay product ($E \times D$ or EDP) subject to a power constraint. In this case, the controller needs to perform some search to find the best configuration. Hence, each SSV controller is augmented with an optimizer module (Figure 4.3). We discuss the operation of the optimizer in Section 4.5.4.

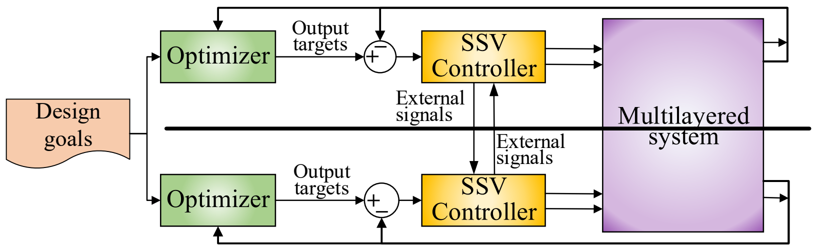


Figure 4.3: Yukta controller augmented with optimizers.

4.4 SCALABILITY TO SEVERAL LAYERS

In an environment with several layers, we envision the controller of a given layer to communicate mostly or only with the controllers of its two neighboring layers. This is consistent with the design of abstractions in a layered structure. As layer i passes signals to layer *i+1*, such signals already implicitly include the contribution of layers *i-1*, *i-2*, etc. The latter layers should not need to communicate directly with layer *i+1*.

4.5 PROTOTYPING YUKTA

We prototype a multilayer SSV controller in a challenging environment: an ODROID XU3 board [137], which has an 8-core asymmetric processor running Linux. The processor is Samsung Exynos 5422, built using ARM big.LITTLE technology [138]. It has a cluster of four little cores (the in-order, low power Cortex A7), and a cluster of four big cores (the out-of-order, high performance Cortex A15). The multicore runs Ubuntu 15.04, which contains the HMP (Heterogeneous Multi-Processing) task scheduler [139, 140]. This scheduler was designed for ARM big.LITTLE platforms. The scheduler can turn cores on/off dynamically (called CPU hotplugging) based on requests from a thermal management module. Figure 4.4 shows a picture of our experimental platform.

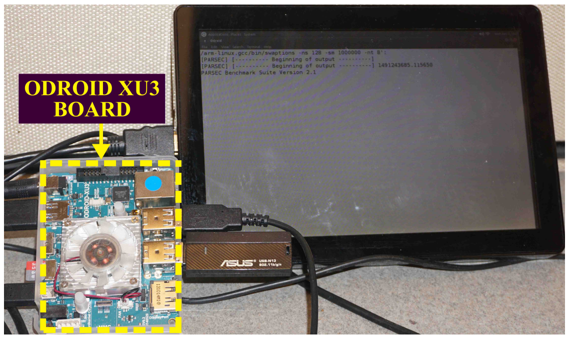


Figure 4.4: The Odroid XU3 used for our prototype.

We prototype a two-layer SSV controller. One controller controls hardware parameters (hardware controller), and another controls thread scheduling parameters (software/OS controller). Our goal for the hardware controller is to minimize EDP while keeping power and temperature below certain limits. Our goal for the software controller is to simply minimize EDP. It relies on the hardware controller to keep power and temperature within limits.

Our choice of controllable parameters is limited by what is feasible on the board. We cannot actuate on internal structures of the processor, such as its pipeline configuration. Similarly, the HMP scheduler for big.LITTLE systems has dependencies on parts of the OS [141, 142], so we need to carefully choose what we modify. Since our goals involve minimizing EDP, we also design optimizer modules for each of the controllers. The resulting system is shown in Figure 4.5. In the following sections, we consider each controller in turn.

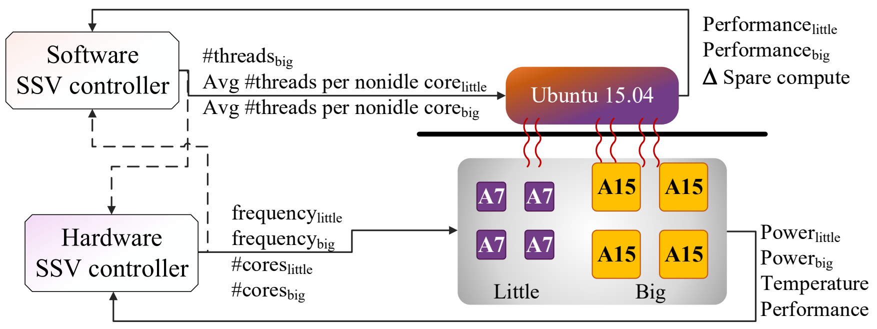


Figure 4.5: Prototyped controllers on the Odroid XU3.

4.5.1 Designing a Hardware Controller

Table 4.1 shows the inputs, outputs, and external signals for the hardware controller. The controller actuates on four system inputs: number of big cores, number of little cores, frequency of the big cluster, and frequency of the little cluster. The number of active cores in either cluster can vary from 1 to 4. The big cluster frequency can vary from 0.2 to 2.0 GHz, and the little cluster frequency from 0.2 to 1.4 GHz, both in steps of 0.1 GHz. As per Section 2.3, SSV designs take saturation and quantization information for each input signal. Hence, we give the possible values that each input can take.

Table 4.1: Parameters of the hardware controller in our prototype multilayer SSV system in an ODROID XU3 board.

Goal	**Inputs**		**Outputs**		**External Signals**	Δ_{unc}
	Signals	**Wts**	**Signals**	**Bnds**		
Minimize EDP s.t. Power$_{big}$ < Power$_{big}^{max}$, Power$_{little}$ < Power$_{little}^{max}$, and Temp < Temp^{max}	#big cores	1	Performance	±20%	#threads$_{big}$, avg #threads per non-idle core in cluster$_{big}$, and avg #threads per non-idle core in cluster$_{little}$	±40%
	#little cores	1	Power$_{big}$	±10%		
	frequency$_{big}$	1	Power$_{little}$	±10%		
	frequency$_{little}$	1	Temp	±10%		

We set the weights (listed as Wts in the table) of each input. The relative weights of the inputs determine the eagerness of the controller to change each input. Specifically, the controller will change low-weight inputs more eagerly than high-weight ones. Since the overhead of changing a cluster's frequency is comparable to the overhead of turning a core on/off with hotplugging, we set the weights of all the inputs to be the same.

Additionally, the absolute values of these weights determine the aggressiveness of the controller response. High absolute weights produce a sluggish controller, which changes the inputs only slowly when the outputs are perturbed from their target value. Low input weights produce an eager controller, which changes the inputs quickly. Neither extreme is desirable in processor control. We perform a sensitivity analysis of weight values in Section 4.7.5. Based on that, we set all the weights to 1 (Table 4.1).

The hardware controller monitors four system outputs: the performance of the workload measured in total billions of instructions committed per second (BIPS), the big cluster power, the little cluster power, and the hot-spot temperature. To set the bounds of the output deviations (listed as Bnds in Table 4.1), we proceed as follows. When we characterize the processor with a training set of applications to build the system model (Section 4.5.3), we record the range of values exhibited by each output. We then set the bounds to be a percentage of such range.

Of the four outputs considered, the power of both clusters and the temperature are critical for the integrity of the board. Hence, we assign them a bounds range that is $\pm 10\%$ of their maximum range; for the performance, since it is less critical, we assign a $\pm 20\%$ bounds range. The synthesis routines inform the designer when tighter bounds than those specified can be achieved. Alternatively, if any of these bounds is too tight, the MATLAB SSV controller synthesis routines will fail to build the controller. At runtime, the controller keeps the deviations of all outputs within these bounds for feasible targets. If it cannot, it keeps the deviations at least proportional to their relative bounds values as given by the designer. We perform a sensitivity analysis of bounds ranges in Section 4.7.5.

We provide three external signals to the hardware controller (Table 4.1). They are the signals that the software controller actuates on (i.e., its inputs). We will discuss them later.

Finally, as we generate the SSV controller, we need to provide an uncertainty guardband (Δ_{unc} in Table 4.1). Uncertainty is the result of limitations in how the model describes the real system, and of unpredictability in various system components. An example of the latter is aspects of the HMP scheduler, which sometimes packs multiple threads on a core while leaving another core idle. In Section 4.7.5, we evaluate several uncertainty guardbands for the hardware controller. Based on that, we pick a guardband of $\pm 40\%$. If the guardband is too large, MATLAB routines cannot build a controller that can deliver the output deviation bounds. If the guardband is too small, the controller will report so at runtime.

In contrast to these few intuitive parameters, industry heuristic controllers have an order of magnitude more parameters. For example, in the Samsung Exynos 5422 hardware we use, to change the big cluster frequency based on the current temperature, there are many thresholds (each with its own rule) [143, 144, 145]. These rules are used to assess the impact of the temperature, detect whether temperature is rising or falling, and then change the big cluster frequency. Furthermore, to control all the four hardware outputs (i.e., performance, power of big and little clusters, and temperature), the Exynos 5422 uses tens of interdependent settings that require tuning. Our approach eliminates the need for this extensive tuning.

4.5.2 Designing a Software Controller

Table 4.2 shows the inputs, outputs, and external signals for the software controller. The controller assigns the application's threads to cores. Ignoring any differences between threads, the first decision is how to partition the threads between the big and little clusters. The second decision is how to assign the threads in a cluster to cores, possibly leaving some cores idle. For example, in a cluster with 4 threads and 4 cores, it may be better to assign two threads per core, enabling the hardware controller to power-off two cores. Therefore, the software controller actuates on three inputs: the number of threads assigned to the big cluster (leaving the rest for the little cluster), the average number of threads running on each non-idle big core, and the average number of threads running on each non-idle little core (Table 4.2).

Table 4.2: Parameters of the software controller in our prototype multilayer SSV system in an ODROID XU3 board.

Goal	**Inputs**		**Outputs**		**External Signals**	Δ_{unc}
	Signals	**Wts**	**Signals**	**Bnds**		
Minimize EDP	#threads$_{big}$	2	Performance$_{little}$	±20%	#big cores, #little cores, frequency$_{big}$, and frequency$_{little}$	±50%
	Avg #threads per non-idle core in cluster$_{big}$	2	Performance$_{big}$	±20%		
	Avg #threads per non-idle core in cluster$_{little}$	2	Δ SpareCompute$_{big-little}$	±20%		

To set the input weights, we first note that changing any of the three inputs involves migrating a thread. Since the change overhead is roughly the same for all three inputs, we assign the same weight to all inputs. However, we want the software controller to react more conservatively to output changes than the hardware controller. This is because applications change the number of threads dynamically in an unpredictable manner for the controller — e.g., some threads block on I/O. We do not want the controller to react immediately and cause oscillations. Consequently, we set the weight of all inputs to 2 (Table 4.2), which happens to be twice the weight of the hardware controller's inputs.

The controller monitors three outputs: performance of the big-cluster threads (in total committed BIPS), performance of the little-cluster threads, and difference in Spare Compute Capacity (SC) between the big and little clusters. At a high level, the higher the difference in SC is, the more threads the controller will move from the little to the big cluster.

The SC of a cluster is estimated as follows. SC should be raised when there are many cores in the cluster that are both on and idle. On the other hand, SC should be lowered when the cluster has many threads multiplexed on the busy cores; these threads could be spread over all the cores that are on. So, we define a cluster's SC as:

$$SC = \#idle\_cores\_on - (\#threads - \#cores\_on) \tag{4.1}$$

Since we consider all outputs to have similar importance, we set their deviation bounds to ±20% of their maximum range — like the non-critical outputs in the hardware controller.

To coordinate with the hardware controller, the software controller takes as external signals all the signals that the hardware controller actuates on. Finally, the uncertainty guardband used for the software controller should be higher than that of the hardware controller. This is because the main action of the software controller (i.e., assign threads to cores) is directly affected by an unpredictable event: dynamic changes in the number of application threads. After evaluating several uncertainty guardbands, we set the guardband value to ±50%.

4.5.3 Modeling the Controlled System

The process of designing a controller requires that we build a model of the controlled system — i.e., the Odroid board. To build the models for both controllers, we use the System Identification methodology [109] (Section 2.4). From the system identification data, we use the Box-Jenkins polynomial model structure [146] to obtain a dynamic model of the system. The model generated for both controllers has dimension four — i.e., it predicts the value of an output at time T as a function of the values of all the outputs at times T-1, ... T-4, and the values of all the inputs at times T, ... T-3.

4.5.4 Designing Optimizers

The goal of each of the optimizers is to provide increasingly better targets for the output signals, so that the corresponding controller can tune the input signals (Figure 4.3). To see how they operate, consider the hardware controller. The optimizer reads the outputs of the system (Perf, Power$_{big}$, Power$_{little}$, and Temp), computes the resulting EDP, and changes the output targets passed to the controller (Perf$_{\circ}$, Power$_{big\_\circ}$, Power$_{little\_\circ}$, and Temp$_{\circ}$) to attain a lower EDP. This will trigger the controller to actuate on the input signals (#big cores, #little cores, freq$_{big}$, and freq$_{little}$) so the system converges to the new output targets. The optimizer will then read the new outputs and repeat the process, progressively generating better targets that produce lower EDP values.

Recall that EDP is proportional to Power/Perf^2. Hence, to lower EDP, the optimizer keeps increasing Perf$_{\circ}$ a lot while increasing Power$_{big\_\circ}$ and Power$_{little\_\circ}$ a little. When the result is that EDP has increased, the optimizer discards the latest move, and moves in the opposite direction: it decreases Perf$_{\circ}$ a little while decreasing Power$_{big\_\circ}$ and Power$_{little\_\circ}$ a lot. Eventually, the optimizer settles into a desirable set of output targets. The algorithm of the optimizer is given in Section A.1 of the appendix.

4.6 EXPERIMENTAL METHODOLOGY

4.6.1 Infrastructure

The ODROID XU3 has on-board power sensors that measure the power drawn by the big and little clusters. These sensors update every 260 ms. There are on-chip sensors that measure temperature. We set up performance counters on all cores using the Linux `perf` API [147] to measure the number of instructions committed per second. The number of cores in each cluster and the cluster frequency can be changed by writing to appropriate `cpufreq` files. Thread scheduling is performed through `sched_setaffinity` system calls.

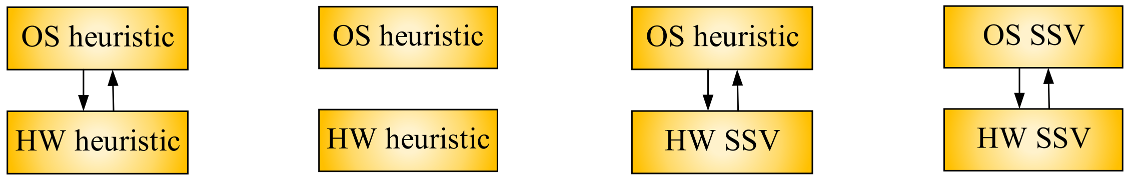
The controllers are invoked every 500 ms. This time is determined by the update period of the power sensors. Many prior works that use real systems use comparable sampling intervals (e.g., 0.5 s – 2 s in [65,66,78,148]). Both controllers are implemented as independent privileged processes, as we cannot add hardware modules to the board.

The power and temperature limits that we use in our evaluation are constrained by the emergency power and thermal heuristics of the board. These heuristics are automatically triggered when power or temperature increase beyond preset thresholds for extended periods of time [143, 144, 145]. We identify the minimum thermal threshold that triggers these heuristics and use it as the limit for temperature. Similarly, we set the limits for the power consumed by the little and big clusters to be below the emergency-triggering values. The limits we use are 0.33 W, 3.3 W, and 79 °C for the power of the little cluster, power of the big cluster, and temperature.

We evaluate Yukta with 8-threaded PARSEC programs with native datasets (*blackscholes*, *bodytrack*, *facesim*, *fluidanimate*, *raytrace*, *x264*, *canneal*, *streamcluster*), 8 copies of SPEC06 programs with train datasets (*h264ref*, *mcf*, *omnetpp*, *gamess*, *gromacs*, *dealII*), and program mixes. For training, we use a different set of programs: *swaptions* and *vips* from PARSEC, *astar* and *perlbench* from SPECINT06, and *milc* and *namd* from SPECFP06.

4.6.2 Schemes for Comparison

In our ODROID XU3 board, we implement the four two-level controllers shown in Figure 4.6. Table 4.3 lists their names and describes them in detail.



(a) Coordinated heuristic (b) Decoupled heuristic (c) Yukta:HW SSV+ OS heuristic (d) Yukta:HW SSV+ OS SSV

Figure 4.6: Two-level controllers evaluated.

Table 4.3: Description of the controllers.

Scheme	**Description of the OS and HW controllers**
(a) *Coordinated heuristic*	OS: Scheduler with power and performance heuristics. Uses number, type, and frequency of cores.
	HW: Increases frequency and #cores while operation is safe. Uses thread distribution to make decisions.
(b) *Decoupled heuristic*	OS: Roubd-robin assignment of threads to cores.
	HW: Sets frequency, #cores to maximum value. On a violation, it reduces frequency first, then #cores.
(c) *Yukta: HW SSV+ OS heuristic*	OS: Like the OS controller in *Coordinated heuristic*.
	HW: SSV design from Section 4.5.1.
(d) *Yukta: HW SSV+ OS SSV*	OS: SSV design from Section 4.5.2.
	HW: SSV design from Section 4.5.1.

In the *Coordinated heuristic* scheme, the OS controller is similar to the HMP task scheduler from ARM, Linaro and Samsung [139, 140], except that it is modified to optimize E×D. The OS controller coordinates with the hardware controller in that it uses the number, type, and frequency of the available cores to schedule threads. The hardware controller sets the number of cores and their frequency to maximum values until the power or temperature exceeds the limits; when this happens, it finds a lower, safe frequency value for that cluster. It coordinates with the OS controller in that it uses how the threads are distributed across all the cores to determine the safe frequency. This OS-hardware scheme is representative of industry-standard controllers in big.LITTLE systems, and we use it as a baseline.

The *Decoupled heuristic* scheme takes uncoordinated decisions at each layer. The OS controller distributes threads on cores in a round-robin manner. The hardware controller is similar to the `performance` power governor in Linux [149]. It sets the number of cores and their frequency to maximum values whenever temperature and power are within limits. When the limits are exceeded, it uses threshold-based rules to temporarily reduce frequency first, and then the number of cores — irrespective of the number of threads.

We design two schemes based on our proposed coordinated Yukta methodology. The first one, *Yukta: HW SSV+OS heuristic*, uses an SSV hardware controller as in Section 4.5.1 and a heuristic-based OS controller like the one in Coordinated heuristic. The second one, *Yukta: HW SSV+OS SSV*, uses an SSV hardware controller as in Section 4.5.1 and an SSV OS controller as in Section 4.5.2.

4.7 EVALUATION

4.7.1 Multilayer Controller Evaluation

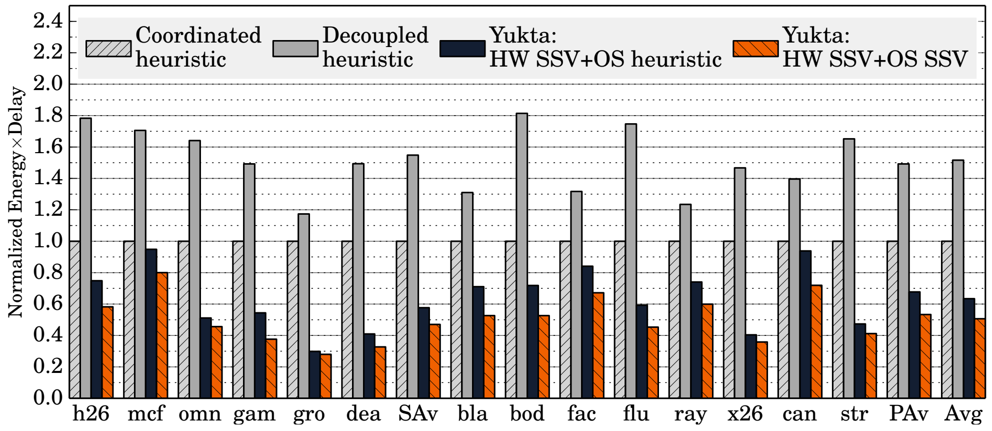
Figure 4.7 compares our four two-layer controller schemes running our applications. Figure 4.7(a) shows the EDP of the applications, and Figure 4.7(b) the execution time. In each chart, the bars from left to right correspond to individual SPEC applications, the average of the SPEC applications (*SAv*), individual PARSEC applications, the average of the PARSEC applications (*PAv*), and the average of all the applications (*Avg*). Each application has a bar for each of the four controller schemes. The bars are normalized to *Coordinated heuristic*.

Figure 4.7(a) shows that *Decoupled heuristic* has higher EDP than *Coordinated heuristic*. On average, decoupling the controllers results in a 52% higher EDP. On the other hand, using Yukta causes EDP to decrease. On average, *Yukta: HW SSV+OS heuristic* has a 37% lower EDP than *Coordinated heuristic*. Furthermore, having both SSV controllers as in *Yukta: HW SSV+OS SSV* results in an average EDP that is 50% lower than *Coordinated heuristic*. Thus, SSV controllers offer substantial improvements over existing systems.

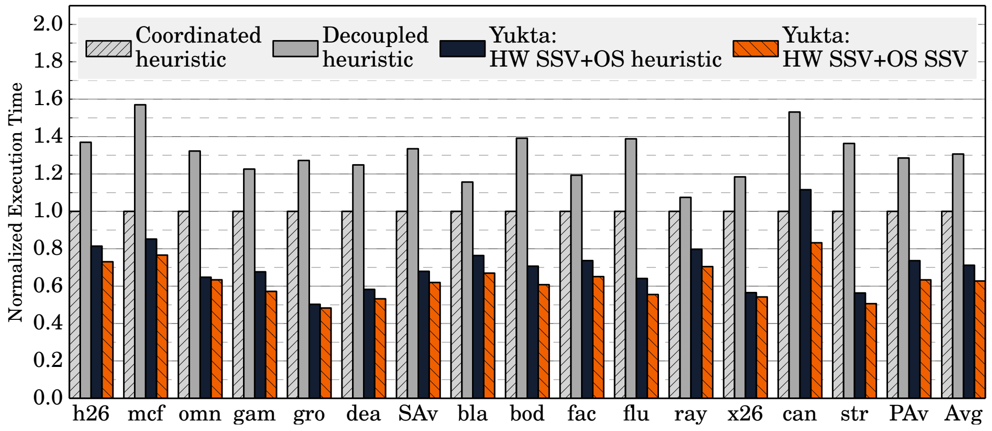
The execution times in Figure 4.7(b) show similar results. *Decoupled heuristic* increases the execution time by 30% on average. On the other hand, *Yukta: HW SSV+OS SSV* reduces the time by 29% on average, and *Yukta: HW SSV+OS SSV* by even more, namely a substantial 38% on average.

To gain insight into the impact of Yukta, we focus on the execution of the *blackscholes* application (labeled *bla* in Figure 4.7). This application begins with a single thread and later executes 8 parallel threads. The work in the parallel phase does not have large variations. Figure 4.8 shows the power consumed by the big cluster in *blackscholes* as a function of time, for the four controller schemes. Recall that the limit in sustained power is 3.3 W.

The figure shows that, under all schemes, the power fluctuates. At certain points, it goes over the limit but, immediately after, the system reacts and brings the power down again. What varies between the four schemes is the number and amplitude of these peaks and valleys, and the average value of the power in the steady-state periods. In general, a better



(a) Normalized Energy×Delay (EDP)



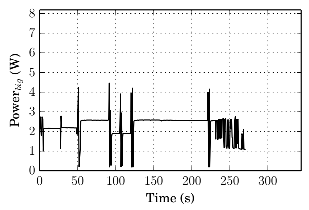
(b) Normalized execution time

Figure 4.7: Energy×Delay (a), and execution time (b) for the four two-layer controller schemes considered.

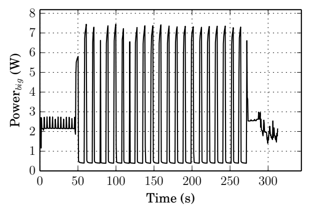
controller will minimize the number and amplitude of these peaks and valleys, and keep the power in the steady-state periods as close as possible to 3.3 W.

In *Decoupled heuristic* (Figure 4.8(b)), there are many oscillations. In this scheme, the hardware controller increases the number of cores and their frequency to the maximum, while the OS controller simply assigns threads round-robin. This causes the power to go over the limit and trigger the emergency system, which reduces the frequency of the cores and shuts off some cores. The power then drops to low values, and the hardware controller again increases the number of cores and their frequency to the maximum. The result is continuous power oscillation.

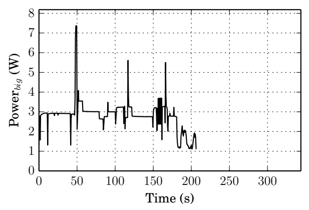
The *Coordinated heuristic* scheme (Figure 4.8(a)) drastically reduces the amplitude and number of these peaks and valleys. This is thanks to the coordination between the two controllers: the hardware controller knows the distribution of the active threads, and the OS controller knows the number, type, and frequency of the active cores.



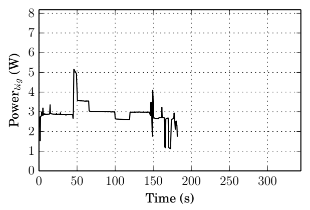
(a) *Coordinated heuristic*



(b) *Decoupled heuristic*

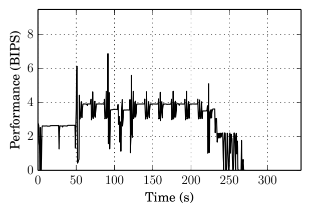


(c) *Yukta: HW SSV+OS heuristic*

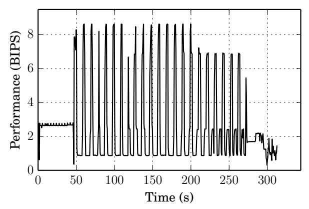


(d) *Yukta: HW SSV+OS SSV*

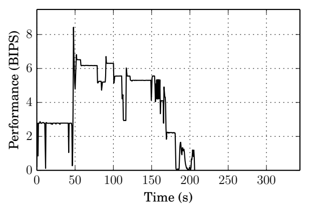
Figure 4.8: Power consumed by the big cluster in *blackscholes* as a function of time for the four controller schemes.



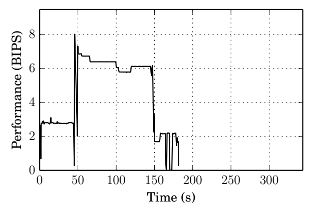
(a) *Coordinated heuristic*



(b) *Decoupled heuristic*



(c) *Yukta: HW SSV+OS heuristic*



(d) *Yukta: HW SSV+OS SSV*

Figure 4.9: Performance of *blackscholes* in BIPS, as a function of time for the four controller schemes.

As we move to *Yukta: HW SSV+OS heuristic* (Figure 4.8(c)) and, especially, *Yukta: HW SSV+OS SSV* (Figure 4.8(d)), the number of peaks and valleys decreases. Moreover, the power during the steady-state periods gets closer to 3.3W. The Yukta controllers control power much better.

These differences in power control translate directly into different performance. Figure 4.9 shows the performance of blackscholes in BIPS, as a function of time, for the four schemes. We see that the performance of the *Decoupled heuristic* scheme (Figure 4.9(b)) oscillates, and the application takes nearly 320 seconds to complete. In the *Coordinated heuristic* scheme (Figure 4.9(a)), the steady-state performance increases, and the application completes in 270 seconds. Finally, in *Yukta: HW SSV+OS heuristic* (Figure 4.9(c)) and *Yukta: HW SSV+OS SSV* (Figure 4.9(d)), the steady-state performance keeps increasing, and the application completes sooner, in 205 and 180 seconds, respectively.

4.7.2 Comparing to LQG Control

We compare Yukta to the MIMO LQG controller described in the previous chapter. Like Yukta, the LQG controller can change many inputs to meet many output targets, and accepts weights for inputs and outputs. Unlike Yukta, however, it does not accept external signals, deviation bounds for outputs, saturation/quantization of inputs, or design with uncertainty guardbands.

Since an LQG controller cannot use external signals, we evaluate the two ways in which it can be used for multilayer control: one that has independent LQG controllers in the hardware and OS layers (*Decoupled HW LQG+OS LQG*), and one that has a single LQG controller that manages both layers (*Monolithic LQG*). The latter is the use in Chapter 3. We use input and output weights comparable to the SSV controllers.

Figures 4.10 and 4.11 compare the EDP and execution time, respectively, of *Coordinated heuristic*, *Decoupled HW LQG+OS LQG*, *Monolithic LQG*, and *Yukta: HW SSV+OS SSV*. The bars are normalized to *Coordinated heuristic*. We see that, on average, *Decoupled HW LQG+OS LQG* delivers EDP and performance similar to *Coordinated heuristic*. This is because each controller works independently without coordination, making the system inefficient.

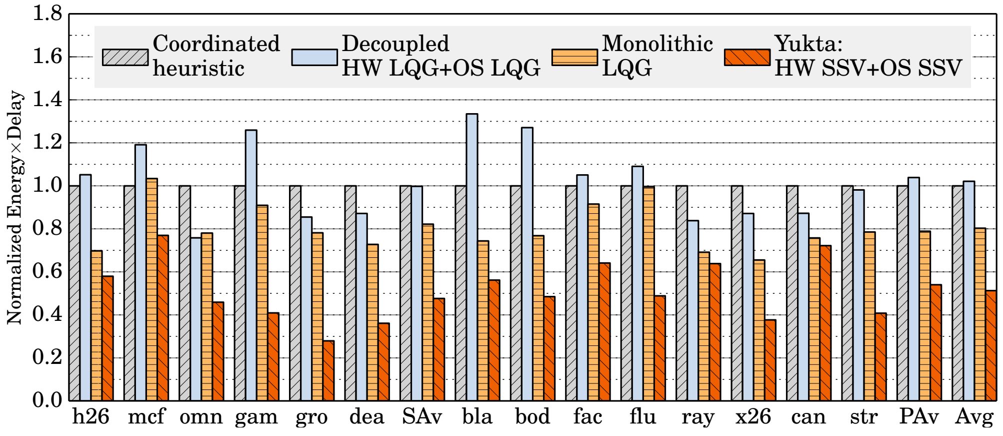


Figure 4.10: Comparing EDP for LQG-based designs.

Monolithic LQG delivers better results, thanks to the centralized decisions taken by the controller. On average, it reduces EDP by 20% and execution time by 11% relative to *Coordinated heuristic*. This is consistent with the 16% EDP reduction reported in [92]. However, these gains are small compared to those of *Yukta: HW SSV+OS SSV*, which attains average reductions of 50% in EDP and 38% in execution time.

The reason for this gap is that LQG controllers have several limitations, as listed above. First, they assume that inputs are continuous and have no bounds. Hence, a controller sometimes attempts to change an input beyond its physical limit, and observes no output change. Only later does the controller try changing another input. This slows down the configuration search. For example, in *bodytrack*, the LQG controller wastes 9% of the time

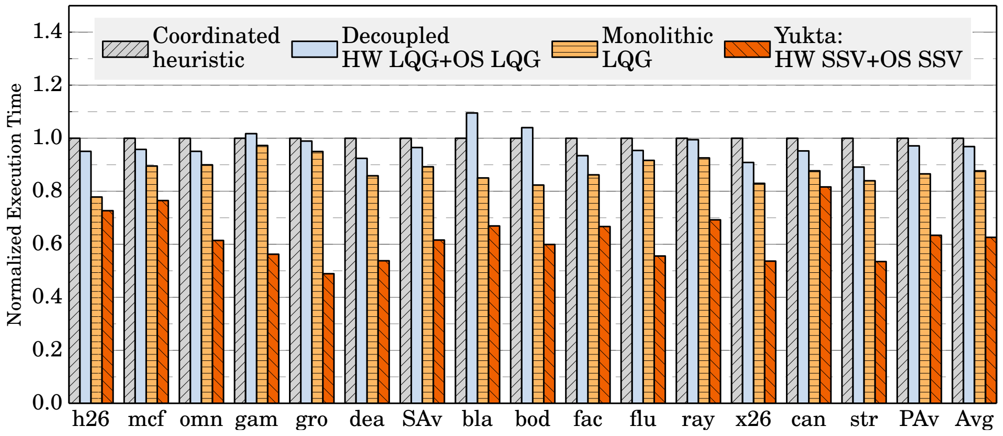


Figure 4.11: Comparing performance for LQG-based designs.

trying to change an input beyond its limit and observing no change.

Second, LQG controllers accept no output bounds; they try to keep output deviations to be proportional to the inverse of the output weights. As a result, the optimizer steers the system to a less optimal configuration, or takes longer to find the best configuration. For example, it can be shown that, in *bodytrack*, the LQG controller takes on average 6 sampling intervals to make the big cluster power converge to a specified target; the SSV controller can achieve this in 2 sampling intervals. Over the entire application, the optimizer takes 90 intervals to find the optimum targets for the LQG controller, while it takes only 30 for the SSV one.

Finally, LQG controllers are not natively optimized for uncertainty. The framework that generates LQG controllers uses uncertainty guardbands to discard unstable designs (Chapter 3). When this happens, it changes the output weights, which slows down the controller. In the framework that generates SSV controllers, instead, uncertainty is an explicit parameter. Hence, the resulting controller is optimal within the uncertainty guardband. Overall, LQG controllers are no match for Yukta designs.

4.7.3 Heterogeneous Workloads

We evaluate four heterogeneous workloads created by combinations of 4-threaded PARSEC codes and 4 copies of SPEC codes: *blmc* (blackscholes+mcf), *stga* (streamcluster+gamess), *blst* (blackscholes+streamcluster), and *mcga* (mcf+gamess).

Figure 4.12 compares the normalized EDP of these workloads under all the heuristic, LQG and Yukta-based designs we built. The results are similar to the homogenous workloads, with the Yukta-based designs exhibiting the lowest EDP, then *Monolithic LQG*, and then

Coordinated heuristic. The reduction in *Yukta: HW SSV+OS SSV* is 47%, which is close to the 50% attained before. This demonstrates the robustness of the Yukta-based designs in diverse environments.

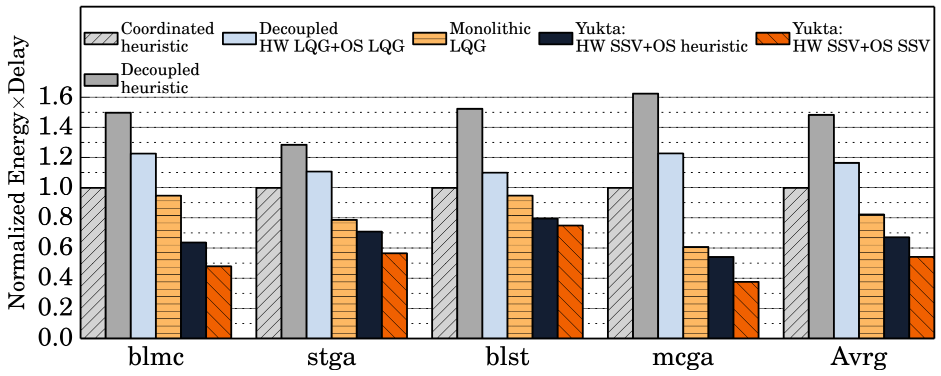


Figure 4.12: Comparing EDP for heterogeneous workloads.

4.7.4 Implementing a Hardware SSV Controller

A hardware implementation of our hardware SSV controller is a simple state machine. From Section 2.3.1, it is characterized by the dimensionality of its state (N), and the number of inputs (I), outputs (O), and external signals (E). It implements the following equations in hardware [91]:

$$x(T+1) = A \times x(T) + B \times \Delta y(T) \tag{4.2}$$

$$u(T) = C \times x(T) + D \times \Delta y(T) \tag{4.3}$$

where x is the state of the controller (N-entry vector), Δy is the external signals and the deviation of outputs from their targets (vector of O+E entries), u is the new inputs (I-entry vector), A is the controller evolution matrix (N×N), B is the matrix of impact of output deviations on the state (N×$(O$+$E)$), C is the state-to-input conversion matrix (I×N), and D is the matrix of feed-through of output deviations to inputs (I×$(O$+$E)$). In our case, I=4, O=4, E=3, and N=20. At every millisecond-level invocation [1], the controller performs these computations, which are nearly 700 32-bit fixed-point operations (additions and multiplications), and needs to store nearly 2.6 KB of data. We have measured that performing these computations on an ARM Cortex A7 core consumes ≈20–25 mW and takes ≈28 μs. We envision that a hardware state machine implementation of this functionality would consume a few milliwatts and have negligible area.

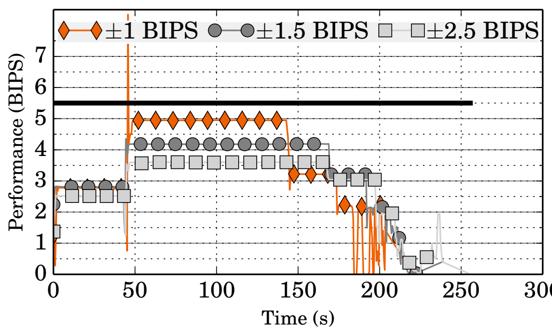
4.7.5 Hardware SSV Controller Analysis

Analysis of Output Deviation Bounds

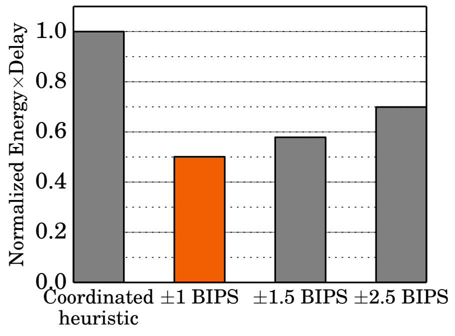
The hardware controller of *Yukta: HW SSV+OS SSV* in Section 4.5 has deviation bounds of ±20% for its performance output (i.e., ±1 BIPS in absolute terms). In this section, we change them to ±30% (i.e., ±1.5 BIPS) and ±50% (i.e., ±2.5 BIPS). Since the OS controller also monitors the performance of each of the clusters, we also increase the bounds for the OS controller proportionally, to ±30% and ±50% for the big and little clusters.

We perform two experiments. In the first one, we set fixed targets for each of the outputs. Specifically, for the hardware controller, we set the performance target to 5.5 BIPS, the power of the big and little clusters to 2.5 W and 0.2 W, respectively, and the temperature of the big cluster to 70°C. For the OS controller, we set the performance targets of the little and big clusters to 1 BIPS and 4.5 BIPS, respectively, and the difference in SC between big and little clusters to 1.

Figure 4.13(a) shows the performance of the computer system as a function of time for the three output deviation bounds (absolute values of bounds are shown for convenience). The data is for blackscholes. Ignoring the initialization and termination stages, we see that the performance remains close to the target, and within the deviation bounds. The tighter the bounds are, the closer the performance is to the target.



(a) Tracking 5.5 BIPS of performance.



(b) Minimizing EDP.

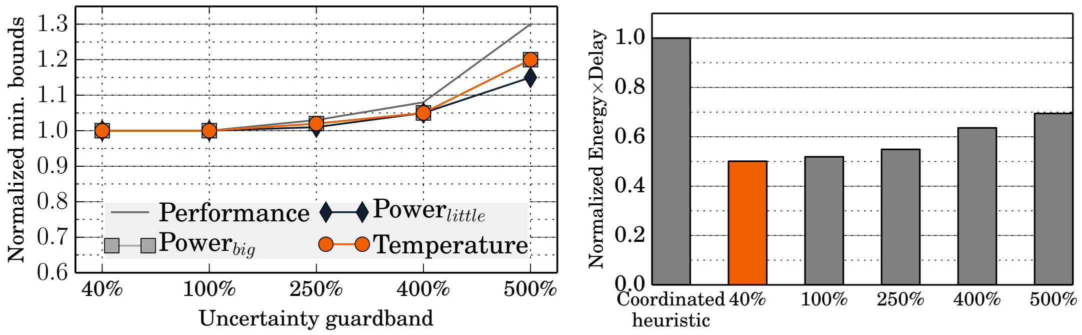
Figure 4.13: Sensitivity to the output deviation bounds.

The second experiment is like the one in Section 4.7.1, where we minimize EDP. Figure 4.13(b) shows the EDP of *Yukta: HW SSV+OS SSV* for the different output deviation bounds (absolute values of bounds shown for convenience), and of *Coordinated heuristic*. The bars are the average of all the applications, and are normalized to *Coordinated heuristic*. We see the EDP with deviation bounds of ±20% (±1 BIPS), ±30% (±1.5 BIPS), and ±50%

(±2.5 BIPS) to be 50%, 41%, and 30% lower than with *Coordinated heuristic*, respectively. As bounds grow wider, the execution is less optimal: output changes that would cause a controller with tight bounds to actuate, do not cause a controller with loose bounds to actuate.

Analysis of Uncertainty Guardband

We examine uncertainty guardbands from ±40% to ±500%. Figure 4.14(a) shows how the output deviation bounds guaranteed by the controller change with different uncertainty guardbands. These bounds are normalized to those in Section 4.5.1, namely ±20% for performance, and ±10% for the rest.



(a) Normalized output bounds. (b) EDP.

Figure 4.14: Sensitivity to the uncertainty guardband.

The figure shows that the guaranteed output deviation bounds increase slowly with the uncertainty guardband. Even for a ±250% guardband, we can synthesize a controller with similar deviation bounds as for a ±40% guardband. This is thanks to using robust control theory.

Figure 4.14(b) shows EDP for different uncertainty guardbands, all normalized to *Coordinated heuristic*. For ±40% guardband, EDP is 50% lower than the baseline. For large guardbands, EDP increases for two reasons. First, the controller is slower to respond to the optimizer-generated targets. Second, the output bounds grow larger, which causes the controller to work less effectively. Overall, we use ±40% as our default guardband.

Analysis of Input Weights

We examine input weights from 0.5 to 2 for all the inputs. This results in controllers that respond at different speeds to output changes. In our experiment, we consider the big-cluster power output and set its target value to 2.5 W. Figure 4.15 shows the big-cluster power as a function of time for the different input weights. The data corresponds to *blackscholes*.

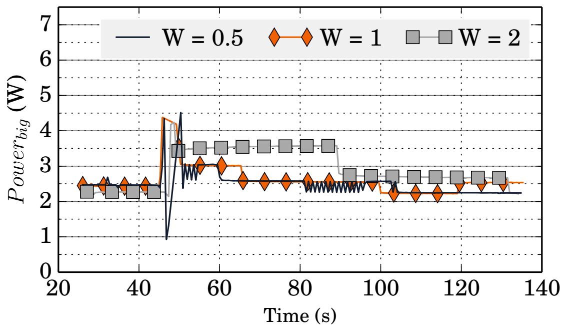


Figure 4.15: Big-cluster power for different input weights.

Ideally, the power should remain at 2.5 W for the whole execution. However, at 45 s, the application launches multiple threads, causing power to rise suddenly. The controller with input weights of 0.5 responds rapidly, creating a series of quick power oscillations. The system is too ripply. The controller with input weights of 2 is slow to change its inputs, keeping the big-cluster power high for about 40 s, before stabilizing at the reference value. Finally, a controller with input weights of 1 responds at modest speed and has no oscillations. Hence, we use input weights of 1.

4.8 CHAPTER SUMMARY

To address the challenge of computers increasingly operating in constrained environments, this chapter presented a new approach to build coordinated multilayer formal controllers for computer systems. The approach uses SSV controllers from robust control theory. These controllers can read external signals from other controllers to coordinate multilayer operation. In addition, they allow designers to specify the discrete values allowed in each input, and the desired bounds on output value deviations. Finally, they accept uncertainty guardbands, which incorporate the effects of interference between the different controllers. We called this approach *Yukta*. To assess its effectiveness, we prototyped it in an 8-core big.LITTLE board. We built a two-layer SSV controller, and showed it was very effective. Yukta reduced the EDP and the execution time of a set of applications by an average of 50% and 38%, respectively, over what advanced heuristic-based coordinated controllers attain. We expect that the Yukta design can be applied to many computing environments.

CHAPTER 5: TANGRAM: INTEGRATED CONTROL OF HETEROGENEOUS COMPUTERS

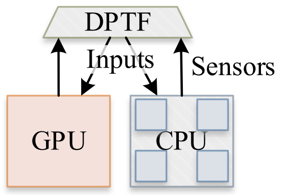
Large computers are integrated from heterogeneous subsystems like CPUs, GPUs and accelerators built by different vendors. In managing such systems, there is a tension between the need to quickly generate local decisions in each subsystem and the desire to coordinate the different subsystems for global optimization. Leading systems from IBM, Intel, and AMD manage their computers with centralized heuristic-based control. Centralization results in slow response. In addition, control is often ineffective because the separately designed subsystems do not expose enough information to the central controller. On the other hand, implementing global coordination among decoupled control is considered hard and is usually avoided. We now consider the problem of developing a fast, decentralized and coordinated resource control framework for heterogeneous systems. Then, we will describe the *Tangram* framework that solves the problem effectively.

5.1 COMPUTER CONTROL TODAY

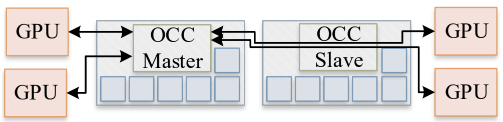
Controlling the operation of heterogeneous computer systems is a challenging problem that is currently addressed in different ways.

5.1.1 Organization

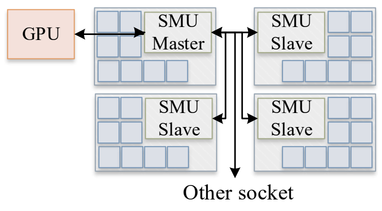
Most controllers from leading vendors are centralized (Figure 5.1). As shown in Figure 5.1a, Intel uses the Dynamic Power and Thermal Framework (DPTF) to manage the CPU and GPU in their multi-chip Core i7-8809G [51, 150]. The DPTF is a centralized kernel driver. Each chip exposes sensor data and allows DPTF to set its controllable inputs.



(a) Intel DPTF [150]



(b) IBM OCC [47]



(c) AMD SMU [9, 50]

Figure 5.1: State-of-the-art resource control architectures for heterogeneous computers from leading vendors.

Figure 5.1b shows the On Chip Controller (OCC) in IBM POWER9. It is a centralized hardware controller that actuates each on-chip core and the GPUs attached to the chip [47,151]. In a 2-socket system, one OCC becomes the master for global control, and the other a slave with restricted decision-making. The slave OCC is limited to sending sensor data and applying input values given by the master.

Figure 5.1c shows the hardware System Management Unit (SMU) design from AMD [9]. AMD's EPYC and Ryzen processors consist of one or more dies, each of which has one SMU. When multiple dies are used in a socket, one SMU becomes the master and the others are slaves, as with IBM. The slave SMUs handle only events like high temperatures or current, where quick response is necessary. The master SMU makes centralized decisions for the modules in all the dies in the system. In a two-socket system, there is a single master for both sockets.

When an on-die GPU is integrated with the CPU, AMD uses the Bidirectional Application Power Management (BAPM) algorithm, which is a centralized algorithm running on firmware to manage power between the CPUs and GPU [50]. With discrete GPUs, there is no communication channel between the CPU and GPU controllers. Therefore, system-level coordination needs to be handled through software drivers, as with the Intel design above.

Centralized hardware control is also the choice in research [52, 53, 54, 55, 56, 57, 58, 59, 92, 94, 111]. Unfortunately, centralized control is slow because data and decisions must cross chip boundaries, and experience contention at the single controller. It is also non-modular because integrating a new component or using a different configuration of the subsystems requires a full re-design of the controller.

As an alternative, Raghavendra et al. [61] describe a *Cascaded* design for power capping in datacenters. The proposal is shown in Figure 5.2. The datacenter is organized as a set of enclosures, each containing a set of server blades. A Group divider splits the total power budget among the enclosures. Then, the Enclosure divider in each enclosure splits its designated power level $P_{enclosure}$ among its blades. Then, each blade supervisor (Sup) receives its designated power level P_{blade}, and enforces it by providing a target for a PID controller. The PID controller changes the blade frequency to achieve the target. The enclosure divider and the blade supervisor always keep the power of the enclosure and blade at the respective $P_{enclosure}$ and P_{blade} values they receive.

While this design is scalable, it has the limitation that the dividers are not controllers that could optimize the system; they just divide the power budget. In addition, changing a blade's power budget requires a long chain of decisions. When it is necessary to increase the blade's power, the group divider first increases $P_{enclosure}$, after which the enclosure divider can raise P_{blade}. Then, the supervisor changes the target and, finally, the PID controller can increase frequency. As the group divider's decision loop is much slower than the innermost

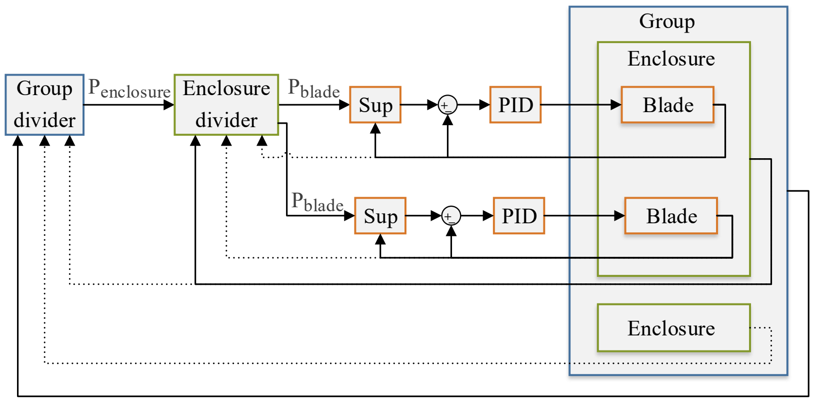


Figure 5.2: Cascaded control of a datacenter.

PID controller [61], there is a long delay between the need for a power increase and the actual increase. This may cause suboptimal operation and instability [152].

5.1.2 Controller Objectives

There are *Safety* controllers that protect the system from dangerous conditions (e.g., high current or voltage droops), and *Enhancement* controllers that optimize the execution for goals like power, performance, or EDP [9]. Safety controllers usually provide continuous monitoring and an immediate response, while enhancement controllers periodically search through a multidimensional trade-off space for the best operating point. In current industrial designs, these two types of controllers typically operate in a decoupled manner [9].

Designs from research typically focus on enhancement, disregarding interaction with safety mechanisms. Some exceptions are works that consider temperature as a soft constraint (e.g., [61, 94, 113]), or those that probabilistically characterize safety mechanisms like circuit breaker tripping (e.g., [58]).

5.1.3 Formal Control vs Heuristics

Current industrial designs typically use heuristics for resource control [3, 4, 47, 50, 153, 154]. A few use heuristics plus PID controllers [9, 46, 51, 81]. In most cases, controllers monitor one single parameter (i.e., output), like power or skin temperature, and actuate a single parameter (i.e., input), like frequency. Hence, they are SISO. Often, multiple controllers actuate the same input, such as the CPU frequency. In this case, the conflicting decisions are combined using heuristics. For example, IBM's OCC assigns each controller a vote and a majority algorithm sets the input [151].

Heuristics can result in unintended inefficiencies [78, 79, 92, 101, 111, 155]. Further, they make it difficult to decentralize resource control [51], which is necessary for fast response and modularity. Paul et al. [52, 53] show real examples of how multiple controllers using heuristics fail to coordinate in a system with a CPU and GPU. For instance, in a workload whose performance is limited by the GPU, CPU heuristics see low CPU memory traffic and boost CPU frequency. This does not improve performance and wastes power.

5.2 TANGRAM: DECENTRALIZED CONTROL

Our goal is to design and prototype a control framework for heterogeneous computers that is decentralized, globally coordinated, and modular. Decentralization is needed for fast control. However, it should not come at the expense of global optimization. Further, the framework should be modular to be usable in different computer configurations. These requirements rule out the conventional centralized and cascaded organizations. Moreover, for effectiveness, the controllers in this framework should combine safety and enhancement functionalities, be formal rather than heuristic-based, and be MIMO. We call our new framework *Tangram*.

In this section, we start by introducing the novel controller in Tangram, and then the Tangram modular control framework. Later, in Section 5.3, we build a Tangram prototype.

5.2.1 Controller Architecture

To the *Safety* and *Enhancement* types of controllers, we add a third one, which we call *Preconfigured*. Table 5.1 shows the controller differences and the control strategies they follow. Section 5.1.2 described safety and enhancement controllers. A preconfigured controller looks for a certain well-known operating condition. When the execution is under such a condition, the preconfigured controller uses a preset decision to bring the system to an optimal configuration. The priority of preconfigured controllers is lower than safety controllers but higher than enhancement controllers.

Table 5.1: Types of controllers.

	Safety	**Enhancement**	**Preconfigured**
Goal:	Hardware safety	Optimality	Optimality
Strategy:	Simple, preset	Complex, search based	Simple, preset
Priority:	Highest	Low	Medium
Operation:	Nearly always	Periodic	Nearly always
Response time:	Immediate	Fast	Immediate

In Tangram, we propose to build a controller that combines enhancement, safety, and preconfigured engines.

Enhancement Engine

We use robust control [91] to build a MIMO enhancement controller. The controller monitors all local outputs to be optimized, like performance, power, and temperature, and sets the local inputs to keep all outputs close to the desired targets. It works with a *Planner*, which changes these targets to match changing conditions and to optimize metrics combining multiple outputs like EDP. The combination of controller and planner is the enhancement engine (Figure 5.3).

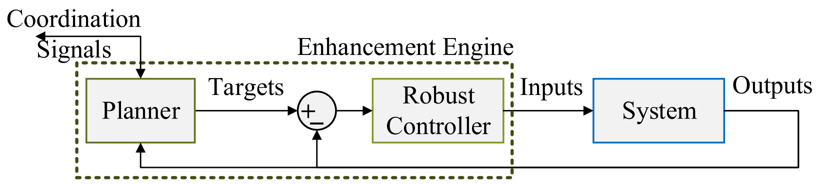


Figure 5.3: Enhancement engine.

For example, to minimize EDP, the planner can search along two directions: increasing performance targets more than power targets, or decreasing power targets more than performance targets. For each target point selected, the robust controller will determine what inputs can make the system outputs match the targets. The planner then computes the EDP and may select other performance and power targets that may deliver a better EDP. In Section A.2 of the appendix, we describe a generic search algorithm for the planner. Algorithms like Gradient Descent can also be used to search for the best targets.

The planner is also the point of communication with other controllers, if any, and exchanges coordination signals with them. It uses some of these incoming signals to generate the local targets.

Adding Safety and Preconfigured Engines

We add a safety engine that continuously monitors for hazards like high temperature or current. If the engine is triggered, it picks the most conservative values of the inputs. This is done without any search overhead. For example, if the computer overheats, the safety engine simply runs the cores at the lowest frequency.

Similarly, we add a preconfigured engine that continuously monitors for execution conditions that are well understood and for which there is an optimal configuration. If the engine is triggered, it sets the inputs to a predefined configuration, skipping any search by the enhancement engine. For example, if there is a single thread running, the engine boosts the active core's frequency, and power-gates the other cores.

Figure 5.4 shows the full architecture of our controller, with potentially multiple safety and preconfigured engines in parallel with the enhancement engine. All engines are connected to an arbiter. A mode detector chooses one engine by controlling the arbiter. Each engine can monitor potentially different outputs.

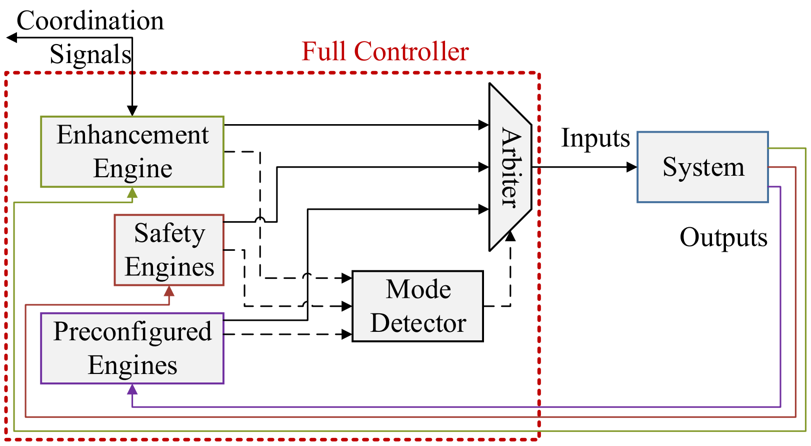


Figure 5.4: Proposed controller.

The mode detector uses the following priority order to select the engine that sets the system's inputs: (i) any active safety engine, starting from the most conservative one, (ii) any active preconfigured engine, starting from the most conservative one, and (iii) the enhancement engine.

At runtime, the enhancement engine optimizes the system. It may inadvertently trigger a safety engine, which then sets the inputs to the lowest values. The change induced by the safety engine is within the uncertainty guardband used in the controller's design. Once the hazardous condition is removed, the enhancement engine resumes operation but it remembers (using its state) to avoid further safety triggers. Thus, the enhancement engine can optimize inputs without repeated conflicts with the safety engines [106].

5.2.2 Subsystem Interface

Computers are organized as a hierarchy of subsystems, possibly built by different manufacturers. For example, a motherboard that contains a GPU and a CPU chip is a subsystem, and a multicore chip that contains multiple cores is also a subsystem. To build decentralized, globally-coordinated modular control, we propose that each subsystem has a controller with a standard interface.

Figure 5.5 shows the interface. To understand it, we logically break a subsystem into its controller and the rest of the subsystem, which we call the *Component*. The figure shows one subsystem with its controller and its component. The controller generates or receives three sets of signals:

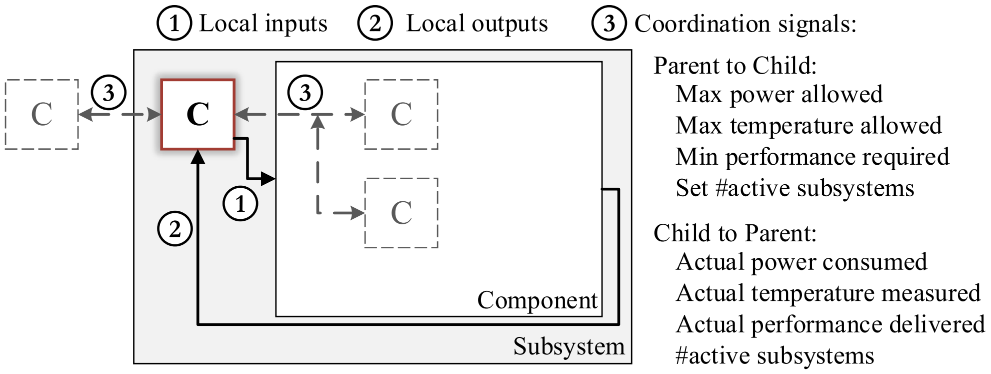


Figure 5.5: Proposed controller interface. In the figure, *C* means controller.

• **Coordination Signals ③**. These signals connect a controller with its parent controller and its potentially multiple child controllers. The figure shows a controller with two child controllers. The Coordination signals are shown on the right of the figure. From a controller to its child controllers, the signals set the operating constraints (i.e., the maximum power allowed, maximum temperature allowed, minimum performance required, and number of active subsystems). The child controllers use this information as constraints as they optimize their own components. From a controller to its parent controller, the signals report on the operating conditions (i.e., actual power consumed, actual temperature measured, actual performance delivered, and number of active subsystems). The parent controller uses this information to potentially assign new constraints to all of its children. The coordination signals use parameters readily available in current systems.

• **Local Inputs ①** and **Local Outputs ②**. These are the conventional signals that a controller uses to change and sense its component, respectively. They require no coordination and, therefore, can be manufacturer-specific. Examples of local inputs are frequency and cache size, and of local outputs are performance, power, temperature, and dI/dt. Figure 5.4 shows how the inputs are generated from the output measurements.

To build a modular control framework, the manufacturer of a subsystem has to include a controller that provides and accepts the standard coordination signals from parent and child controllers.

5.2.3 Tangram Control Framework

As a computer is built by assembling different subsystems hierarchically, the controllers of different subsystems are also connected hierarchically, exchanging the standard Coordination signals (Section 5.2.2). The result is the Tangram control framework. Figure 5.6 shows the framework – without the proprietary Local Input and Local Output signals – for a computer node that contains two modules, with one of the modules containing two chips.

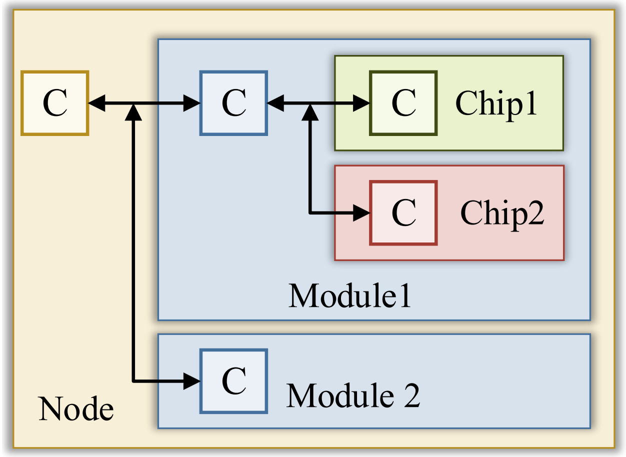


Figure 5.6: The decentralized, modular Tangram framework. *C* means controller.

The Tangram control framework is modular, fast, and globally coordinated. It is modular because each controller is built with knowledge of only its subsystem. For example, in Figure 5.6, the designers of Chip 1 and Chip 2 develop their controllers independently. Similarly, the Module 1 controller is developed without knowing about the Chip 1 and Chip 2 controllers. Further, changing a subsystem is easy – only the interfacing controllers need to be rewired and reprogrammed. For example, if we change Module 2, only the Node controller is affected.

The framework is fast because each controller makes decisions on its own subsystem immediately. This is unlike in cascaded designs where, to make a change that affects the local system requires a long chain of decisions (Section 5.1.1). It is also unlike centralized systems, where decisions are made in a faraway central controller.

Finally, the framework is globally coordinated because there are coordination signals that propagate information and constraints across the system. These signals are used differently than in cascaded systems. In Tangram, the local controller in a subsystem uses the constraints given by the coordination signals from the parent to identify the subsystem's best operating point; the local controller in a subsystem passes constraints to the child controllers. In the cascaded design discussed earlier (Section 5.1.1), instead, the divider simply provides, at each level, the exact parameters that fully determine the subsystem's operating point; the local controller at the leaf node tries to keep the outputs at this operating point.

5.2.4 Comparison to Contemporary Systems

The modular structure of Tangram may make the design appear obvious. Therefore, why do current systems not use a similar framework? A major reason is that their controllers do not use formal control. The use of a MIMO robust controller in each subsystem ensures that its optimizations work in the presence of other controllers in other subsystems. The controllers connected in a hierarchy can coordinate their actions. These benefits cannot be guaranteed by the current heuristic controllers used in individual subsystems, and simply using them together does not lead to cohesive decisions [51, 53].

5.2.5 Tangram Implementation

Tangram can be implemented in hardware or in software. For time-critical and hardware-specific measures such as DVFS, the controllers should be implemented in hardware or firmware, and signals should be carried by a special control network. Examples of such a network are AMD's SMU in EPYC systems [9] and IBM's OCC in POWER9 [47] (Section 5.1.1). For less critical measures, controllers can be implemented in software, and communication between controllers can proceed using standard software channels.

In a hardware implementation, it is not necessary to have dedicated pins and physical connections for every signal. A few ports and links are sufficient, as controllers can pass information in the form of `<property,value>` pairs.

While verifying a decentralized system is a challenging task in general, Tangram reduces verification cost because it uses formal control. Further, each Tangram controller is simple, compared to a single, large centralized controller.

5.2.6 Example of Tangram's Operation

Figure 5.7 is an example of how Tangram works. The figure considers a module composed of a CPU chip and a GPU chip. It shows the timeline of the actions of the three controllers, as they run in preconfigured, safety, or enhancement modes. The figure shows the coordination signals passed between the three controllers, which can be the local values measured (solid) or new constraints (dashed). For simplicity, we only consider power-related and activation/deactivation signals. As shown in the # Tasks timeline, the execution starts with zero tasks, then one CPU task appears, then one GPU task is added, and then many CPU tasks are added.

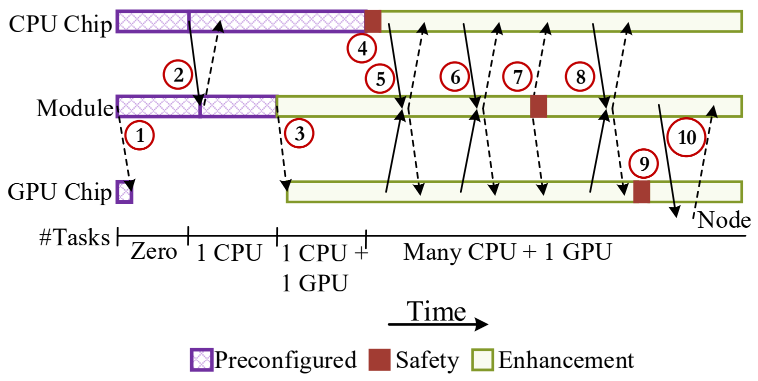


Figure 5.7: Example of coordination in Tangram. Solid arrows send measured local values, while dashed arrows provide new constraints.

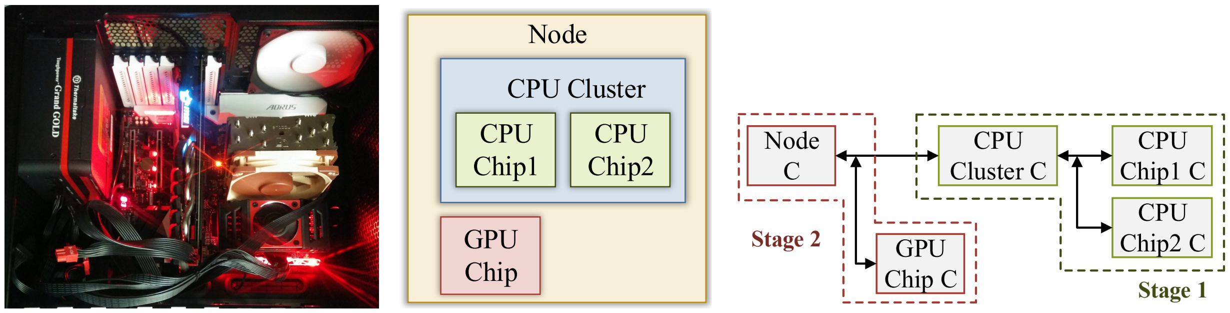
All controllers begin in preconfigured modes. At ①, the module controller turns the GPU chip off. When a CPU task arrives, the CPU controller changes to a new preconfigured mode. As the module realizes that there is one thread running (②), it changes the power assignment to the CPU chip, and changes to a new preconfigured state. When a GPU task arrives, the module changes to enhancement mode, wakes up the GPU chip and assigns it a power budget (③). The GPU controller enters enhancement mode. The GPU chip optimizes itself using the power assigned. When many CPU tasks arrive, there is a current emergency in the CPU chip, which causes the controller to enter safety mode (④). The CPU chip controller eventually transitions to enhancement mode. At ⑤, the module reads values from both chips and shifts power from the GPU chip to CPU chip. There is local optimization and some communication to find the best power across the module in ⑥. At ⑦, there is a thermal emergency in the module, which causes the controller to lower the power limits of the chips. On recovery, the module controller continues in enhancement mode, reading values and providing constraints (⑧). At ⑨, the GPU chip overheats and recovers from it, but the other subsystems are unaffected. At ⑩, the Node controller, which is the parent of the module controller, reads the module's state and provides new constraints for it.

5.2.7 Scalability of Tangram

Tangram's scalability is helped by the fact that Tangram uses MIMO control and has a hierarchical organization. Using MIMO helps scalability because we can increase the number of inputs and outputs of the controller, and the controller's latency increases only proportionally to the sum of the number of inputs and outputs. A hierarchical organization helps scalability because the depth of Tangram's network typically grows only logarithmically with the number of subsystems. Of course, with more subsystems, the root controller has longer timescales. However, this is generally not a problem because the time constants at which control is required also grow with large systems. For example, a chip's power supply cares about fine-grain current changes because it has a modest input capacitance, but a node's power supply only cares about longer timescales because it has a large capacitance.

5.3 TANGRAM PROTOTYPE

We prototype Tangram in a multi-socket heterogeneous server that we build using components from different vendors. We bought a computer motherboard from GIGABYTE [156], which we call the *Node*. The motherboard comes with an AMD Ryzen 7 1800X CPU cluster that has two quadcore processors with 2-way SMT [157]. To this, we add a GPU card from MSI [158] that contains an AMD Radeon RX 580 GPU [159]. Figure 5.8a shows a picture of the computer, and Figure 5.8b its organization. The system is a node with two subsystems: a CPU Cluster and a GPU Chip. The CPU Cluster has two quad-core CPU chips. The computer has subsystems from GIGABYTE, AMD, and MSI.



(a) Physical system. (b) Subsystems. (c) Controllers.

Figure 5.8: Tangram organization.

Figure 5.8c shows the Tangram framework. To demonstrate Tangram's modularity, we build it in two stages, similar to how we assembled the computer. In stage one, we design and interconnect the controllers in the CPU Cluster subsystem. In stage two, we design the controllers for the GPU Chip and the Node subsystems, and interconnect them with the stage one controllers to build the full Tangram network.

We built the controllers as software processes. They run as privileged software, accessing the System Management Units (SMUs) of the subsystems with internal calls. An alternative, higher performance implementation in hardware requires major changes to the testbed, as the SMUs are inside the chips.

The controllers read outputs and change inputs using Model Specific Registers (MSRs) [50] and internal SMU calls with proprietary libraries. Since AMD GPUs do not provide public access to dynamic performance counters, we read them using an internally developed library that intercepts OpenCL calls to identify the running kernel and its performance.

5.3.1 Stage One: CPU Cluster Subsystem

Figure 5.9 shows the Tangram network for the CPU cluster subsystem, with the different signals labeled. Table 5.2 shows the controllers' output and input signals that we use, based on the available sensors and actuators in our testbed. As we see, in a controller, the enhancement, safety, and preconfigured engines measure different outputs.

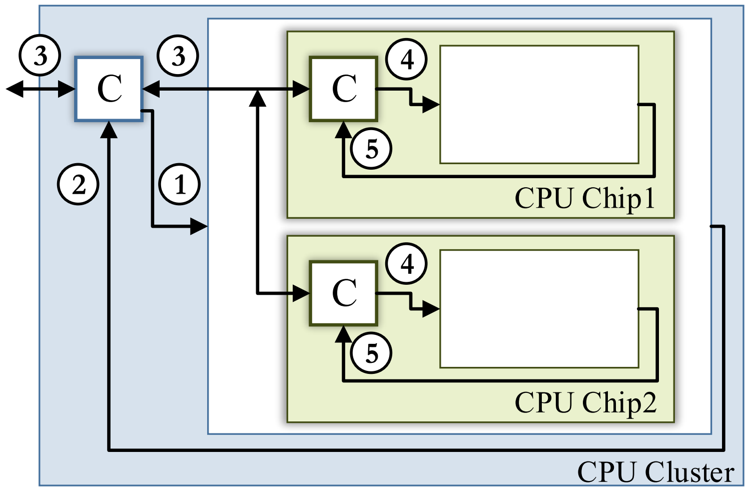


Figure 5.9: Tangram network in Stage One.

Table 5.2: Inputs and outputs of Stage One controllers.

Controller	Local outputs			Local inputs
	Enhancement	Safety	Preconfigured	
CPU Chip	⑤Chip performance, power	⑤Current, temperature	⑤ #threads	④ frequency, #cores on
CPU Cluster	②Cluster performance, power	②Current, temperature	②#threads	① Cluster frequency, #chips on

Consider Table 5.2. A CPU chip controller monitors many outputs. The enhancement engine monitors the chip's performance (measured in billions of instructions committed per second or BIPS), and power. The safety engine monitors the Thermal Design Current (TDC) used to prevent voltage regulator overheating [160], and the hotspot temperature. The preconfigured engine monitors the number of running threads. The controller sets two inputs, namely, the chip's frequency (2.2 GHz – 3.6 GHz) and the number of active cores (0 – 4).

The CPU cluster controller monitors the same outputs at its level, which combine the contributions of both chips, caches, and other circuitry in the cluster. The controller sets the cluster frequency of peripheral components (1.6 GHz – 3.6 GHz) and the number of active chips (0 – 2).

The coordination signals (③) measure the values and set the constraints discussed in Section 5.2.2.

5.3.2 Stage Two: Node Subsystem

Figure 5.10 shows the Tangram network for the whole node. It shows the details for the new controllers at this stage, namely, the controllers for the GPU Chip and Node. Table 5.3 lists the controllers' output and input signals, organized as in Table 5.2.

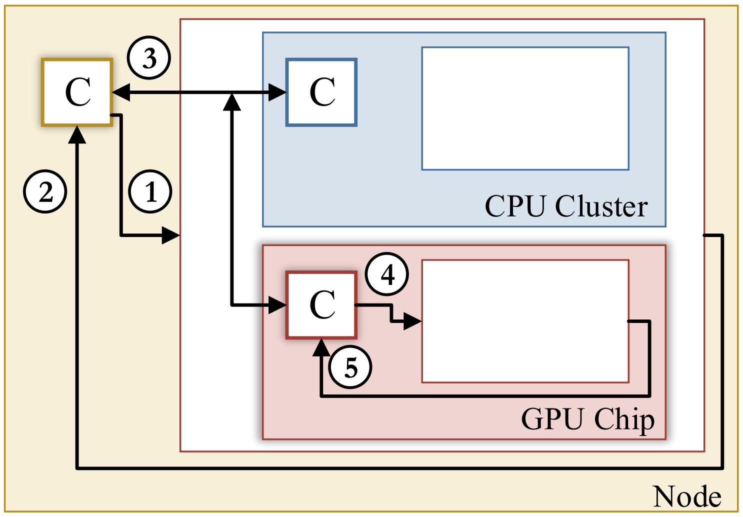


Figure 5.10: Tangram network in Stage Two.

As shown in Table 5.3, the GPU chip controller monitors the following outputs: the GPU performance and power (enhancement), the current and temperature (safety), and the number of kernels (preconfigured). The controller sets the frequency of the GPU compute units (300 – 1380 MHz) and of the graphics memory (300 – 2000 MHz). The node controller monitors similar outputs at its level (including whether the threads are CPU-type, GPU-type, or both), and sets the node frequency for the board's circuitry (300 – 2000 MHz).

Table 5.3: Inputs and outputs of Stage Two controllers.

Controller	Local outputs			Local inputs
	Enhancement	Safety	Preconfigured	
GPU Chip	⑤ GPU performance, power	⑤Current, temperature	⑤#kernels	④ Compute frequency, memory frequency
Node	② Node performance, power	②Current, temperature	②#tasks, task type	① Node frequency

The coordination signals (③) measure the values and set the constraints discussed in Section 5.2.2. The Node controller has no parent controller.

5.3.3 Structures in the Controllers

We now build the structures in each controller: enhancement engine (robust controller plus planner), safety engines, and preconfigured engines.

Enhancement Engine - Robust Controller: To design a robust controller, we need to (i) model its system (e.g., a CPU chip), and (ii) set the controller's input weights, uncertainty guardband, and output deviation bounds [91]. For the former, we use the System Identification modeling methodology [109] (Section 2.4). In this approach, we run training applications on the system and, during execution, change the system inputs. We log the observed outputs and the inputs. From the data, we construct a dynamic polynomial model of the system.

For the CPU chip and CPU cluster, we run two training applications from NAS and two from PARSEC. The obtained models have an order of 2 for the CPU chip and CPU cluster (i.e., $m = n = 2$ in Equation 2.6). For the GPU chip and node, we use two training applications from Chai [161]. The model orders for the GPU chip and node are 2 and 4, respectively.

Next, we need to set the controller parameters: input weights, uncertainty guardband, and output deviation bounds.

We specify the weight for each input in a subsystem based on the relative overhead of changing that input. For the CPU chip, turning a core on/off takes at least twice as long as changing the frequency. Hence, we use weights of 1 and 2 for the CPU frequency and number of active cores, respectively. For the CPU cluster, we also use weights of 1 and 2 for the CPU cluster frequency and number of active CPU chips. The GPU chip inputs are compute frequency and memory frequency. They have similar overheads and hence, we use weights of 1 for both. The node has a single input, namely the node frequency, and we set its weight to 1.

Next, we specify the uncertainty guardbands. The robust controllers must work with safety engines which can force the inputs to their minimum values when hazardous conditions are detected. For example, while the CPU chip frequency can range from 2.2 GHz to 3.6 GHz, the worst case is when the enhancement engine wants to set it to the maximum value and the safety engine sets it to the minimum one. Hence, we set the uncertainty guardband of every robust controller to 100%, to ensure optimality in this scenario.

With the system model, weights, and uncertainty guardband, MATLAB [108] gives the smallest output deviation bounds the controller can provide. We use the priority of outputs along with these suggestions to set the final output deviation bounds. In each controller, we rank performance bounds as less critical than power bounds. With these specifications, MATLAB generates the set of matrices that encode the robust controller (Equations 2.4 and 2.5). The output deviation bounds guaranteed by the robust controller in the CPU chip, CPU cluster, and GPU chip are $[\pm 15\%, \pm 10\%]$ for performance and power, respectively. For the node, the bounds are $[\pm 25\%, \pm 20\%]$ for performance and power, respectively.

With the model and these parameters, standard tools [108] generate the set of matrices that encode the robust controller (Section 2.3).

Enhancement Engine - Planner: The planner monitors local outputs and receives coordination signals from the parent and child controllers. With this information, it issues the best targets for all local outputs to the robust controller, and coordination signals to parent and child controllers. For example, the planner in a CPU chip's controller receives power, performance, temperature, and activation settings from the CPU cluster controller, and generates targets for its controller to optimize EDP. Our planners use the Nelder-Mead algorithm [162] described in Section A.2 of the appendix to search for the best targets under constraints received from the parent controller. We choose this search algorithm for its simplicity, effectiveness, and low resource requirements to run on firmware controllers [162].

Safety Engines: We consider two safety conditions: current and temperature. A hazard occurs if any exceeds the limits. The maximum values that we use for current (TDC) and thermal safety of each subsystem are shown in Table 5.4.

Table 5.4: Limits used in safety engines in all controllers.

	CPU Chip	**CPU Cluster**	**GPU Chip**	**Node**
Current (A)	15	25	30	50
Temperature ($^\circ$C)	55	65	60	70

If a hazard occurs in the CPU chip, the controller turns off all the cores except one, and sets the latter to the lowest frequency. If it occurs in the CPU cluster, the controller turns off one CPU chip and runs the other at the lowest frequency. If it occurs in the GPU chip or in the Node, the controllers set the frequencies to the lowest values.

Preconfigured Engines: We build the preconfigured engines of the different controllers to have the modes in Table 5.5.

Table 5.5: Modes of the different preconfigured engines.

Controller	Preconfigured Regime	Action
CPU chip	No active thread	Only one core on, which runs at the lowest frequency
	Single active thread	Only one core on, which runs at the highest frequency
CPU cluster	No active thread	One CPU chip can use up to 1/8 of its TDP; the other CPU chip is turned off
	Single active thread	One CPU chip can use up to 1/2 of its TDP; the other CPU chip is turned off
	#threads $\leq$ 8 (i.e., # of SMT contexts in a chip)	One CPU chip can use its full TDP; the other CPU chip is turned off
GPU chip	No active task	GPU chip goes to a low power mode
Node	CPU-only tasks	CPU cluster can use its full TDP; GPU chip can use up to 1/8 of its TDP
	GPU-only tasks	CPU cluster can use up to 1/8 of its TDP; GPU chip can use its full TDP

5.3.4 Controller Overhead and Response Time

To show the nimbleness of our prototype, we list the overhead and response time of the controllers. Table 5.6 lists the overhead of the four structures that comprise the CPU Chip controller. For each structure, the table lists the dimension, storage required, number of instruction-like operations in the computation needed to produce an output, latency of computation, and power consumption. A robust controller's dimension is the number of elements in its state (i.e., the length of vector x in Equation 2.4 of Section 2.3.1). A planner's dimension is the number of possible modes in the Nelder-Mead search in the appendix (Section A.2). From the table, we see that the storage, operation count, latency, and power values are very small – especially for the safety and preconfigured engines. The overheads of the controllers in the CPU cluster, GPU chip, and Node are similar.

We now consider the response time of the enhancement engines. Each enhancement engine has a robust controller and a planner (Figure 5.3). The response time of the robust controller

Table 5.6: Overheads of Tangram's CPU chip controller.

Structure	**Dimension**	**Storage**	**# Ops**	**Latency**	**Power**
Robust controller	9	1 KB	≈245	≈15 μs	10-15mW
Planner	5	125 B	≈350	≈25 μs	10-15mW
Safety engines	–	8 B	2	$< 1\,\mu$s	$<$ 1mW
Preconfigured engines	–	8 B	2	$< 1\,\mu$s	$<$ 1mW

includes reading outputs and targets, deciding on new inputs, and applying the new inputs. The response time of the planner includes reading outputs and coordination signals, deciding on targets, and communicating targets to the controller. Table 5.7 shows the measured response time of the robust controllers and planners in the enhancement engines of the different controllers in Tangram. For a parent controller/planner, the response time includes the time for its decisions to propagate through all the children and grandchildren until they affect the leaf robust controller's decision to change inputs. This may take multiple invocations of the leaf robust controller, which is activated every 50 ms. For comparison, we also show data for a centralized and a cascaded control framework that we implemented.

Table 5.7: Response time of the enhancement engines.

Subsystem	**Tangram**		**Centralized**		**Cascaded**	
	Robust Controller	Planner	Robust Controller	Planner	Robust Controller	Divider
CPU Chip	15 ms	15 ms	–	–	15 ms	–
CPU Cluster	115 ms	115 ms	–	–	–	165 ms
GPU Chip	15 ms	15 ms	–	–	15 ms	–
Node	515 ms	515 ms	200 ms	200 ms	–	665 ms

The Tangram robust controller and planner in the CPU chip and GPU chip have a response time of 15 ms. Hence, performance, power, and temperature can be controlled in a fine-grain manner. As we move up in the hierarchy of controllers, the response time increases. At the node level, the response times are close to 500 ms.

In *Centralized*, we place the single enhancement engine in the Node subsystem. Since the engine has to read many sensors, buffer the data, and change many inputs across the system, it has a sizable response time (200 ms). Hence, it is not suited for fast response.

In *Cascaded*, we build the design in Figure 5.2. Only the leaf subsystems (CPU and GPU chips) have robust controllers, and their response time is similar to those in Tangram. Higher levels in the hierarchy only have dividers, which set the power levels. A leaf controller cannot steer the system to new outputs until all the dividers in the hierarchy, starting from the topmost one, have observed a change in regime and, sequentially, agreed to a change in the power assignment. Because each divider is activated at longer and longer intervals as we move up the hierarchy, a round trip from the leaf subsystem to the outermost divider in the Node and back to the leaf takes 665 ms. Therefore, *Cascaded* has a long response time.

Table 5.7 shows that our software implementation of Tangram is fast. In mainstream processors, control algorithms are typically implemented as firmware running on embedded micro-controllers [1,9,47,51,163], and operate at ms-level granularity. Therefore, we envision the controllers in Tangram to be deployed as vendor-supplied firmware running on micro-controllers in their respective subsystems. This requires little change to existing hardware. Further, the storage overhead and number of operations from Table 5.6 indeed show that Tangram can be easily run as firmware on a micro-controller. With a firmware implementation, we estimate that Tangram's response times in Table 5.7 reduce by about one order of magnitude, providing much better real-time control. A firmware implementation would also lower the response times of the other frameworks in Table 5.7, but is unlikely to change the relative difference between the frameworks.

5.4 EVALUATING THE PROTOTYPE

5.4.1 Applications

We use the Chai applications [161], which exercise both the CPUs and the GPU *simultaneously*, unlike most benchmarks. They cover many collaboration patterns and utilize new features in heterogeneous processors like system-wide atomics, inter-worker synchronization, and load balancing of data parallel tasks. We use two applications for training (*pad* and *sc*) and five for evaluation (*bfs, hsti, rscd, rsct*, and *sssp*). For Stage One controllers, we run NAS 3.3 [164] and PARSEC 2.1 [165]. From NAS, we use two applications to train (*bt* with dataset D and *mg* with dataset C) and nine for evaluation (*dc* with dataset B, *cg, ft, lu, sp* and *ua* with dataset C, and *ep, is* and *mg* with dataset D). From PARSEC, we use two applications to train (*raytrace* with dataset native and *swaptions* with dataset simlarge) and eight for evaluation (*blackscholes, bodytrack, facesim, ferret, swaptions, fluidanimate, vips* and *x264*, all with dataset native).

5.4.2 Designs for Comparison

Our evaluation is comprised of three sets of comparisons, each evaluated using the appropriate subsystem of the prototype that can give us the most insights. The systems compared are: different enhancement engine designs on a CPU chip, different control architectures on a

CPU cluster, and different complete frameworks on our full prototype. In all cases, our goal is to minimize the EDP of the system under constraints of maximum power, temperature, and current in each subsystem.

Comparing Enhancement Engine Designs.

We compare our enhancement engine (which we call *Robust*) to alternative designs, such as *LQG* and *Heuristic* (Table 5.8) on a CPU chip. *LQG* is the Linear Quadratic Gaussian approach proposed in Chapter 3. *Heuristic* is a collection of heuristics that use a gradient-free search to find the inputs that optimize the EDP metric. Instead of using a controller or a planner, it approximates the gradient using the past 2 output measurements and navigates the search space. The search uses random-restart after convergence, to avoid being trapped in local optima. This design is based on industrial implementations [9, 50, 166].

Table 5.8: Comparing enhancement engine designs.

Strategy	Description
LQG	LQG controller from [92].
Heuristic	Industrial-grade gradient-free optimization heuristics [50].
Robust	Our enhancement engine of Figure 5.3

Comparing Control Architectures.

We take our proposed controller from Figure 5.4 and use it in Tangram, Centralized, and Cascaded architectures on a CPU cluster. *Centralized* uses a single instance of our Figure 5.4 controller in the CPU cluster. *Cascaded* follows the design by Raghavendra et al. [61]. It uses a controller in each leaf subsystem, and a divider in the CPU cluster. For *Centralized* and *Cascaded*, each subsystem has its own safety engine for fast response time, as in existing systems [9].

Comparing Complete Frameworks.

Finally, we compare complete framework designs on our full computer. The framework designs are built with combinations of the above control architectures and enhancement engine designs, as listed in Table 5.9. Specifically, *Tangram Robust* is our proposed framework with our robust controller. *Cascaded LQG* is a state-of-the-art design combining prior work [61, 92]. *Tangram LQG* uses our control framework with LQG-based controllers. *Tangram Heuristic* uses our control framework with controllers based on industry-class heuristics. For this complete framework evaluation, we use the Chai programs.

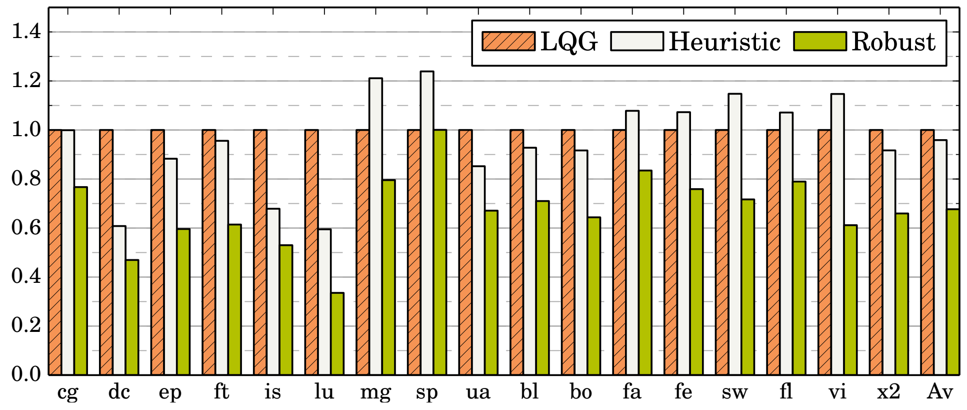
Table 5.9: Comparing complete control frameworks.

Control System	Description
Cascaded LQG	Architecture based on [61] with LQG controllers from [92].
Tangram LQG	Tangram architecture using LQG-based controllers.
Tangram Heuristic	Tangram architecture using industry-standard heuristics.
Tangram Robust	Tangram architecture with our proposed controllers.

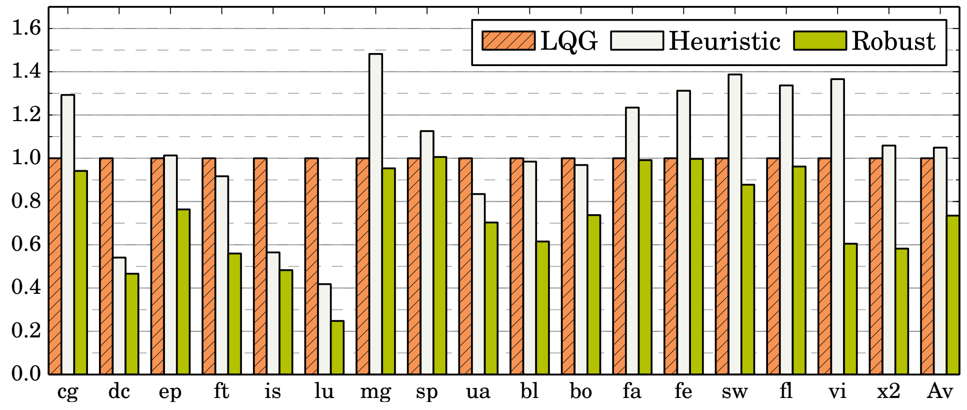
5.5 RESULTS

5.5.1 Comparing Enhancement Engine Designs

We compare the enhancement engine designs in Table 5.8 on a single CPU chip running NAS and PARSEC applications. Figures 5.11a and 5.11b show the execution time and EDP, respectively, with *LQG*, *Heuristic*, and *Robust* enhancement engines, normalized to *LQG*.



(a) Normalized execution time (lower is better).



(b) Normalized EDP (lower is better).

Figure 5.11: Comparing enhancement engine designs.

We see that *LQG* has the highest average execution time. To understand why, note that LQG controllers converge more slowly than robust controllers, as they are less capable of handling the relatively unpredictable execution of the applications. For example, LQG controllers in our design converge on the targets given by a planner after ≈6 intervals. For a robust controller, this value is 2.

This effect worsens when there is interference with safety engines. For example, when the LQG enhancement engine inadvertently increases current or temperature too much, the safety engines lower the frequency. As a result, performance falls much below its target. Then, the LQG engine responds aggressively to reduce its output deviations, but lacks the robustness to avoid future safety hazards. The planner does revise the output targets, but it is invoked only every 6 intervals of the LQG, because of the LQG's longer convergence time.

Heuristic also operates inefficiently, with oscillations between safety and enhancement engines as with *LQG*. It cannot effectively identify a configuration that is optimal and safe with heuristics alone. It has the highest average EDP. Finally, *Robust* has the fastest execution because the robust controller learns to optimally track output targets without safety hazards. Since it converges faster than *LQG* and keeps the output deviations within guaranteed bounds, the planner's search is effective and completes fast. Overall, Figure 5.11 shows the superiority of the *Robust* engine. On average, it reduces the execution time by 33%, and the EDP by 27% over *LQG*.

For more insight, Figure 5.12 shows a partial timeline of the power consumed by a CPU chip when running *ep*, an embarrassingly parallel NAS application that has a uniform behavior. The power is shown normalized to the maximum power that the CPU chip can consume in steady state. For this application, *LQG* and *Heuristic* have many oscillations due to switching between the safety and enhancement engines. However, the power in *Robust* converges rapidly and stays constant, thanks to the better control of its enhancement engine.

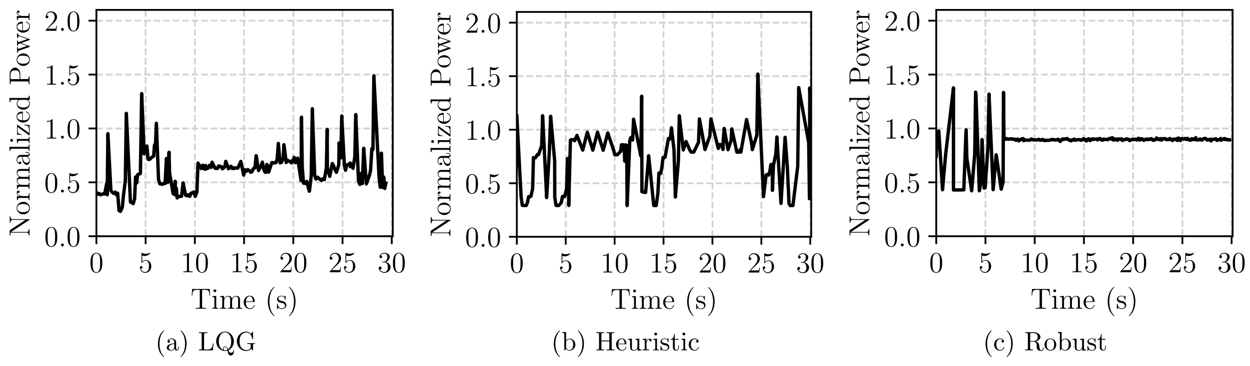
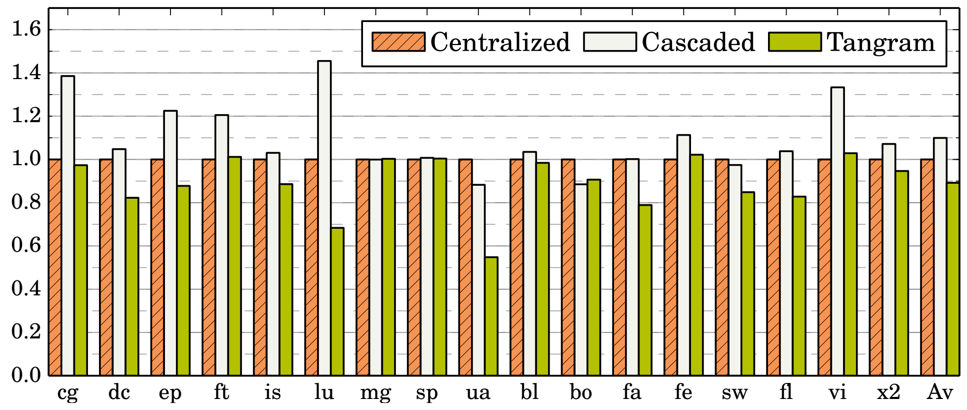


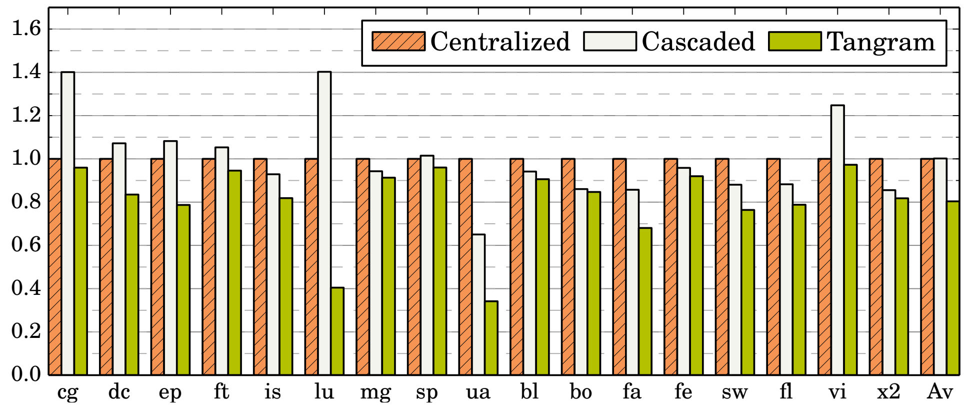
Figure 5.12: Partial timeline of the power consumed by *ep*.

5.5.2 Comparing Control Architectures

We compare Tangram to the *Centralized* and *Cascaded* architectures running applications in the CPU cluster. They all use our proposed controller from Figure 5.4 with a robust enhancement engine. Tangram uses a controller in each subsystem, *Centralized* uses a single controller in the CPU cluster, and *Cascaded* uses a controller in each leaf subsystem and a divider in the CPU cluster. Figures 5.13a and 5.13b show the execution time and EDP of the architectures, respectively, normalized to *Centralized*.



(a) Normalized execution time (lower is better).



(b) Normalized EDP (lower is better).

Figure 5.13: Comparing control architectures.

These architectures differ mainly in the response times of their robust controllers and planners/dividers. In *Centralized*, there is a single planner and robust controller that have long response times. As a result, they sometimes miss opportunities to adjust power and performance, even though they have a global view. This results in inefficient execution. In *Cascaded*, the CPU chip controllers respond fast, but their interaction with the next-level divider is slow. As a result, the targets used by the controller lag behind the system state, limiting efficiency. Moreover, a divider is not as effective as a controller in setting the targets.

Finally, *Tangram* has the lowest response times at all levels. Therefore, on average, it reduces the execution time by 11% and the EDP by 20% over *Centralized*.

To provide more insight, Figure 5.14 shows the power consumed by the entire CPU cluster as a function of time in *dc*, another NAS application. The power is shown normalized to the maximum power that the CPU cluster can consume in steady state. We see that *Tangram* uses higher power than *Centralized* and *Cascaded*. This is because it is responsive to application demands with its fast response time. While controllers communicate, they independently optimize their components for changing conditions by generating output targets and inputs fully locally. Therefore, the application finishes the earliest and even consumes the least energy.

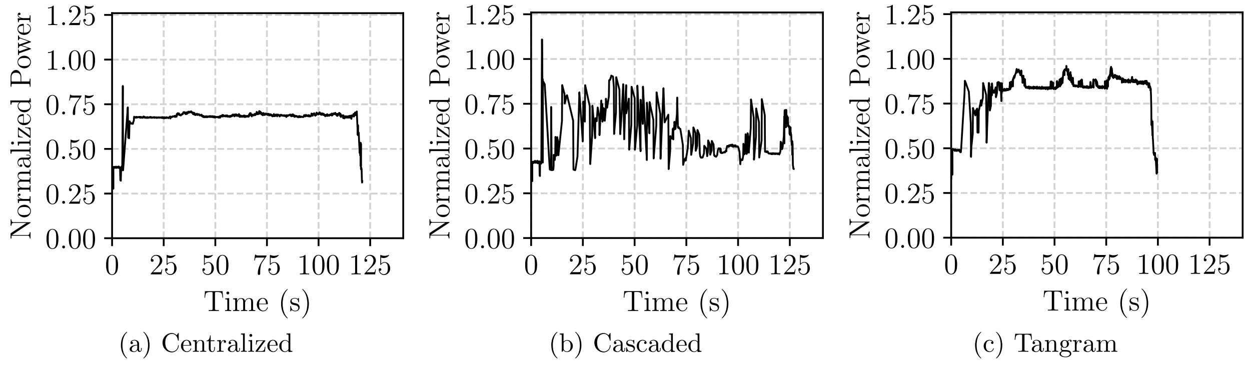


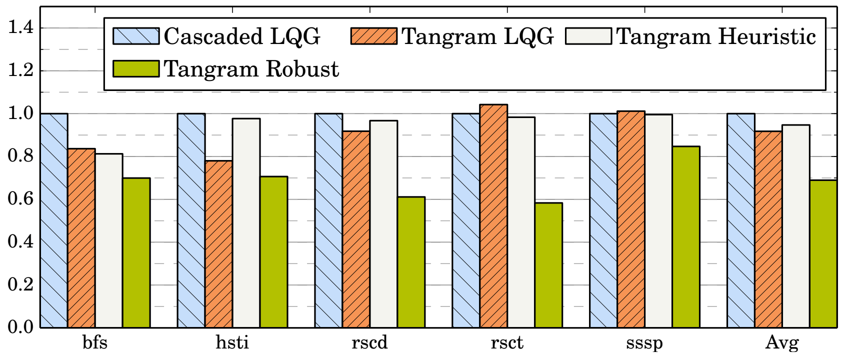
Figure 5.14: CPU cluster power in *dc* as a function of time.

Due to their longer response time, *Centralized* and *Cascaded* consume lower power even when the application can use higher power. *Cascaded* attempts to follow application demands, thanks to the low response times of the CPU chip controllers. However, the longer response time of the next level of control often results in stale targets, which triggers oscillations. This results in worse behavior than *Centralized*.

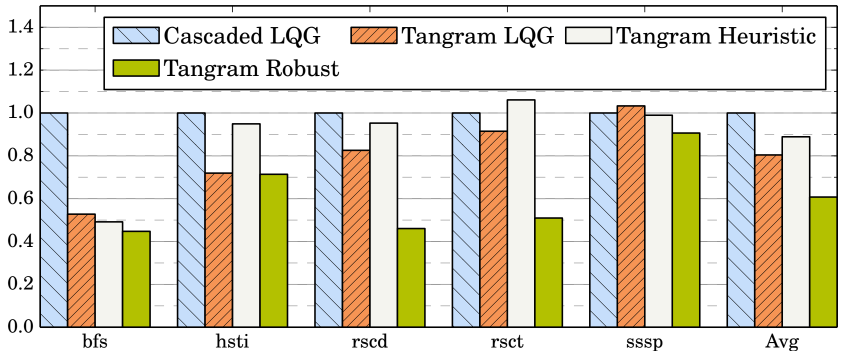
5.5.3 Comparing Complete Frameworks

Finally, we compare our proposed *Tangram Robust* framework to state-of-the-art designs (Table 5.9) on our full heterogeneous prototype. Figures 5.15a and 5.15b show the execution time and EDP, respectively, running the heterogeneous Chai applications. The bars are normalized to those of *Cascaded LQG*, which we consider the state-of-the-art.

If we compare *Cascaded LQG* to *Tangram LQG*, we see that the latter has a lower execution time and EDP. This is because *Cascaded* is a slow response-time architecture in large systems (Table 5.7). Moreover, it lacks the hierarchy of MIMO controllers that optimize each level of



(a) Normalized execution time (lower is better).



(b) Normalized EDP (lower is better).

Figure 5.15: Comparing complete control frameworks.

the control hierarchy. In particular, the *bfs* application suffers from this limitation.

Comparing *Tangram LQG* to *Tangram Heuristic*, we see that the latter is a worse design. With heterogeneity and complex application patterns (e.g., *hsti*), the heuristics are unable to identify system-wide efficient settings.

Finally, we see that our proposed framework (*Tangram Robust*) has the lowest execution time and EDP. This efficiency is due to two factors: its fast response time (Table 5.7), and the safety and optimality guarantees from using robust controllers at each level of the hierarchy. We see large gains even for programs like *rsct* that finely divide compute between the CPUs and the GPU. Overall, *Tangram Robust* reduces, on average, the execution time by 31% and the EDP by 39% over the state of the art *Cascaded LQG*. This makes *Tangram Robust* a significant advance.

For more insight, consider *rsct*, which has rapidly-changing GPU kernels and CPU threads. Figure 5.16 shows a partial timeline of the number of active CPU threads and GPU kernels. Figure 5.17 shows a partial timeline of the power consumed by the CPU Cluster and the GPU. The power is shown normalized to the maximum power that the node can consume in steady state.

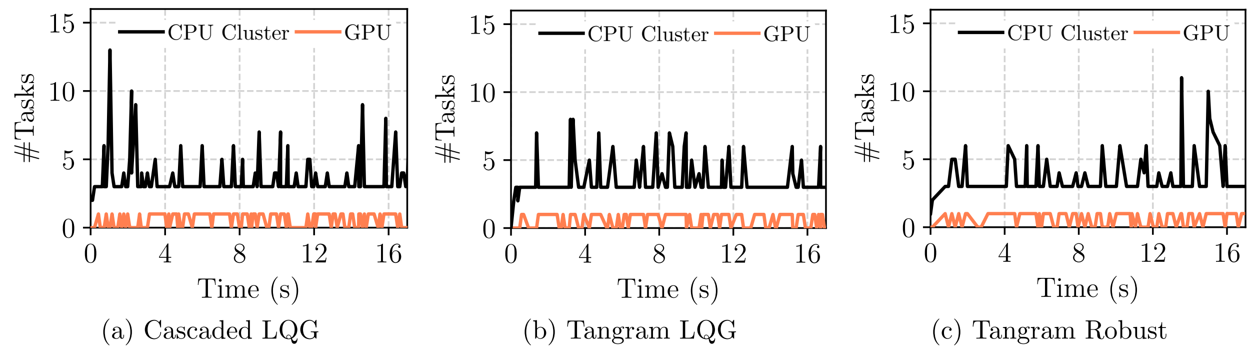


Figure 5.16: Partial timeline of the number of active threads in the CPU cluster and number of kernels in the GPU in *rsct*.

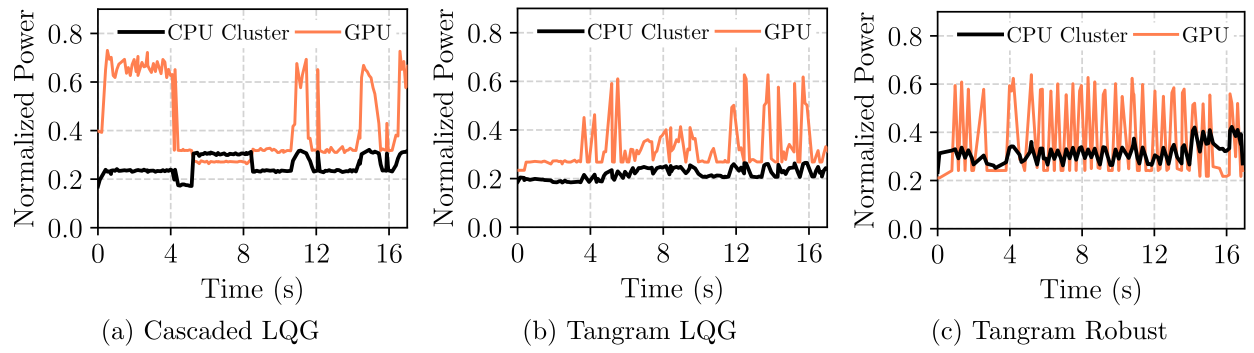


Figure 5.17: Partial timeline of the power consumed by the CPU cluster and the GPU in *rsct*.

The frequent peaks and valleys in the three charts of Figure 5.16 show that this application is very dynamic. The number of active threads in the CPU cluster and the number of kernels in the GPU changes continuously. Hence, all the frameworks try to continuously change the power assigned to the CPU cluster and the GPU, based on their activity. In many cases, they trigger the preconfigured engines, such as when only a few CPU threads are active or no GPU kernel is running.

However, as shown in Figure 5.17, the different frameworks manage power differently. In Figure 5.17a, we see that *Cascaded LQG* is slow to shift power between the CPU cluster and the GPU, and vice-versa. This framework reacts slowly for two reasons. First, the cascaded architecture intrinsically has a long response time (Table 5.7). Second, the LQG controller takes long to converge. As a result, many tasks in the CPU or GPU start and complete before the power to that subsystem is changed.

In Figure 5.17b, we see that *Tangram LQG* is more responsive. However, the slow LQG engine in *Tangram LQG* can hardly match the fast-changing execution. In contrast, *Tangram*

Robust in Figure 5.17c quickly reassigns power to the subsystem which best improves the overall EDP. This capability explains the programs' lower execution time and EDP in this framework.

5.6 CHAPTER SUMMARY

To control heterogeneous computers effectively, this chapter introduced Tangram, a new control framework that is fast, globally coordinated, and modular. Tangram introduces a new formal controller that combines multiple engines for optimization and safety, and has a standard interface. Building the controller for a subsystem requires knowing only about that subsystem. As a heterogeneous computer is assembled, the controllers in the different subsystems are connected, exchanging standard coordination signals. To demonstrate Tangram, we prototyped it in a multi-socket heterogeneous server that we assembled using subsystems from multiple vendors. Compared to state-of-the-art control, Tangram reduced, on average, the execution time of heterogeneous applications by 31% and their energy-delay product by 39%.

CHAPTER 6: MAYA: OBFUSCATING POWER SIDE CHANNELS WITH FORMAL CONTROL

Security is the prime concern in some environments, even above performance or power. Unfortunately, a computer's physical outputs like its power, temperature, or electromagnetic (EM) emissions leak information about the execution. Attackers can measure such outputs to recover a wealth of information like keystrokes, details of active applications or browsing history. Prior defenses try to keep the signals at constant levels or add noise, using special circuits. However, these techniques require new hardware, leaving existing systems in the field vulnerable. In this chapter, we describe *Maya* which, for the first time, uses formal control to dynamically re-shape the power of a computer to confuse attackers.

6.1 PHYSICAL SIDE CHANNELS

Physical side channels like power, temperature, and EM emissions can be used to uncover many details about an execution. For example, attackers have used these signals to infer the characters typed by a user [25], to identify the running application, the length of passwords on smartphones [24], and browser activity on personal computers [167], to disrupt operation in multi-tenant datacenters [168], and even to recover encryption keys from a cryptosystem [169].

Physical side channels originate because the dynamic power of the computer is proportional to its switching activity. This activity varies with instructions, groups of instructions, and application tasks, which all leave distinct fingerprints in the power trace [24, 25, 37, 170, 171]. Temperature and EM emissions are also directly related to the computer's power, and leave similarly-analyzable patterns [30, 172, 173].

6.1.1 Signal Measurement

Attackers can capture physical signals in many ways, most of which are non-intrusive. For example, attackers can use a malicious application that reads unprivileged hardware and OS counters for power or temperature [4, 30, 174]. In cloud systems, an application can use the thermal coupling between cores to infer the temperature or power profile of a co-located application using its own counters [30]. When power/thermal counters are unavailable, attackers can estimate power from OS metrics like utilization or from code analysis [39]. Malicious smart-batteries are another source of energy counters [25].

Power can also be measured by tapping AC electricity outlets [175], power distribution units (PDUs) [31], and public USB charging booths [26]. If proximity to the victim is possible, low-cost infrared thermometers and antennas can be used to read temperature and EM emissions respectively [176, 177]. With direct access to the computer, attackers can use multimeters or oscilloscopes [28]. Such high-end equipment is necessary to extract encryption keys.

Trojan hardware such as chips, co-processors, FPGAs, and other IP modules that are co-located with the target chip can also surreptitiously measure the target's chip-level power or temperature [40, 43, 44, 178]. Cloud systems share FPGAs across processors and accelerators, and can be exploited for remote power measurement [44, 178, 179]. In multicore systems, the hierarchical power management policies can be abused to act as power covert channels between cores [33].

Recently, it has even been shown that the detailed power activity of a computer can be measured from a different room in a large building, as long as the victim and the attacker computers are connected to the same power delivery network [42]. The attacker needs no physical access, and can measure power using widely-available commercial equipment. This greatly amplifies the risk of leaking information through power signals.

6.1.2 Signal Analysis

To extract sensitive information from signals, attackers can apply machine learning (neural networks), signal processing, and statistical analysis techniques [24, 25, 180]. Such techniques can identify information-carrying patterns in the signal, like its phase behavior and peak locations over time, and its frequency spectrum after a Fourier transform. To extract encryption keys, attackers either use simple power analysis (SPA) on a single trace [176], or differential power analysis (DPA) over thousands of traces [28, 29].

The timescale over which the signals are analyzed is determined by the information that attackers seek and the available measurement channels. Most attacks steal information like the details of the running applications, keystrokes, or browser data, and are performed with samples at intervals of milliseconds or more [24, 25, 26]. These are the timescales that our design focuses on. For cryptographic keys, it is necessary to record and analyze signals with samples at intervals of a few microseconds or less [28].

6.1.3 State-of-the-Art Defenses

Prior defenses against power side-channel attacks have mostly focused on encryption circuits. In practice, there are many attacks that are easier to mount, and which use system- or chip-level power measurements to steal sensitive information not related to encryption, like application activity, passwords and browsing data [24, 25, 26, 27, 29, 30, 31, 32].

Moreover existing defenses try to mask activity information by keeping physical signals at constant levels or by adding noise [28, 84, 85, 86, 87, 88, 89]. Unfortunately, all of these defenses need new hardware and, hence, cannot protect existing systems in the field.

Some of these defenses have additional limitations. For example, adding noise [89] or randomizing DVFS in the encryption circuits is easily countered by averaging multiple signal samples [90]. Further, some of these circuit defenses first measure the encryption circuit's power and then change their own activity to keep the overall power constant. Unfortunately, since the defense reacts only after observing the power changes, they too, cannot fully hide application activity [87].

It is possible to implement software versions of these defenses to defend against information leaking through chip-level or system-level power signals. However, as we will show later, these software schemes also have limitations.

An alternate strategy is to modify applications so that they do not leak information through the physical signals [41]. This is possible for a few critical applications (e.g., OpenSSL) but is impractical for the rest – like browsers, videos or cameras. To our knowledge, there are no defenses that can be readily used in existing machines against power side channels in an application-transparent manner.

There is an urgent need to develop effective defenses against power side channels that do not rely on special hardware, and which can be implemented as firmware or privileged software in an application-transparent manner. It is relevant to note that many common attacks that steal personal data like keystrokes or browser activity, analyze signals by sampling at intervals of several milliseconds or longer — suggesting that a firmware- or software-level defense is a good choice.

6.2 THREAT MODEL

We consider power side-channel attacks that perform signal analysis at the timescale of milliseconds, and which use pattern recognition techniques such as machine learning, signal processing, and statistics to analyze the signal. As demonstrated in the recent attack [42], such attacks do not need physical access and can use widely-available commercial equipment. These attacks can steal information like the identity of the running applications, the keystrokes typed, and the browser data accessed. This threat model covers the majority of attacks [24, 25, 26, 27, 30, 33, 37, 42, 44, 168, 180, 181] described in Section 6.1.1, except for those attacks identifying encryption keys [41, 169, 176, 177, 178]. The latter attacks are harder to mount, and typically need more expensive equipment and detailed knowledge of the cryptosystem being attacked.

We assume that attackers can know the algorithm used by Maya to reshape the computer's power. They can run Maya's algorithm and see its impact on the time-domain and frequency-domain behavior of applications. Using these observations, they can develop machine learning models to adapt to the defense and try to defeat it.

Finally, we assume that the firmware or privileged software that implements the control system to reshape the power trace is uncompromised. In a software implementation, the OS scheduler and DVFS interfaces need to be uncompromised.

6.3 OBFUSCATING POWER WITH CONTROL

We propose that a computer system defend itself against power attacks by distorting its power consumption. Unfortunately, this is hard to perform successfully because simple distortions like adding noise can be removed by attackers using signal processing. This is especially the case if, as we assume in this work, the attacker knows the general defense algorithm used to distort the signal. Indeed, past approaches have been unable to provide a solution to this problem. In this chapter, we propose the new approach of using *formal control* to re-shape power. In the following, we describe the architecture of Maya, the rationale behind using formal control, and the generation of effective distortions.

6.3.1 Maya Defense Architecture

Figure 6.1 shows the Maya architecture. Maya has a *Mask Generator*, a *Controller*, and mechanisms or inputs to change the power of a computer that is running an application. At each time-step, the mask generator computes the target power to mislead attackers and communicates it to the controller. The controller reads this target and the actual power consumed by the computer as given by the sensors. Then, it actuates all the inputs so that power is brought to the target.

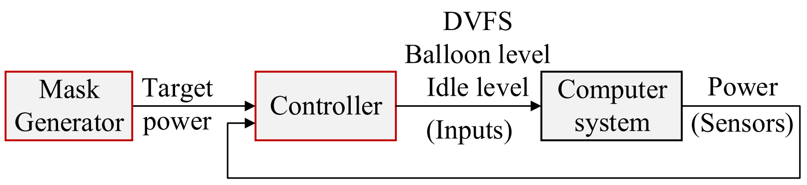


Figure 6.1: High-level architecture of Maya.

The inputs that the controller actuates are the levels of DVFS, the *balloon* task, and the idle activity. A balloon task is one that performs power-consuming operations (e.g., floating-point operations) in a tight, tunable loop. The balloon level determines the number of power-consuming operations. The idle activity level determines the percentage of processor cycles in which the processor is forced into an idle state.

To understand the environment targeted by Maya, consider Table 6.1. The table shows two types of power side-channel environments, which we call *Mainstream* and *Specialized*. Our envisioned Maya design targets the Mainstream environment.

Table 6.1: Two types of power side-channel environments.

Characteristic	Mainstream	Specialized
Attacks	[24, 25, 26, 27, 30, 33, 37, 42, 44, 168, 180, 181]	[41, 169, 176, 177, 178]
Attacker's sensors	counters, electric line tapping	Oscilloscopes, on-die trojan circuits
Signal analysis	$\geq$milliseconds	$\leq$microseconds
Controller type	Matrix-based controller in firmware or privileged software	Table-based controller in hardware
Controller response time	5–10 μs	$\approx$ 10 ns
Example actuations	Change frequency and voltage, regulate balloon and idle levels	Insert compute instructions and bubbles in pipeline
Example uses	Hide what application runs or the keystrokes typed	Hide features of a crypto algorithm

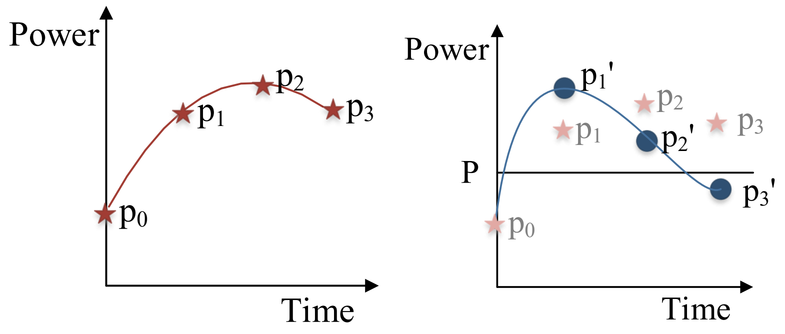
In Mainstream, attackers measure power with methods like reading counters or tapping electric power outlets. Since the signal analysis is at the granularity of a few milliseconds, one can use typical matrix-based controllers, as described in Equations 2.4 and 2.5. They are implemented in firmware or privileged software. The controller can respond in 5–10 μs, and can actuate by setting the DVFS level, and regulating the balloon and idle activity levels. This implementation can hide information like the application running or the keystrokes typed. This environment is the focus of this dissertation, as it is the easiest to set up and is widely used.

Table 6.1 also shows the Specialized environment, which would require a different design for Maya. Here, attackers use better sensors, such as oscilloscopes or on-die trojan circuits, and perform signal analysis at the nanosecond timescale. In this case, the controller would have to be fast, and hence, cannot use the matrix-based approach. Instead, it has to use a table of precomputed values, from which it quickly reads the action to be taken. This controller has to be implemented in hardware and have a response time of no more than $\approx$10 ns.

Possible actuators in this environment are hardware modules that insert compute-intensive instructions or bubbles into the pipeline. With such fast actuation, this implementation could be used, e.g., to prevent information leaking from crypto-algorithms. We do not consider this environment in this dissertation.

6.3.2 Why Use Formal Control?

Formal control is necessary to reliably keep the computer's power close to the target power given by the mask generator. To understand the importance of formal control, consider the following scenario. We measure the power consumed by an application at fixed timesteps to record a trace, as shown in Figure 6.2a. To prevent information leakage, the trace must be distorted into a different, uncorrelated shape using the balloon application and idle activity.



(a) Original (b) Distorted

Figure 6.2: Example of the power trace for an application.

One way to mislead the attacker is to try to keep the power consumption at constant level P (Figure 6.2b). To achieve this, we can measure the difference between P and the actual power p_i at each timestep, and schedule a combination of balloon thread and idle level based on $P - p_i$. Unfortunately, this approach is too simplistic to be effective. It ignores how the application's power itself changes. For example, when a balloon thread is scheduled at the 0^{th} timestep based on $P - p_0$, the power in the 1^{st} timestep may end-up being p'_1 rather than our target P.

If this control algorithm is repeatedly applied, it will always miss the target and we will obtain the trace in Figure 6.2b, where the measured power is not close to the target, and in addition, has many features of the original trace.

An approach that uses control theory is able to get much closer to the target power level. This is because the controller makes more informed power changes at every interval based on history. Moreover, the controller can easily set multiple inputs at a time for accurate control.

To understand why, we rewrite the equations of controller operation (Equations 2.4 and 2.5 from Section 2.3.1) slightly:

$$\begin{aligned} State(T+1) &= A \times State(T) + B \times Error(T) \\ Action(T) &= C \times State(T) + D \times Error(T) \end{aligned} \tag{6.1}$$

The second equation shows that the action taken at time T (in our case, regulating the balloon and idle activity) is a function of the tracking error observed at time T (in our case, $P - p_0$) *and* the controller's state. The state is a summary of the controller's experience in regulating the computer's power. The new state generated for the next timestep is determined by the current state and error. Thus, the "accumulated experience" in the state helps to get closer to the target.

Further, the controller's actions and state evolution are influenced by the matrices A, B, C, and D, which were generated when the controller was designed. That process included running a set of *training* applications while scheduling the balloon and idle threads and measuring the resulting power changes. Consequently, these matrices embed the intrinsic behavior of the applications under these conditions.

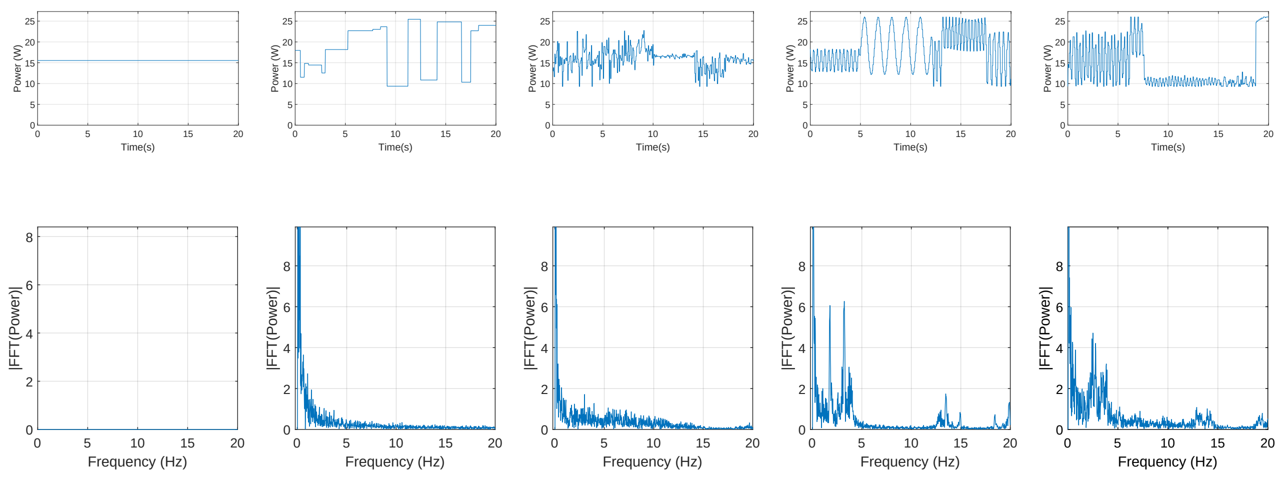
Finally, with formal control, the outputs can be kept close to the targets even when runtime behavior is unpredictable [91], which is often the case with computers. Overall, with a formal controller, the resulting power trace will be much closer to the target. If the target signal is chosen appropriately, the attacker cannot obtain application information.

6.3.3 Generating Effective Targets (Masks)

It is important to construct the target power function (or *mask*) such that it can hide application activity effectively. Consider what happens if the target is simply set to a constant. As the application activity changes, any method to maintain the computer's power at a fixed level would have to first observe power deviating from the target, and then set the inputs accordingly. Hence, the output signal would have power activity leaking at all change-points in the application.

On the other hand, choosing a random target power at every timestep is not a good design either. The attacker could run the application many times, and then use signal processing techniques to remove the noise. Then, the native change-points in the application would stand out. Therefore, the targets must be changed intelligently to hide such inadvertent leakage, and this is the role of Maya's mask generator.

An effective mask must hide information in both the time domain and the frequency



(a) Constant (b) Uniformly Random (c) Gaussian (d) Sinusoid (e) Gaussian Sinusoid

Figure 6.3: Examples of different masks. In each case, the time-domain curve is at the top, and the frequency-domain one at the bottom.

domain (i.e., after obtaining its FFT). We postulate that such a mask must have three properties. First, the mask should have several phases, each with a different mean level and variance (e.g., Figure 6.3c top).

Second, the phase transitions must have different rates – from smooth to abrupt. As a consequence, the FFT of the mask is *spread* over a range of frequencies (e.g., Figure 6.3c bottom). If the mask has the first and second properties, the resulting power signal will have many artificially-induced change-points that are indistinguishable from the original ones, effectively hiding them.

Finally, the mask must have repetitive activity with varying periodicity. This is to create several *peaks* in the power signal's FFT (Figure 6.3e bottom). Applications naturally ccreate peaks in the FFT if they have loops. By introducing repetitive activity, any natural peaks are overwritten and/or hidden.

We now examine generating a mask with the above properties using standard signals. Table 6.2 lists some well-known signals, showing whether each signal changes the mean and the variance in the time domain, and if it creates spread and peaks in the frequency domain. Figure 6.3 shows segments from each signal graphically.

A *Constant* signal (Figure 6.3a) has no change in either time or frequency domain. As discussed earlier, perfectly constant power cannot be realized in practice and information leaks at application change-points when a constant target is used.

In a *Uniformly Random* signal (Figure 6.3b), a value is chosen randomly from a range, and

Table 6.2: Some standard signals and what they change in the time and frequency domains.

	Time-domain		Frequency-domain	
Signal	Mean	Variance	Spread	Peaks
Constant	–	–	–	–
Uniformly Random	Yes	–	Yes	–
Gaussian	Yes	Yes	Yes	–
Sinusoid	Yes	Yes	–	Yes
Gaussian Sinusoid	Yes	Yes	Yes	Yes

is used as a target for a random duration. After this period, another value and duration are selected, and the process repeats. This signal changes the mean but not the variance in the time domain. In the frequency domain, the signal is spread across a range but has no peaks. This mask too, is not a good choice because any repeating activity in the application would be hard to hide in the time domain signal.

The *Gaussian* noise (Figure 6.3c) is constructed by sampling values from a Gaussian distribution whose mean and variance are randomly changed over time. The resulting FFT is spread over multiple frequencies, but does not have peaks.

The *Sinusoid* signal (Figure 6.3d) generates a sinusoid and keeps changing the frequency, amplitude, and the offset randomly with time. This signal changes the mean and variance in the time domain. In the FFT, it has clear sharp peaks at each of its sinusoid frequencies. However, there is no spread. Therefore, this signal is not effective at masking abrupt application changes.

Finally, the *Gaussian Sinusoid* (Figure 6.3e) is the addition of the previous two signals. This signal has all the properties that we want (Table 6.2): it changes the mean and variance in the time domain, and has spread and peaks in the frequency domain. Specifically, consider the FFT plots. The Gaussian signal (Figure 6.3c) has a noisy spectrum that is spread across a continuous range of values. In contrast, the Sinusoid signal (Figure 6.3d) has sharp and tall peaks. Therefore, the combination of the two signals (Figure 6.3e) results in a spectrum that has peaks that are both large and spread across a range. This is the mask that we propose.

6.4 IMPLEMENTATION ON THREE SYSTEMS

We implement Maya to protect the three different computers listed in Table 6.3. *Sys1* is a consumer-class machine with 6 physical cores, each with 2-way SMT, totaling 12 logical cores. *Sys2* is a server with 2 sockets, each having 10 cores of 2-way SMT, for a total of 40 logical cores. *Sys3* is another consumer-class machine with 4 physical cores, each with 2-way SMT.

On all systems, the architecture of Maya is the same, as shown in Figure 6.1. We target the Mainstream attack environment described in Table 6.1. Therefore, the controller and mask generator run as privileged software, actuating software-level parameters.

Table 6.3: Implementation platforms.

Name	Configuration
Sys1	Sandy Bridge (12 cores) + CentOS 7.6
Sys2	Sandy Bridge (40 cores) + CentOS 7.6
Sys3	Haswell (8 cores) + CentOS 7.7

In the systems, the Maya controller measures the power in the processors plus L1 caches (Sys1), in the package (Sys2), and in the chip (Sys3) using RAPL [182] every 20 ms. It actuates three inputs: the DVFS level of all cores, the percentage of idle activity, and the balloon power level. DVFS levels are set through the `cpufreq` utility [149], and can change from 1.2 GHz to 2.0 GHz on Sys1, from 1.2 GHz to 2.6 GHz on Sys2, and from 800 MHz to 3.5 GHz on Sys3, with 0.1 GHz increments in all cases.

The idle activity level is changed using Intel's `powerclamp` driver interfaces [183], and can be set from 0% to 48% in steps of 4%. The powerclamp system launches as many kernel-level threads as the number of cores. These threads repeatedly displace other running threads and force the cores into idleness, until the desired level of idleness is achieved.

We develop a simple balloon application that runs floating-point operations in a loop. The percentage of the balloon activity is set using a `sysfs` file and can change from 0% to 100% in steps of 10%. The balloon application first spawns as many threads as the total number of cores. Then, in the main loop, the master thread reads the desired balloon activity and configures each thread to run a loop of matrix multiply operations for a few milliseconds followed by sleep cycles. If the desired power balloon level is high, the fraction of sleep is low and vice-versa. One iteration of the main loop (read level, run compute-sleep loop), takes around 10 ms. The balloon threads are created with OpenMP, and are run with root priority.

Maya introduces performance overheads in the system. The slowdown comes because the idle threads and balloon threads can interrupt and displace

the application tasks. The controller and mask generator, by themselves, are simple functions; their overheads, as we will see in Section 6.6.4, are low. Maya's slowdown can be reduced by selectively activating Maya as needed, similar to the power governors in Linux [149]. However, in this dissertation, we show the worst-case performance overheads, which is when Maya is always on.

6.4.1 Designing the Controller

We design the controller using robust control [91]. For this, we need to: (i) obtain a dynamic model of the computer system running the applications, and (ii) set three parameters of the controller (Section 2.3.1), namely the input weights, the uncertainty guardband, and the output deviation bounds [91].

To develop the model, we use the system identification [109] experimental modeling methodology as before. We perform system identification by running two applications from PARSEC 3.0 (*swaptions* and *ferret*) and two from SPLASH2x (*barnes* and *raytrace*) [165] on Sys1. The models we obtain have a dimension of 4.

The input weights are set depending on the relative overhead of changing each input. In our system, all inputs have similar changing overheads. Hence, we set all the input weights to 1. Next, we specify the uncertainty guardband by evaluating several choices. For each uncertainty guardband choice, MATLAB tools [108] give the smallest output deviation bounds the controller can provide. Based on insights from Chapters 3 and 4, we set the guardband to be 40%, which allows the output deviation bounds for power to be within 10%.

With the model and these specifications, standard tools [108] generate the A, B, C, and D matrices that encode the controller (Section 2.3.1). The controller's dimension is 11 i.e., its state vector in Eq. 2.5 has 11 elements. The controller runs periodically every 20 ms; we set this duration based on the update rate of RAPL sensors and the latencies to change inputs.

6.4.2 Mask Generator

As stated in Section 6.3.3, we use a Gaussian sinusoid mask (Figure 6.3e) to generate the targets. This signal is the sum of a sinusoid and a gaussian, and its value at any time T is:

$$Offset + Amp \times \sin(\frac{2\pi \times T}{Freq}) + Noise(\mu, \sigma) \tag{6.2}$$

where the Offset, Amp, Freq, μ and σ parameters keep changing. Each of these parameters is selected at random from a range of values, subject to two constraints. First, the maximum power target is always below the Thermal Design Power (TDP) of the system. Second, the sinusoid's frequency ($Freq$) cannot exceed 25 Hz, because the power measurement rate itself is 50 Hz (from the 20 ms sampling interval). The power measurement has to be at least twice as fast as the sinusoid (Nyquist criteria).

Once a particular set of parameters is chosen, the mask generator uses them for N_{hold} samples, after which the parameters are updated again. N_{hold} itself varies randomly between 6 to 120 samples.

6.5 EVALUATION METHODOLOGY

6.5.1 Machine Learning Based Power Attacks

We consider multiple common attacks based on machine learning as listed in Table 6.4. These attacks try to identify which application is running on the machine, which video is being encoded, and what is the user's browsing activity. They are widely reported in prior work [24, 25, 26, 27, 167, 175, 184]. The defense (i.e., the controller) samples power at 20ms intervals because RAPL provides reliable measurements only at this timescale. The attacker also samples power at 20 ms intervals except in Sys3 where, as we will see, samples at 50 ms intervals because the measurements are taken from an AC power outlet cycling at 60 Hz.

Table 6.4: Machine learning based power attacks.

Attacker's goal	Victim computer	Signal capturing method
Detect the active application	Sys1	Counters
Identify video being encoded	Sys2	Counters
Identify webpages visited	Sys3	AC outlet power

1. Detecting the active application: This is a well-known fundamental attack [24, 25, 175, 184]. Attackers capture many power traces of the applications they want to identify and build a machine learning classifier to recognize the application running from a power trace. We launch this attack on Sys1 using unprivileged RAPL counters to measure power. As in prior work, we assume that a malicious module installed by the attacker captures these counters [24, 25].

We run applications from PARSEC 3.0 (blackscholes, bodytrack, canneal, freqmine, raytrace, streamcluster, vips) and SPLASH2x (radiosity, volrend, water_nsquared, and water_spatial) with native datasets and record 1,000 traces for each application. From each trace, we extract multiple segments of 15,000 RAPL measurements, and average the 5 consecutive measurements in each segment to remove effects of noise. For accurate training, we quantize the power values into 10 levels and encode the traces in one-hot format. We use 60% of the data we collect for training, 20% for validation, and report the results for the remaining 20% test set.

For classification, we use a three-layer multilayer perceptron (MLP) neural network. The network uses ReLU units for its hidden layers and the output layer uses Logsoftmax.

2. Detecting video data: There is one application, a video encoder, that operates on multiple videos, and the attacker's goal is to identify the video being encoded. This is also a common attack [24, 25, 26, 27, 184]. We perform it on *Sys2* targeting the ffmpeg video encoder [185]. As with the previous attack, power signals are captured through RAPL.

We take four common test videos saved in raw format: tractor, riverbed, wind and sunflower [186]. We transcode each video using ffmpeg's x264 compression for 200 runs and record the power traces. From each trace, we obtain multiple windows of 1000 samples long, quantize the power values, and use one hot encoding to train our MLP classifier.

3. Detecting webpages: This is a popular attack [24, 25, 26, 167], and we set it up on *Sys3*. Unlike the previous attacks, we capture the power traces by measuring the *AC electric outlet power*. Figure 6.4 shows a picture of our test platform. We tap the electric outlet used by the victim computer with wires connected to a multimeter. This multimeter (Yokogawa WT310) passes its measurements into another computer using a USB connection. This is a powerful and stealthy attack because information is obtained by simply rigging electricity outlets without installing any modules on the victim. Since the natural frequency of AC is 60 Hz (corresponding to 16.6 ms cycles), the multimeter collects the root mean square (RMS) power samples every 50 ms (i.e. for every three AC cycles).

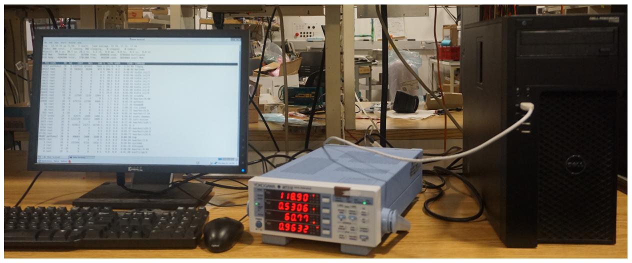


Figure 6.4: Tapping AC electric outlet power.

We record 100 power traces when visiting the popular websites google.com, ted.com, youtube.com, chase.com, ieeexplore.ieee.org/Xplore/home.jsp (IEEE Xplore), amazon.com and paypal.com using the Google Chrome browser. Each trace is nearly 15 seconds long. Unlike before, we use the signals' FFT to train our MLP because browser activity has varying rates of change in a short duration. FFT captures this better.

6.5.2 Designs Compared

Table 6.5 lists the designs whose security we compare. In *Noisy Baseline*, each run of the applications is executed with random DVFS and idle activity levels that are configured before the application runs. Since the application is frequently interrupted to enforce idleness, the power trace is noisy.

Random Inputs is a defense where the values of the inputs (DVFS, balloon and idle activity levels) are changed randomly at runtime. Once a value for an input is chosen, it is kept

Table 6.5: Designs compared.

Design	Description
Baseline	High-performance insecure system without added noise
Noisy Baseline	Each run has new DVFS and idle activity levels
Random Inputs	Input values are set randomly at runtime
Maya Constant	Maya (Figure 6.1) but with a constant mask
Maya GS	Maya with a Gaussian Sinusoid mask (Proposed defense)

unchanged for a random duration, after which another value is selected. This makes the application's power profile significantly noisy.

Maya Constant uses Maya's formal controller but the target is kept constant. Finally, *Maya GS* is our proposal that uses the formal controller and Gaussian Sinusoid mask generator.

We evaluate the security of the designs in Table 6.5 against *Basic* and *Adaptive* attacks. In the Basic attack, attackers collect training data when the victims run with *Noisy Baseline*, and test the MLP classifier on signals obfuscated with the other defenses (*Random Inputs*, *Maya Constant* or *Maya GS*).

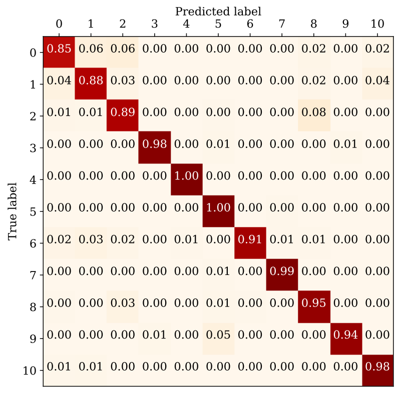
The Adaptive attack is where attackers adapt to each defense. They train their MLP with obfuscated traces collected when a defense is active (e.g., *Maya GS*) and use the MLP to recognize new obfuscated traces from the *same* defense.

6.6 RESULTS

We describe the effectiveness of the defenses, give insights using signal statistics, show the effectiveness of formal control, present the defense overheads, compare the effectiveness across machines, and consider attacks at higher frequency.

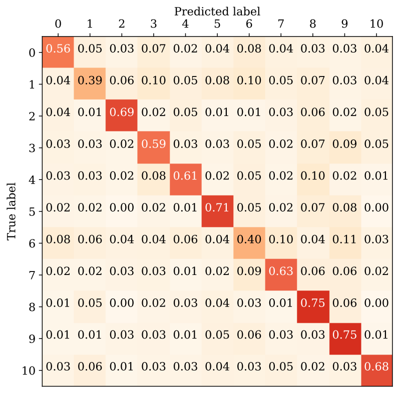
6.6.1 Effectiveness of the Defenses

Detecting the active application: The MLP classifier trained on *Noisy Baseline* signals has an average accuracy of 85% when predicting other *Noisy Baseline* signals. The accuracy drops to nearly 9% when this classifier is tested on signals obfuscated by any of the other defenses in Table 6.5 (*Random Inputs*, *Maya Constant* and *Maya GS*). Note that the random chance of correct classification is ≈9%, as there are 11 applications. An accuracy near or below this value indicates a classification failure. Hence, in the Basic attack mode, the MLP classifier is no better than chance prediction. This is because the obfuscated signals are not similar to the *Noisy Baseline* traces.



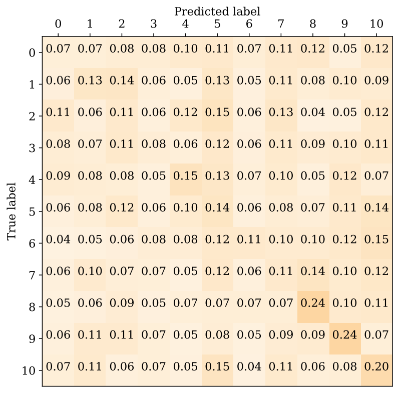
True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10
0	0.85	0.06	0.06	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.02
1	0.04	0.88	0.03	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.04
2	0.01	0.01	0.89	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00
3	0.00	0.00	0.00	0.98	0.00	0.01	0.00	0.00	0.00	0.01	0.00
4	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
6	0.02	0.03	0.02	0.00	0.01	0.00	0.91	0.01	0.01	0.00	0.00
7	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.99	0.00	0.00	0.00
8	0.00	0.00	0.03	0.00	0.00	0.01	0.00	0.00	0.95	0.00	0.00
9	0.00	0.00	0.00	0.01	0.00	0.05	0.00	0.00	0.00	0.94	0.00
10	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.98

(a) Random Inputs (Avg. accuracy 94%)



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10
0	0.56	0.05	0.03	0.07	0.02	0.04	0.08	0.04	0.03	0.03	0.04
1	0.04	0.39	0.06	0.10	0.05	0.08	0.10	0.05	0.07	0.03	0.04
2	0.04	0.01	0.69	0.02	0.05	0.01	0.01	0.03	0.06	0.02	0.05
3	0.03	0.03	0.02	0.59	0.03	0.03	0.05	0.02	0.07	0.09	0.05
4	0.03	0.03	0.02	0.08	0.61	0.02	0.05	0.02	0.10	0.02	0.01
5	0.02	0.02	0.00	0.02	0.01	0.71	0.05	0.02	0.07	0.08	0.00
6	0.08	0.06	0.04	0.04	0.06	0.04	0.40	0.10	0.04	0.11	0.03
7	0.02	0.02	0.03	0.03	0.01	0.02	0.09	0.63	0.06	0.06	0.02
8	0.01	0.05	0.00	0.02	0.03	0.04	0.03	0.01	0.75	0.06	0.00
9	0.01	0.01	0.03	0.03	0.01	0.05	0.06	0.03	0.03	0.75	0.01
10	0.03	0.06	0.01	0.03	0.03	0.04	0.03	0.05	0.02	0.03	0.68

(b) Maya Constant (Avg. accuracy 62%)



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9	10
0	0.07	0.07	0.08	0.08	0.10	0.11	0.07	0.11	0.12	0.05	0.12
1	0.06	0.13	0.14	0.06	0.05	0.13	0.05	0.11	0.08	0.10	0.09
2	0.11	0.06	0.11	0.06	0.12	0.15	0.06	0.13	0.04	0.05	0.12
3	0.08	0.07	0.11	0.08	0.06	0.12	0.06	0.11	0.09	0.10	0.11
4	0.09	0.08	0.08	0.05	0.15	0.13	0.07	0.10	0.05	0.12	0.07
5	0.06	0.08	0.12	0.06	0.10	0.14	0.06	0.08	0.07	0.11	0.14
6	0.04	0.05	0.06	0.08	0.08	0.12	0.11	0.10	0.10	0.12	0.15
7	0.06	0.10	0.07	0.07	0.05	0.12	0.06	0.11	0.14	0.10	0.12
8	0.05	0.06	0.09	0.05	0.07	0.07	0.07	0.07	0.24	0.10	0.11
9	0.06	0.11	0.11	0.07	0.05	0.08	0.05	0.09	0.09	0.24	0.07
10	0.07	0.11	0.06	0.07	0.05	0.15	0.04	0.11	0.06	0.08	0.20

(c) Maya GS (Avg. accuracy 14%)

Figure 6.5: Confusion matrices for detecting the active application from power signals using Advanced attacks.

Next, consider Adaptive attacks where attackers train their MLP on the traces obfuscated by a defense, and classify other traces obfuscated by the *same* defense. We show these results using confusion matrices (Figure 6.5a). The confusion matrix is a table where each row corresponds to the true labels of the applications (0 – 10) and each column has the fraction of the signals classified as the predicted labels. For example, the entry in the 0^{th} row and 1^{st} column gives the fraction of signals that had a true label of 0 but were classified as application 1. The diagonal entries give the correct predictions and averaging all the diagonal entries gives the overall average accuracy.

Figure 6.5 shows the confusion matrices for Adaptive attacks on the three defenses we test. The average classification accuracy is 94%, 62% and 14% for *Random Inputs*, *Maya Constant*, and *Maya GS*, respectively. *Random Inputs* fails at obfuscation because randomly changing the system inputs does not hide the inherent activity of the application. For example, changing DVFS has a different impact in compute and memory bound phases of the application. The MLP catches such differences.

Maya Constant manipulates the system inputs to maintain constant power instead of changing them randomly. It has better obfuscation than *Random Inputs* but is ultimately ineffective. As described in Section 6.3.3, ensuring constant power is not realistic, and information leaks at all application change-points.

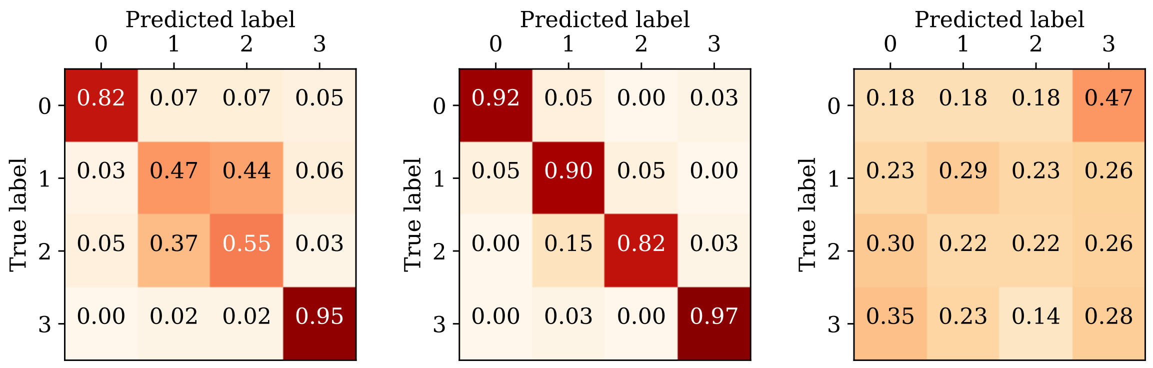
Finally, the advanced attack on *Maya GS* has only 14% average accuracy. This is close to the chance prediction accuracy of 9%. The only difference is because the MLP's classification is biased towards a few labels (e.g., labels 9, 10). This occurs sometimes when the MLP cannot find patterns to learn in the training data. We verify this by training another MLP directly on the Gaussian sinusoid masks, which have no correlation with the applications, and see an accuracy of 13%.

Thus, *Maya GS* achieves excellent obfuscation. The Gaussian sinusoid mask and the formal controller thoroughly overwrite and hide any original patterns in the application with false activity. Moreover, *Maya GS* produces a different trace in each run. Therefore, the MLP cannot find any common pattern.

Detecting video data: In the Basic attack, an MLP trained with *Noisy Baseline* signals has a video detection accuracy of 84%, 25%, 34% and 18% when tested on *Noisy Baseline*, *Random Inputs*, *Maya Constant*, and *Maya GS* signals, respectively. Here, the accuracy of random chance classification is 25%, as we have four videos. Therefore, we can construe that the basic attack does not succeed with any of the 3 defenses.

For the Adaptive attack, Figure 6.6 shows the confusion matrices. The average accuracy of the MLP attack is 72%, 90% and 24% respectively, for *Random Inputs*, *Maya Constant* and *Maya GS*. As with the previous attack, *Random Inputs* and *Maya Constant* fail to obfuscate activity, and only *Maya GS* can hide activity.

Although both *Random Inputs* and *Maya Constant* are ineffective, the MLP has a lower accuracy against the former. It cannot clearly distinguish traces of video 1 (tractor) from video 2 (wind) with *Random Inputs*. Originally, these videos have similar traces except for a few peaks. Therefore, the noise caused by *Random Inputs* results in misclassification. In contrast, *Maya Constant*, makes the peaks more prominent because the signal is otherwise constant. Thus, the MLP has a higher success rate with *Maya Constant*.



(a) Random Inputs (Avg. accuracy 72%)

(b) Maya Constant (Avg. accuracy 90%)

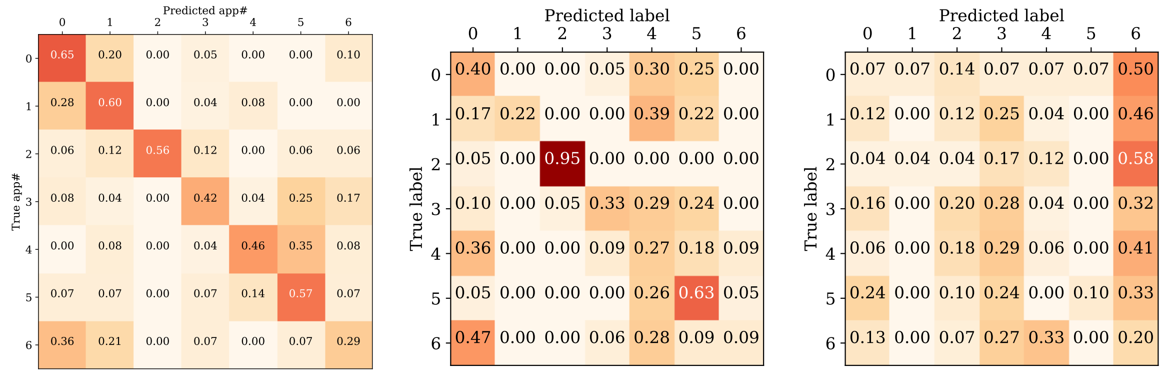
(c) Maya GS (Avg. accuracy 24%)

Figure 6.6: Confusion matrices for Adaptive video detection attacks.

Detecting browser data: Recall that we run this attack using FFT values from AC outlet power traces. Here, the accuracy of random chance classification is 15%, as we have seven webpages. In the Basic attack, an MLP trained with *Noisy Baseline* signals has an accuracy of 88%, 14%, 21% and 14% when tested on *Noisy Baseline*, *Random Inputs*, *Maya Constant*, and *Maya GS* signals, respectively. This indicates that the three defenses can withstand the basic attack.

For the Adaptive attack, the average accuracy of the MLP models is 51%, 40% and 10% respectively, for *Random Inputs*, *Maya Constant* and *Maya GS*. Websites like Google (0), Youtube (2), Chase banking (3) and Amazon (5) are recognized even with *Maya Constant*, thus endangering privacy. In contrast, *Maya GS* achieves perfect obfuscation.

Overall, the several attacks establish that *Maya GS* is successful in obfuscating power side channels. Other defenses that maintain constant power or randomize the system settings fail to achieve this level of security. Maya's strength is clear when it resisted Adaptive attacks where the attacker could train with thousands of signals generated from Maya. This comes from the effective mask (Gaussian Sinusoid) and the formal controller that keeps power close to the mask.



(a) Random Inputs (Avg. accuracy 51%)

(b) Maya Constant (Avg. accuracy 40%)

(c) Maya GS (Avg. accuracy 10%)

Figure 6.7: Confusion matrices for Adaptive webpage detection.

6.6.2 Signal Statistics and Analysis

For more insights, we analyze the signals produced by each defense of Table 6.5 using signal summary statistics and changepoint analysis.

Signal summary statistics: We perform this analysis for each defense to study its behavior across runs. For each application, we collect all the traces produced by the defense across runs and average them. Then, we examine the distribution of values in this averaged signal. An effective defense would have similar distribution across applications.

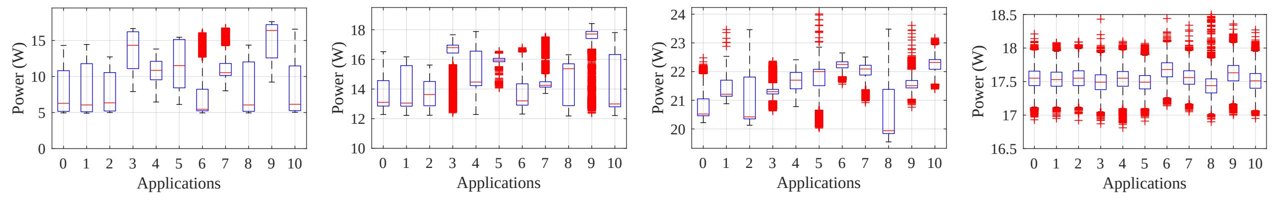
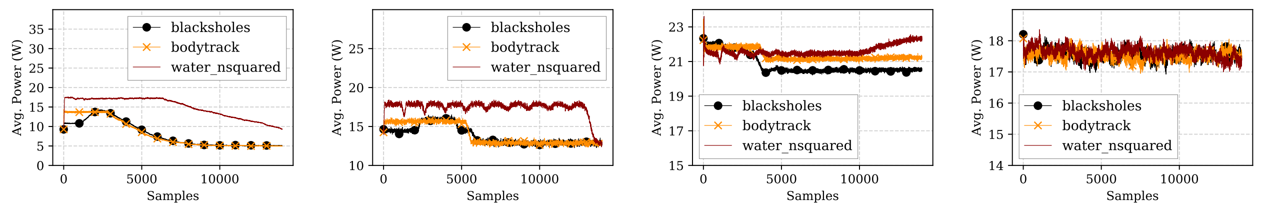

(a) Noisy Baseline (b) Random Inputs (c) Maya Constant (d) Maya GS

Figure 6.8: Summary statistics of the average of 1,000 signals. The Y axis of each chart is drawn to a different scale.



(a) Noisy Baseline (b) Random Inputs (c) Maya Constant (d) Maya GS

Figure 6.9: Average of 1,000 traces for blackscholes, bodytrack and water_nsquared (labels 0, 1 and 9). The Y axis of each chart is drawn to a different scale.

Figure 6.8 shows the box plots of values in the averaged traces for *Noisy Baseline*, *Random Inputs*, *Maya Constant* and *Maya GS*. The averages are obtained from 1,000 raw traces of each PARSEC application. Each chart labels the applications on the horizontal axis from 0 to 10. Each box shows the 25th and 75th percentile values for the application and the line inside the box is the median value. The whiskers of the box extend up to the maxima and minima. The '+' markers represent values detected statistically as outliers in the distribution. For legibility, the Y axis on each chart is drawn to a different scale.

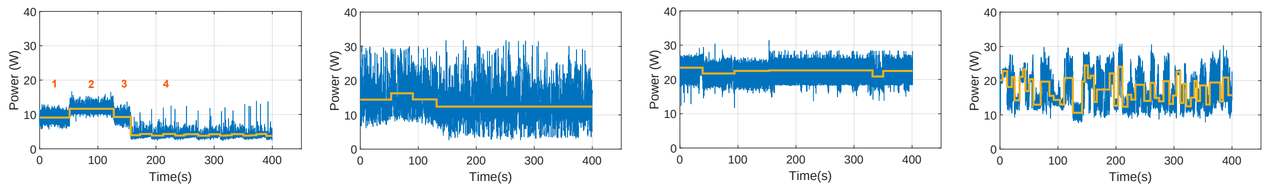
With the *Noisy Baseline* (Figure 6.8a), the value distribution is distinct for each application, and acts like a fingerprint. In *Random Inputs* (Figure 6.8b), the boxes shrink in size, but the relative difference remains the same. This is because changing inputs randomly does not hide the inherent changes induced by the application. With *Maya Constant* (Figure 6.8c), the boxes shrink further and the median values of applications become closer to each other. However, the distribution is sufficiently different to identify each application.

Finally, with *Maya GS* (Figure 6.8d) the distributions are *near-identical*. The median values are nearly the same because *Maya GS* produces a different trace in each run that is uncorrelated with other runs. Moreover, each run uses the whole range of allowed values. Therefore, averaging traces cancels out the patterns — simply leaving a constant-like value with small variance. Hence, the median, mean, variance, and the distribution of the samples are close, indicating a high degree of obfuscation. Note that the resolution of this data is 0.01W.

As an example of the differences, Figure 6.9 shows the averaged signals of three applications for all the designs. Again, the Y axis for each chart is drawn to a different scale. Following the earlier discussion, *Noisy Baseline* (Figure 6.9a), *Random Inputs* (Figure 6.9b) and *Maya Constant* (Figure 6.9c) show distinct traces for each application. Indeed, with *Random Inputs*, the different iterations in the water_nsquared application are clear because the random noise is averaged out. For this application, *Maya Constant* (Figure 6.9c) also leaves some change-points in the initial iterations

It is only in *Maya GS* (Figure 6.9d) that the average traces are indistinguishable from each other. This results in the highest degree of obfuscation.

Change Point Detection: This is a signal-processing technique used to identify times when the properties of a signal change. The properties can be the signal mean, variance, edges, or fourier coefficients. We use a standard change point detection algorithm [187] to identify the phases found in the re-shaped signals. We present the highlights of this analysis using the *blackscholes* application.



(a) Noisy Baseline (b) Random Inputs (c) Maya Constant (d) Maya GS

Figure 6.10: Change point detection in *blackscholes* using traces over time. Figure 11(a) shows all four phases being detected.

In the Noisy Baseline (Figure 6.10a), four phases of the application can be clearly distinguished: (1) sequential, (2) parallel, (3) sequential and (4) idleness after the application completes. The difference between the different phases is not too large and there is some noise because of interference with idle and balloon activity. Nonetheless, the algorithm detects all the four major phases.

With *Random Inputs*, (Figure 6.10b), the profile is significantly noisy. Since the noise is random, the inherent application activity is uniformly perturbed and hence, any phases in the application are still visible. The change-point detection algorithm could identify all the phases.

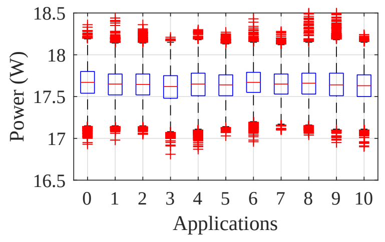
In *Maya Constant* (Figure 6.10c), the power profile is mostly around 25 W because the mask is held constant at that value. However, the algorithm can still recover all the phases.

Recall that a constant target cannot prevent activity from leaking at the phase transitions. There are sharp peaks at all phase change points. The FFT of the signal (not shown) also preserves such abrupt changes. All such information is used by the changepoint detection methods to identify phase behavior.

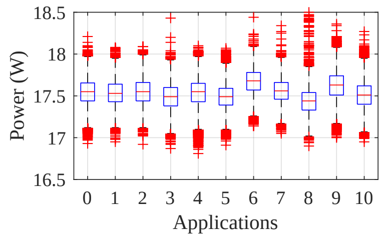
Figure 6.10d shows the signal with *Maya GS*. Change point analysis detects many phases, *but these are all artificial*. The signal and its FFT (not shown) are totally different from the original signal. In fact, it is also *not possible to infer when the application completed*. The application actually completed around 105 s, but the power signal has no distinguishing change at that time.

6.6.3 Effectiveness of Formal Control

Figure 6.11 shows the value distribution in the averaged signals as given by: (i) the Gaussian Sinusoid mask generator (Figure 6.11a) and (ii) the actual power measured from the computer (Figure 6.11b). It can be seen that the formal controller is effective at making the measured power appear close to the target mask. Indeed, this accurate tracking is what helps Maya in effectively re-shaping the system's power and hiding application activity.



(a) Ideal value distribution given by the GS mask.



(b) Measured value distribution from the computer.

Figure 6.11: Value distribution in the average of targets and measured powers, showing high-fidelity power-shaping.

6.6.4 Overheads and Power/Performance

Finally, we examine the implementation overheads of Maya and its impact on application power/performace.

Overheads of Maya: The controller reads one output, sets three inputs and has a state vector $x(T)$ that is 11-element long (Equations 2.4 and 2.5). Hence, the controller needs less than 1 KB of storage. At each invocation, it performs ≈200 fixed-point operations to make a decision. This completes within 1 μs.

The mask generator requires (pseudo) random numbers to compute the mask and change the properties of the gaussian distribution and sinusoid signal (Equation 6.2). In our implementation, we use a software library that takes less than 10 μs to generate the random numbers. A hardware implementation can use off-the-shelf hardware instructions and IP modules to generate them in sub-μs [188, 189].

Maya needs few resources to operate, making it attractive for firmware, software or even hardware implementations. The primary bottlenecks in our implementation were the sensing and actuation latencies, which are in the ms time scale.

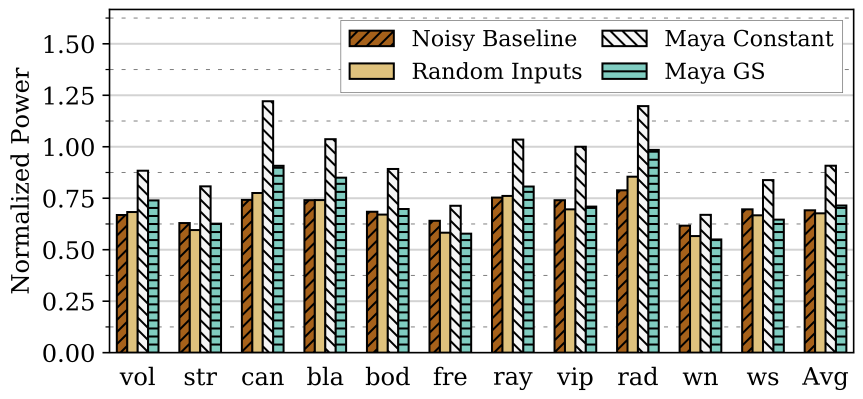
Application-Level Impact: We run the PARSEC and SPLASH2x applications on our three systems with all the designs and on *Baseline*. *Baseline* runs all applications at the highest available frequency without inserting idle threads or the balloon application. We measure the impact of the other designs relative to *Baseline*.

Figure 6.12 shows the power and execution time of all the designs, normalized to that of the high-performance Baseline. From Figure 6.12a, the average power consumed by the applications with *Noisy Baseline*, *Random Inputs*, *Maya Constant*, and *Maya GS* is 30%, 31%, 11% and 31% lower than *Baseline*, respectively. The power is lower because all defenses hinder the execution with idle threads and low DVFS values.

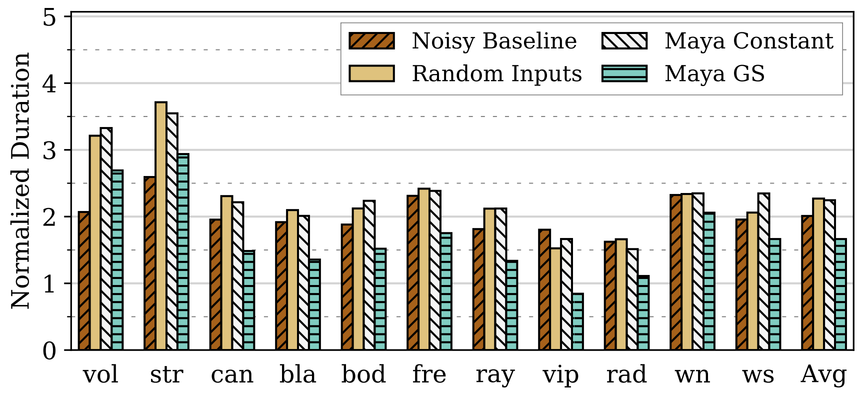
Figure 6.12b shows the normalized execution times. On average, the execution time increases of *Noisy Baseline*, *Random Inputs*, *Maya Constant*, and Maya GS are 95%, 126%, 124% and 61%, respectively, over the Baseline. *Maya Constant* uses a single power target that is lower than the maximum power at which Baseline runs. Therefore, its execution time is the worst. *Noisy Baseline* and *Random Inputs* are also slow because of continuous interference between the application and the idle and balloon threads. *Maya GS* has relatively lower execution time than the others because its execution spans a wider range of power levels thanks to its many choices. As a result, it allows applications to run steadily at higher power occasionally. Note that only *Maya GS* provides security among all the designs.

It can be shown that the power and performance overheads of the designs in *Sys2* and *Sys3* are similar to those in *Sys1*. This shows that Maya is robust across different machines.

We believe that the loss of performance with Maya GS is justified by the highest level of security it provides. To reduce the slowdown further, Maya can be activated only on demand like the Linux power governors [149]. Authenticated users or secure applications can activate Maya before commencing a sensitive task, and stop it once it completes.



(a) Power



(b) Execution Time

Figure 6.12: Overheads of our environments on *Sys1* relative to a high-performance insecure Baseline.

6.6.5 Effectiveness Across Machines

Using multiple machine learning-based attacks, signal analysis, and different forms of signal collection (counters and AC outlet measurements), we showed how *Maya GS* can obfuscate the power signals from different computers. Maya's security comes from an effective mask, and from the formal controller that can shape the computer's power into the given mask. We want to emphasize that we implemented Maya on three different machines *without modifying* the controller or the mask generator, demonstrating our proposal's robustness, security, and ease of deployment.

6.6.6 Attacks at Higher Frequency

We repeat the application detection adaptive attack on Sys1, but reduce the attacker's power sampling interval from 20 ms to 10 ms, 5 ms, and 2 ms. In all cases, the Maya defense samples power at 20 ms. Figure 6.13 shows the average application detection accuracy across the sample intervals. We see that the average detection accuracy does not change much, and remains low like in Figure 6.5c, even with the faster sampling by the attacker.

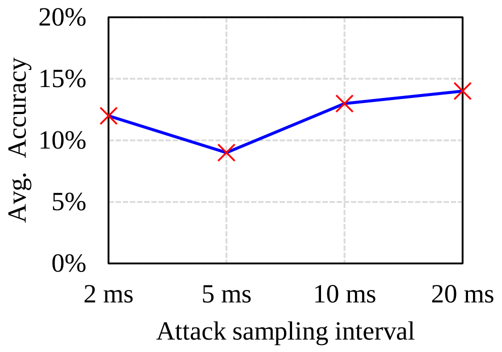


Figure 6.13: Accuracy of machine learning classifiers in the application detection attack on *Maya GS* for different attacker sampling intervals.

Faster sampling does not improve detection accuracy in this attack because the distinguishing patterns of the applications (e.g., phases) occur at longer timescales than the sampling intervals. Furthermore, even though Maya's controller only changes inputs every 20 ms, the idle and balloon threads are always running, and add noise to the application. Finally, faster sampling inherently has more noise, affecting detection.

6.7 THWARTING A NEW REMOTE POWER ATTACK

Maya is practical and mitigates attacks that require no physical access. A recent work [42] implemented a new power covert-channel attack that involved four victim computers connected to AC electric power outlets in a building. An attacker computer situated 90 feet away from the victims taps to an AC electric power outlet connected to the power network of the building. The attacker listens to high-frequency voltage ripples with an oscilloscope every 2 μs, and decodes one bit of information every 33 ms, exfiltrating data from each victim. That work then implements Maya in software, with controller actions taken every 40 ms. It shows that Maya effectively thwarts that covert channel. When Maya is used, the attacker cannot even detect the pilot sequence necessary to extract the transmitted bits. It also shows that Maya is the only software defense that provides security. Other approaches like adding random noise fail to prevent information leakage. Overall, this shows Maya's effectiveness and ease of deployment.

6.8 CHAPTER SUMMARY

This chapter presented a simple and effective solution against power side channels. The scheme, called *Maya*, uses for the first time, formal control to distort, in an application-transparent way, the power consumed by a computer—so that the attacker cannot obtain information about the applications running. The controller keeps the power close to a desired time-varying target, even as runtime conditions change unpredictably. The power signal is made to appear to carry activity information which, in reality, is unrelated to the program. We implemented Maya on three machines using OS threads, and showed that it is very effective at obfuscating application activity. Maya has already thwarted a newly-developed remote power attack.

CHAPTER 7: RELATED WORK

7.1 DYNAMIC CONTROL FOR PROCESSORS

The argument for systematic coordinated control of multiple power/performance management policies has been advocated in prior research [59,61,78,155,190,191]. In addition, hardware support for adaptive power/performance management is increasingly being used in modern processors [1,2,3,4]. The Intel Skylake processor [1] uses a SISO PID controller within its energy management architecture. The IBM POWER 8 processor [2] has reconfiguration registers within the pipeline, and supports fast and fine-grained per-core DVFS. The Intel IvyBridge processor [3] can resize its last-level cache by power gating its ways.

We discuss some past research in this area, following the classification in Table 2.2.

Rule-Based Heuristics: There are some works that use rule-based heuristics to adapt configurable architectures. Some of these works adapt one resource (e.g., [11,12]). Other works adapt multiple resources in a coordinated manner (e.g., [10,78,116,117]). In particular, Vega et al. [78] demonstrate the conflicting nature of decoupled management of multiple policies. Zhang and Hoffmann [63] propose a framework for maximizing performance under a power cap using a heuristic algorithm.

Model-Based Heuristics: There are some works that use models to drive the adaptation heuristics. For example, they use models for joint memory and processor DVFS [55], cache size [192], multicore thermals [115], or on-chip storage [193]. Our MIMO methodology also uses an offline model of the processor dynamics.

Control Theoretic Techniques: In addition to the SISO schemes discussed in Section 2.1([1, 13, 102, 104]), there are works that use multiple SISO controllers managed together with heuristics [194, 195, 196]. The use of such decoupled controllers has to be planned carefully before deploying them. If there are cross dependencies between the input of one and the output of another, then the controllers will conflict with each other.

The approach that combines multiple SISO models to generate a larger MISO controller [99, 101,103] still requires heuristics to encode some decisions. This may make the controllers suboptimal and error prone [113,133]. The MIMO methodology can natively model the required interactions, eliminating unanticipated behavior and finding better solutions.

There have been some designs that use hierarchical control loops to coordinate multiple conflicting policies or objectives. Specifically, Raghavendra et al. [61] propose such a scheme to control power in datacenters, and Fu et al. [100] propose another such design to control utilization in processors. In these proposals, a formal controller is used only in the innermost loop, and each higher level works on a different abstraction and specifies the targets for the lower levels. This approach is specific for hierarchically organized systems. For other cases, it might reduce the freedom of the slower timescale controller, hence resulting in a sub-optimal operation.

Machine Learning techniques: Machine learning techniques have been used to tune architectural parameters (e.g., [59, 114, 197]). They have two main differences with control theory techniques. The first difference is runtime feedback.

Machine learning techniques learn by recording what input values are best for different observed output conditions. However, if they find different output conditions at runtime than those they were trained on, they provide a lower-quality solution unless they go through an expensive re-training phase. Control theory techniques, instead, when they find runtime output conditions to be different than those modeled, they use their intrinsic feedback loop to adapt to the new conditions with low overhead. The second difference is design guarantees. Unlike machine learning techniques, control theory techniques can provide convergence, stability, and optimality guarantees.

Optimization techniques: There are some works that adapt based on optimization formulations. For example, they optimize inputs to minimize power consumption [198, 199], performance subject to power constraints [112], or $E \times D^2$ [120]. STEAM [113] models the power and performance of cores as a function of P- and T-states, IPC, temperature, and memory accesses. It then uses a convex optimization solver to maximize the ratio of performance over power.

7.2 MULTILAYER RESOURCE CONTROL

Expanding on Chapter 1, we discuss additional aspects of prior work on multilayer control. We consider controller organization (Table 2.1) and methodology.

Organization: Decoupled controllers are simple to design and modular, while monolithic controllers can achieve better results due to a global view of the system. However, decoupled techniques can exhibit destructive interference between controllers, even at a single layer [61, 62, 69, 78, 92, 200]. In turn, monolithic techniques have design complexity and are less maintainable, scalable, or portable [61, 62, 63, 65, 66, 69].

Some designs use decoupled controllers that are coordinated implicitly by the use of controller ranking [63, 66, 67, 79, 80]. In these designs, each controller runs at a slower timescale than its immediately higher-ranked one. The highest-ranked controller adjusts one resource first, to attain the most important goal. Each subsequent lower-ranked controller modifies a different resource, to meet a different goal. Bhattacharya et al. [152] and Maggio et al. [111] show that such designs may have responsiveness and stability issues.

Graybox [71] creates middleware that exposes useful OS information to the application to coordinate software controllers. Other authors argue for collaborative control [69]. However, as we see next, most of the existing collaborative controllers (e.g., [68,69,81]) have limitations.

Methodology: Multilayer controllers can be based on heuristics, control theory, machine learning, or optimization theory. Many works rely on heuristics to modify parameters and coordinate controllers [63, 68, 69, 81]. However, researchers have shown how heuristics can fail [67]. Other designs use a combination of heuristics and control theory [61, 66], heuristics and optimization [190], or just optimization [201]. The xTune framework [73] uses Monte Carlo simulations at runtime to pick the statistically-best action from a list of designer-specified actions. Some designs use a combination of machine learning and PID control [67, 202]. Muthukaruppan et al. [65] use price theory for big.LITTLE systems.

7.3 DECENTRALIZED CONTROL OF HETEROGENEOUS COMPUTERS

General Control

There is a large body of work on controlling homogeneous or single-ISA heterogeneous processors [54, 55, 58, 61, 78, 92, 94, 102, 111, 166, 202, 203, 204, 205, 206, 207, 208]. Only a few consider heterogeneous processors with CPUs and GPUs [4, 52, 53, 209, 210]. Still, they use non-modular controllers that do not match the modular heterogeneous environments we target.

Resource control in production computers is predominantly heuristic [1, 2, 3, 4, 9, 50, 52, 53, 78, 81, 153, 154, 166, 211, 212, 213, 214]. As we indicated, heuristic control has limitations.

Research works propose many optimizing controllers [56, 57, 58, 61, 92, 94, 101, 111, 113, 116, 207]. Most do not consider interference from dedicated safety controllers. Therefore, they do not simultaneously guarantee optimality and safety. Some works do consider temperature as a soft constraint [61, 94, 113] while some probabilistically characterize mechanisms like circuit breaker tripping for their search [58]. In real designs, there are many safety engines that interrupt and override optimizing engines unpredictably. We guarantee optimality and safety simultaneously.

Enhancement Approaches

Heuristic Control: Many designs rely heavily on heuristics for resource control [1, 2, 3, 4, 52, 53, 54, 55, 78, 116]. While easy to implement for simple systems, designing, tuning and verifying heuristics becomes dramatically expensive as systems and resource management goals become complex. This can result in unintended inefficiencies [78, 79, 92, 101, 111, 155].

Formal Control. PID controllers are popularly used in many works due to their simplicity [1, 9, 61, 79, 80, 194, 196, 202, 204, 215, 216, 217]. However, PID controllers are SISO designs, inadequate to meet the multiple objectives in computers [92, 94, 111]. LQG [60, 92] and MPC [111] controllers can handle MIMO systems, but are relatively less effective in uncertain and multi-controller environments [91, 94]. Yukta [94] (Chapter 4) proposes the use of robust controllers for computer systems. These controllers operate well in environments that are not fully modeled. Yukta introduced the use of a robust controller for each system layer (e.g., the hardware and OS layers).

Other Systematic Methods. Some works formulate Energy$\times$Delay^n minimization as a convex optimization problem solved with linear programming solvers [113, 201, 203]. Solver-based approaches require more time to generate a decision than robust controllers. Some use market-theory [56, 57] or game-theory [58] to manage resources in specific contexts. Finally, some researchers use machine learning techniques for resource management [59, 197, 218, 219]. Mishra et al. [202] use machine learning to tune a PID controller and a solver that manage a big.LITTLE processor.

Control System Architectures

Centralized. Most works use centralized frameworks (e.g., [52, 53, 92, 101, 111, 113, 201]). Some use two-step proxy designs where a proxy module in each component requests resources and a centralized manager performs the allocation [56, 57, 58, 59, 116, 207]. Centralized designs are not modular, do not scale to multi-chip computers, and do not fit IP-based system designs. As systems grow large, the controller's response time degrades quickly because it runs a bulky algorithm and becomes a point of contention.

Cascaded. Raghavendra et al. [61] propose a multilevel cascaded system to manage power in a datacenter. Rahmani et al. [60] propose a similar 2-level design for a big.LITTLE processor. Here, each component has two LQG controllers. A supervisor chooses one of them to control the system and provides targets for all local outputs. We showed that cascaded is non-modular and has a poor response time.

Other Architectures. Muthukaruppan et al. [66] use a combination of cascaded and decoupled PID controllers for a big.LITTLE processor. Some designs order decoupled controllers by priority for limited coordination [79, 80].

Attacks

Many attacks identify sensitive information from physical signals using machine learning (ML) based pattern recognition [180]. Yan et al. recover the running application's identity and the number of keystrokes typed [24]. They measured power through unprivileged counters. Lifshits et al. identified browser, camera and location activity, and the characters typed [25], using measurements from a malicious battery. Chen et al. recovered Android app usage information from power signals [37]. Yang et al. showed that compromised public USB charging booths can recover the user's browser activity [26].

Clark et al. [167] and Hlavacs et al. [175] identified the webpages and virtual machine applications, respectively, using the server's electrical outlet power. Islam et al. showed the vulnerability of multi-tenant datacenters to voltage, thermal and acoustic (from cooling devices) side channels [31, 168, 181].

Michalevsky et al. showed that malicious smartphone applications could track the user's location without GPS, by only using unprivileged OS-level power counters [27]. Conti et al. [220] showed that a laptop's power signals differ by the user, and they could identify the laptop's original user among other occasional users.

Shao et al. developed a covert channel that uses the power delivery network to which a computer is connected [42]. Masti et al. developed a covert channel in multicores based on temperature coupling between the cores [30]. Khatamifard et al. build a power covert channel based on the hierarchical power management policies in multicores.

There are several attacks that use trojan chips, circuits and FPGAs to measure physical signals of co-located chips [40, 43, 44, 45, 178]. Cloud systems are offering FPGA platforms, and are vulnerable to power analysis attacks [44, 178, 179].

Attackers have also used software analysis or modeling to estimate power when direct measurement is difficult [24, 34, 35, 36, 38].

Kocher et al. [28] give a detailed overview of exploiting power signals with Simple Power Analysis (SPA) and Differential Power Analysis (DPA) to recover encryption keys. However, they assume physical access. There are other attacks which could circumvent this restriction [178].

There are attacks that capture EM emissions using antennas, and recover sensitive data using similar techniques as with power [41, 172, 176, 177, 184, 221].

Defenses

Trusted execution environments like Intel SGX [222] or ARM Trustzone [223] do not contain physical signals [173, 224]. Therefore, several countermeasures against power side channels have been proposed [28, 29, 84, 85, 86, 87, 88, 89, 225, 226, 227, 228]. Their goal is primarily to protect encryption circuits, and require new hardware.

Known defenses usually operate by either suppressing power signal changes that arise due to changing activity [84, 85, 225, 226], or adding noise to drown activity [28, 29], or both [87]. A common approach to adding noise is to randomize DVFS levels using special hardware [86, 88, 89].

Baddam and Zwolinski showed that randomizing DVFS is not a viable defense [229]. Yang et al. suggest randomly scheduling the encryption task among the cores in a multicore, apart from randomly setting the clock frequency and phase [227]. Real et al. showed that adding noise or empty activity can be filtered out, and is ineffective [90]. The defense we propose, Maya, does not simply add noise to the power signal. Instead, it re-shapes it to a suitable mask.

A different approach is to temporarily cut-off a circuit from the outside and run it with a small amount of energy stored inside itself [228]. Alternatively, *each* application can be modified so that its physical outputs do not carry sensitive information [41]. However, it is not practical to modify all applications to have indistinguishable power traces.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

To address the urgent need for building resource efficient and secure computers, this dissertation made a series of contributions. The key idea has been to manage computers with principled methods from formal control. First, it applied MIMO techniques to the development of controllers for processors. Then, it presented *Yukta*, which is a new approach to build coordinated multilayer formal controllers for computer systems. Next, it introduced a new control framework *Tangram*, which is fast, globally coordinated, and modular, to manage heterogeneous systems. Finally, it presented a simple and effective solution against power side-channels called *Maya*. Maya uses, for the first time, formal control to distort, in an application-transparent way, the power consumed by a computer—so that the attacker cannot obtain information about the applications running.

The proposed solutions in the dissertation have been prototyped on several platforms including one built along with AMD engineers. These designs significantly outperformed the state of the art.

The research in this dissertation brought formal control closer to computer architecture and has been well-received in both domains. It has the first application of full-fledged MIMO control for processors, the first use of robust control in computer systems, and the first application of formal control for side-channel defense. We expect this work to be key in architecting future intelligent systems that are efficient, secure, and reliable.

Building on this dissertation, there are several exciting research directions to achieve extreme efficiency and security at scale. A promising approach is to augment the control theory-based techniques proposed by this work with machine learning (ML) methods. ML methods can use data to develop control policies, although the guarantees they provide are about the average case and are stochastic in nature. In contrast, control theory can provide worst case guarantees about stability, convergence and optimality. We can have ML agents dynamically reconfigure parts of the computer proactively, while robust control corrects the settings after sensing the actual conditions.

In another direction, we can use reinforcement learning (RL) to fully customize the system to application activity. However, online RL training is too slow. Hence, we could use only robust control initially and, in parallel, train the RL agent by observing the controller. Once the RL agent is sufficiently confident, it fully takes over the system.

On the security front, it is well known that computers have many side-channels. Extending the Maya approach with MIMO control, we can envision the development of obfuscation controllers (obfuscators) that obfuscate multiple side-channels at the same time. Different parts of the chip, like the prefetcher, cache controller and branch predictor can include these obfuscators to prevent many side-channels. Such obfuscators could be turned-on/off on demand, and even be reconfigured when different energy/performance/security trade-offs are desired. Moreover, these obfuscators could be realized by intelligently re-programming existing controllers that already exist in computers (e.g., cache or memory controllers), to reduce the hardware overhead.

An important related question to answer is: how to achieve both resource efficiency and security? Fundamentally, these are opposing goals. Efficiency requires customizing resource utilization to application activity. However, this makes system outputs like power to be highly correlated with the application activity, which leaks information. A possible approach is to use information analysis and properties of formal control to identify the optimal strategy for combining efficiency and security.

APPENDIX A: OPTIMIZER AND PLANNER ALGORITHMS

A.1 OPTIMIZER ALGORITHM IN YUKTA

The optimizer searches for the output targets to minimize a metric in its layer. Intuitively, the optimizer searches along two directions: a high throughput region (*Up*) and a low throughput region (*Down*). To move up in the high throughput region, the algorithm increases the target of Throughput and the target of the output, which is significantly below its maximum limit. Alternatively, to move down in the low throughput region, the algorithm decreases the target of Throughput and the target of the output, which is close to its limit.

The algorithm that the hardware optimizer runs to maximize $\frac{Throughput^2}{Power}$ (equivalent to minimizing EDP) under constraints (Section 4.5) is listed in Algorithm A.1. It is based on the algorithm in Section 3.2, and it is modified to support search constraints and multiple outputs. The hardware optimizer reads the outputs in its layer, the limits (Section 4.6), convergence bound ϵ, and restart probability δ. The optimizer converges if the relative improvement in the metric being optimized is below ϵ. The restart probability δ determines the probability with which the optimizer's search can restart, even after convergence is achieved. The optimizer is initialized to search in the *Up* direction.

When invoked, the optimizer first computes the *margins* of all outputs [defined as the difference between the maximum limits and the actual values of the outputs (line 1)] and the *errors* [defined as the difference between the targets and the actual outputs (line 2)]. It then identifies $Output_{agg}$ (the output other than Throughput that has the smallest margin), $Output_{laz}$ (the output other than Throughput that has the largest margin), and $Output_{lag}$ (the output with the largest error).

A negative margin of $Output_{agg}$ means that an output exceeded its limit (line 3). The optimizer must then reduce the target for this output (line 4). Additionally, the optimizer reduces the target of $Output_{lag}$ because its tracking error is the largest (its target is too high). When an output's target is too high, the controller may cause the remaining outputs to go over their targets, so that the lagging output is brought closer to the target. Since this can result in some of the outputs going over their limits, the target for $Output_{lag}$ must be reduced. The reduction in targets is performed by the *decrease()* function in lines 3 and 4.

When no output is above its limit, the optimizer computes the value of the metric that was achieved with the previous choice of targets (line 7). It then calculates $\Delta metric$, which is the relative improvement of the metric's value over the previously achieved value (line 8). If the relative improvement is smaller than the convergence bounds ϵ, then the targets are

Algorithm A.1: Algorithm for the hardware optimizer in Yukta.

```
Input: outputs, targets, limits (Section 4.6), convergence bound $\epsilon$, restart probability $\delta$
Output: New targets for hardware outputs
Initialize : direction $\leftarrow$ Up, prev_metric $\leftarrow$ 0, stop_search $\leftarrow$ False
1  margins $\leftarrow$ limits $-$ outputs
2  errors $\leftarrow$ targets $-$ outputs
   // $Output_{agg}$ is the output other than Throughput with the smallest margin
   // $Output_{laz}$ is the output other than Throughput with the largest margin
   // $Output_{lag}$ is the output with the largest error
3  if $margin[Output_{agg}] < 0$ then
4      target[$Output_{agg}$] $\leftarrow$ $decrease$(target[$Output_{agg}$])
5      target[$Output_{lag}$] $\leftarrow$ $decrease$(target[$Output_{lag}$])
6  else
7      metric $\leftarrow$ $\frac{Throughput^2}{Power}$
8      $\Delta metric \leftarrow abs(\frac{metric - prev\_metric}{prev\_metric})$
9      if $\Delta metric > \epsilon$ or $rand() < \delta$ then
10         if $direction = Up$ then
11             if $metric > prev\_metric$ then
12                 target[Throughput] $\leftarrow$ $large\_increase$(target[Throughput])
13                 target[$Output_{laz}$] $\leftarrow$ $increase$(target[$Output_{laz}$])
14             else
15                 dir $\leftarrow$ Down
16                 target[Throughput] $\leftarrow$ $decrease$(target[Throughput])
17                 target[$Output_{agg}$] $\leftarrow$ $large\_decrease$(target[$Output_{agg}$])
18             end
19         else
20             if $metric > prev\_metric$ then
21                 target[Throughput] $\leftarrow$ $decrease$(target[Throughput])
22                 target[$Output_{agg}$] $\leftarrow$ $large\_decrease$(target[$Output_{agg}$])
23             else
24                 dir $\leftarrow$ Up
25                 target[Throughput] $\leftarrow$ $large\_increase$(target[Throughput])
26                 target[$Output_{laz}$] $\leftarrow$ $increase$(target[$Output_{laz}$])
27             end
28         end
29     end
30     prev_metric $\leftarrow$ metric
31 end
```

not modified. New targets must be generated when $\Delta metric$ is larger than ϵ, or with a small probability δ, even when $\Delta metric$ is below ϵ (line 9). The $rand()$ function in line 8 returns a number drawn randomly from a uniform distribution between 0 and 1. If the search direction is *Up* and the metric's value is increasing, the optimizer continues to search in the *Up* region. It increases the target for Throughput and $Output_{laz}$ that has the largest margin from the limit (lines 12 – 13). The target for Throughput is increased by a larger amount than the increase in the target for $Output_{laz}$. It is expected that the controller will be able to increase throughput much more than the increase in power or temperature. Increasing the target for $Output_{laz}$ gives the controller some freedom to increase Throughput. Otherwise, it may be possible that Throughput cannot be increased without an increase in power or temperature.

When the metric's value is not improving in the *Up* region, the optimizer reverses its direction to *Down* (line 15). In this direction, it decreases the target for Throughput and $Output_{agg}$ that has the smallest margin from the limit (lines 16 – 17). The target for Throughput is decreased by an amount smaller than the decrease in the target for $Output_{agg}$. The expectation of this move is that the controller would be able to reduce power or temperature much more than the reduction in throughput.

Similar decisions occur when the optimizer's search direction is *Down* (lines 20–26). The optimizer continues to proceed in the *Down* direction until the metric improves. If not, it reverses to the *Up* direction.

On the prototype computer, the *increase()* and *decrease()* functions increase and decrease the targets by 15% and 10%, respectively. The *large_increase()* and *large_decrease()* functions perform the respective changes by 20% and 15%. When decreasing, the targets are not reduced below zero; when increasing, they are capped at the maximum values the outputs can withstand. The convergence bound ϵ is 0.05 and restart probability δ is 0.05. The search does not cycle through the same points.

The optimizer's algorithm is simple but effective in practice. The algorithm follows the intuition that the best value of the metric $\frac{Throughput^2}{Power}$ occurs either in the high throughput region or the low throughput region. Therefore, instead of searching for all output targets simultaneously, each decision of the algorithm changes the throughput target and the target of another output necessary to improve throughput. Additionally, the algorithm did not have to explicitly account for changing system conditions because it relies on the SSV controllers to robustly keep outputs near the targets. Finally, implementing the algorithm requires only small computation and a few comparisons (which is one of the design requirements).

The algorithm for the OS optimizer differs only slightly and is not shown. The OS optimizer also has two search directions: Big-side (where the big cores contribute more to performance) and Little-side (where the little cores contribute more to performance). The algorithm finds the best targets for the OS outputs in this space.

A.2 NELDER-MEAD SEARCH USED BY THE PLANNERS IN TANGRAM

The planners inside the Tangram controllers use Nelder-Mead search [162] to generate the local output targets and the coordination signals to the child controllers. As shown in the outline below, the algorithm moves through the following five modes.

1) Initialize: To find N targets, chose $N + 1$ initial points with random output targets and observe the EDP at each point.

2) Rank: Based on EDP , identify three points out of these $N + 1$ and rank them as Best, Worst, and Lousy (i.e., the point better only than Worst). Compute the centroid of all the $N + 1$ points except Worst. The Best, Worst, and Lousy points, plus the centroid are shown in Figure A.1.

3) Reflect: Find a new point by reflecting the Worst point about the Centroid. This is shown as Point 1 in Figure A.1. If the EDP at this point is better than Worst, Point 1 replaces Worst and the search returns to Step 2. Otherwise, the search moves to Step 4.

4) Contract: The search finds a new point which is the midpoint between Centroid and Worst. This is Point 2 in Figure A.1. If this point is better than Worst, it replaces Worst and the search returns to Step 2. Otherwise, the search moves to Step 5.

5) Shrink: All points except Best are moved towards Best, and search returns to Step 2.

This process repeats until the value of the Best point converges.

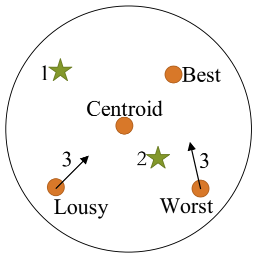


Figure A.1: Nelder-Mead search used by the planner.

REFERENCES

[1] E. Rotem, “Intel Architecture, Code Name Skylake Deep Dive: A New Architecture to Manage Power Performance and Energy Efficiency,” Intel Developer Forum, Aug. 2015.

[2] B. Sinharoy, R. Swanberg, N. Nayar, B. Mealey, J. Stuecheli, B. Schiefer, J. Leenstra, J. Jann, P. Oehler, D. Levitan, S. Eisen, D. Sanner, T. Pflueger, C. Lichtenau, W. Hall, and T. Block, “Advanced features in IBM POWER8 systems,” *IBM Jour. Res. Dev.*, vol. 59, no. 1, pp. 1:1–1:18, Jan. 2015.

[3] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, “Power Management of the Third Generation Intel Core Micro Architecture formerly Codenamed Ivy Bridge,” in *Hot Chips: A Symposium on High Performance Chips*, Aug. 2012.

[4] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, “Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge,” *IEEE Micro*, vol. 32, Mar. 2012.

[5] Terry Myerson, “Windows 10 Embracing Silicon Innovation,” https://blogs.windows.com/windowsexperience/2016/01/15/windows-10-embracing-silicon-innovation/, 2016, Windows Blog.

[6] I. Rickards and A. Kucheria, “Energy Aware Scheduling (EAS) progress update,” http://www.linaro.org/blog/core-dump/energy-aware-scheduling-eas-progress-update/.

[7] E. Fluhr, S. Baumgartner, D. Boerstler, J. Bulzacchelli, T. Diemoz, D. Dreps, G. English, J. Friedrich, A. Gattiker, T. Gloekler, C. Gonzalez, J. Hibbeler, K. Jenkins, Y. Kim, P. Muench, R. Nett, J. Paredes, J. Pille, D. Plass, P. Restle, R. Robertazzi, D. Shan, D. Siljenberg, M. Sperling, K. Stawiasz, G. Still, Z. Toprak-Deniz, J. Warnock, G. Wiedemeier, and V. Zyuban, “The 12-Core POWER8^{TM} Processor With 7.6 Tb/s IO Bandwidth, Integrated Voltage Regulation, and Resonant Clocking,” *IEEE J. Solid-State Circuits*, vol. 50, no. 1, pp. 10–23, Jan. 2015.

[8] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar, “Power and Thermal Management in the Intel Core Duo Processor,” *Intel Technology Journal*, vol. 10, no. 2, pp. 109–122, May 2006.

[9] T. Burd, N. Beck, S. White, M. Paraschou, N. Kalyanasundharam, G. Donley, A. Smith, L. Hewitt, and S. Naffziger, ““Zeppelin”: An SoC for Multichip Architectures,” *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 133–143, Jan. 2019.

[10] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster, “Dynamically Tuning Processor Resources with Adaptive Processing,” *Computer*, vol. 36, no. 12, pp. 49–58, Dec. 2003.

[11] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-purpose Processor Architectures," in *International Symposium on Microarchitecture*, 2000.

[12] C. Isci, G. Contreras, and M. Martonosi, "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management," in *International Symposium on Microarchitecture*, 2006.

[13] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron, "Control-theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[14] K. Sundararajan, T. Jones, and N. Topham, "Smart cache: A self adaptive cache architecture for energy efficiency," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, Jul. 2011.

[15] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. S. B. Altaf, N. E. Jerger, and G. H. Loh, "Modular Routing Design for Chiplet-Based Systems," in *International Symposium on Computer Architecture*, 2018.

[16] T. Vijayaraghavan, Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan, "Design and Analysis of an APU for Exascale Computing," in *International Symposium on High Performance Computer Architecture*, 2017.

[17] Marvell Corporation, "MoChi Architecture," http://www.marvell.com/architecture/mochi/, 2019.

[18] S. Sutardja, "The Future of IC Design Innovation," in *International Solid-State Circuits Conference*, 2015.

[19] Andreas Olofsson, "Common Heterogeneous Integration and IP Reuse Strategies (CHIPS)," https://www.darpa.mil/program/common-heterogeneous-integration-and-ip-reuse-strategies, 2016, Defense Advanced Research Projects Agency.

[20] R. Mahajan, R. Sankman, N. Patel, D. Kim, K. Aygun, Z. Qian, Y. Mekonnen, I. Salama, S. Sharan, D. Iyengar, and D. Mallik, "Embedded Multi-die Interconnect Bridge (EMIB) – A High Density, High Bandwidth Packaging Interconnect," in *IEEE Electronic Components and Technology Conference*, 2016.

[21] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *International Symposium on Computer Architecture*, 2017.

[22] Advanced Micro Devices, Inc, "AMD Ryzen Mobile Processors with Radeon Vega Graphics," https://www.amd.com/en/products/ryzen-processors-laptop, 2019, accessed: 2019.

[23] GlobeNewswire, "AMD Delivers Semi-Custom Graphics Chip For New Intel Processor," http://www.nasdaq.com/press-release/amd-delivers-semicustom-graphics-chip-for-new-intel-processor-20171106-00859, Nov. 2017.

[24] L. Yan, Y. Guo, X. Chen, and H. Mei, "A Study on Power Side Channels on Mobile Devices," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, ser. Internetware '15. New York, NY, USA: ACM, 2015, pp. 30–38. [Online]. Available: http://doi.acm.org/10.1145/2875913.2875934

[25] P. Lifshits, R. Forte, Y. Hoshen, M. Halpern, M. Philipose, M. Tiwari, and M. Silberstein, "Power to peep-all: Inference Attacks by Malicious Batteries on Mobile Devices," *Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 141–158, 2018. [Online]. Available: https://content.sciendo.com/view/journals/popets/2018/4/article-p141.xml

[26] Q. Yang, P. Gasti, G. Zhou, A. Farajidavar, and K. S. Balagani, "On Inferring Browsing Activity on Smartphones via USB Power Analysis Side-Channel," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 5, pp. 1056–1066, May 2017.

[27] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "PowerSpy: Location Tracking Using Mobile Device Power Analysis," in *USENIX Security Symposium*. Washington, D.C.: USENIX Association, 2015, pp. 785–800. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/michalevsky

[28] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to Differential Power Analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr. 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y

[29] L. Zhang, L. Vega, and M. Taylor, "Power Side Channels in Security ICs: Hardware Countermeasures," *arXiv preprint arXiv:1605.00681, https://arxiv.org/abs/1605.00681*, 2016.

[30] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun, "Thermal Covert Channels on Multi-core Platforms," in *USENIX Security Symposium*. Washington, D.C.: USENIX Association, 2015, pp. 865–880. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti

[31] M. A. Islam and S. Ren, "Ohm's Law in Data Centers: A Voltage Side Channel for Timing Power Attacks," in *Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2018, pp. 146–162. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243744

[32] M. Enev, S. Gupta, T. Kohno, and S. N. Patel, "Televisions, Video Privacy, and Powerline Electromagnetic Interference," in *Conference on Computer and Communications Security*, 2011, pp. 537–550.

[33] S. K. Khatamifard, L. Wang, A. Das, S. Kose, and U. R. Karpuzcu, "POWERT Channels: A Novel Class of Covert CommunicationExploiting Power Management Vulnerabilities," in *International Symposium on High Performance Computer Architecture*. IEEE, 2019, pp. 291–303.

[34] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi, "Deconstructing the Energy Consumption of the Mobile Page Load," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3084443

[35] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring," in *USENIX Annual Technical Conference*, ser. USENIX ATC12. USA: USENIX Association, 2012, p. 36.

[36] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES/ISSS 10. New York, NY, USA: Association for Computing Machinery, 2010, p. 105114. [Online]. Available: https://doi.org/10.1145/1878961.1878982

[37] Y. Chen, X. Jin, J. Sun, R. Zhang, and Y. Zhang, "POWERFUL: Mobile App Fingerprinting via Power Analysis," in *IEEE International Conference on Computer Communications*, 2017, pp. 1–9.

[38] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shojafar, A. I. A. Ahmed, S. A. Madani, K. Saleem, and J. J. Rodrigues, "A Survey on Energy Estimation and Power Modeling Schemes for Smartphone Applications," *International Journal of Communication Systems*, vol. 30, no. 11, p. e3234, 2017, e3234 IJCS-16-0606.R1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/dac.3234

[39] Y. Qin and C. Yue, "Website Fingerprinting by Power Estimation Based Side-Channel Attacks on Android 7," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 1030–1039.

[40] Z. Xu, T. Mauldin, Z. Yao, S. Pei, T. Wei, and Q. Yang, "A Bus Authentication and Anti-Probing Architecture Extending Hardware Trusted Computing Base Off CPU Chips and Beyond," in *International Symposium on Computer Architecture*, 2020.

[41] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA," in *USENIX Security Symposium*. Baltimore, MD: USENIX Association, 2018, pp. 585–602. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/alam

[42] Z. Shao, M. A. Islam, and S. Ren, "Your Noise, My Signal: Exploiting Switching Noise for Stealthy Data Exfiltration from Desktop Computers," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 1, May 2020. [Online]. Available: https://doi.org/10.1145/3379473

[43] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori, "Remote Inter-chip Power Analysis Side-channel Attacks at Board-level," in *IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: ACM, 2018, pp. 114:1–114:7.

[44] I. Giechaskiel, K. Rasmussen, and J. Szefer, "C3APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage," in *IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 909–922. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00070

[45] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks, "ICAS: An Extensible Framework for Estimating the Susceptibility of IC Layouts to Additive Trojans," in *IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1078–1095. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00083

[46] C. Lefurgy, "Avoiding Core Meltdown! - Adaptive Techniques for Power and Thermal Management of Multi-Core Processors," https://researcher.watson.ibm.com/researcher/files/us-lefurgy/DAC2013_Lefurgy_v6.pdf, 2013, Tutorial, DAC.

[47] T. Rosedahl, M. Broyles, C. Lefurgy, B. Christensen, and W. Feng, "Power/Performance Controlling Techniques in OpenPOWER," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Springer International Publishing, 2017, pp. 275–289.

[48] N. Brookwood, "EPYC: A Study in Energy Efficient CPU Design," https://www.amd.com/system/files/documents/The-Energy-Efficient-AMD-EPYC-Design.pdf, 2018.

[49] Advanced Micro Devices, Inc, "Understanding Power Management and Processor Performance Determinism," https://www.amd.com/system/files/documents/understanding-power-management.pdf, 2018, Whitepaper.

[50] ——, "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 10h-1FFh Processors," http://developer.amd.com/resources/developer-guides-manuals/, Jul. 2015, Advanced Micro Devices, Inc.

[51] S. Chennupaty, "Thin & Light & High Performance Graphics," in *Hot Chips: A Symposium on High Performance Chips*, 2018, Intel Corporation.

[52] I. Paul, V. Ravi, S. Manne, M. Arora, and S. Yalamanchili, "Coordinated Energy Management in Heterogeneous Processors," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2013.

[53] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili, “Cooperative Boosting: Needy Versus Greedy Power Management,” in *International Symposium on Computer Architecture*, 2013.

[54] I. Paul, W. Huang, M. Arora, and S. Yalamanchili, “Harmonia: Balancing compute and memory power in high-performance GPUs,” in *International Symposium on Computer Architecture*, 2015.

[55] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini, “CoScale: Coordinating CPU and Memory System DVFS in Server Systems,” in *International Symposium on Microarchitecture*, Dec. 2012.

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

[57] ——, “XChange: A Market-based Approach to Scalable Dynamic Multi-resource Allocation in Multicore Architectures,” in *International Symposium on High Performance Computer Architecture*, 2015.

[58] S. Fan, S. M. Zahedi, and B. C. Lee, “The Computational Sprinting Game,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[59] R. Bitirgen, E. Ipek, and J. F. Martinez, “Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach,” in *International Symposium on Microarchitecture*, 2008.

[60] A. M. Rahmani, B. Donyanavard, T. Müch, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt, “SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[61] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, “No ”Power” Struggles: Coordinated Multi-level Power Management for the Data Center,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[62] P. Tembey, A. Gavrilovska, and K. Schwan, “A Case for Coordinated Resource Management in Heterogeneous Multicore Platforms,” in *Computer Architecture*. Springer Berlin Heidelberg, 2012.

[63] H. Zhang and H. Hoffmann, “Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[64] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration," in *International Symposium on Computer Architecture*, 2006.

[65] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price Theory Based Power Management for Heterogeneous Multi-cores," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

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

[67] H. Hoffmann, "JouleGuard: Energy Guarantees for Approximate Applications," in *ACM Symposium on Operating Systems Principles*, 2015.

[68] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel, "Reducing the Energy Usage of Office Applications," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2001.

[69] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile Application-aware Adaptation for Mobility," in *ACM Symposium on Operating Systems Principles*, 1997.

[70] "Distributed Extensible Open Systems (the DEOS project)," http://www.cc.gatech.edu/systems/projects/DEOS/, 2004, Georgia Institute of Technology - College of Computing.

[71] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and Control in Gray-box Systems," in *ACM Symposium on Operating Systems Principles*, 2001, pp. 43–56.

[72] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "DYNAMO: A Cross-Layer Framework for End-to-End QoS and Energy Optimization in Mobile Handheld Devices," *IEEE J. Sel. Areas Commun.*, vol. 25, no. 4, pp. 722–737, May 2007.

[73] M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian, "xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, pp. 73:1–73:23, Jan. 2013.

[74] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. Dutt, R. Gupta, A. Nicolau, S. Shukla, and N. Venkatasubramanian, "A Cross-Layer Approach for Power-Performance Optimization in Distributed Mobile Systems," in *International Parallel and Distributed Processing Symposium*, 2005.

[75] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated Power-performance Optimization in Manycores," in *International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[76] V. Vardhan, D. G. Sachs, W. Yuan, A. F. Harris, S. V. Adve, D. L. Jones, R. H. Kravets, and K. Nahrstedt, "GRACE-2: Integrating Fine-Grained Application Adaptation with Global Adaptation for Saving Energy," *Intl. J. Embed. Sys.*, vol. 4, no. 2, pp. 152–169, 2009.

[77] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "A Taxonomy of Compositional Adaptation," Michigan State University, Tech. Rep., 2004.

[78] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani, "Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control," in *International Symposium on Microarchitecture*, 2013.

[79] A. Filieri, H. Hoffmann, and M. Maggio, "Automated Multi-objective Control for Self-adaptive Software Design," in *Joint Meeting on Foundations of Software Engineering*, 2015.

[80] S. Shevtsov and D. Weyns, "Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 229–241.

[81] M. Broyles, C. J. Cain, T. Rosedahl, and G. J. Silva, "IBM EnergyScale for POWER8 Processor-Based Systems," IBM, Tech. Rep., Nov. 2015.

[82] E. Rotem, U. C. Weiser, A. Mendelson, R. Ginosar, E. Weissmann, and Y. Aizik, "H-EARtH: Heterogeneous Multicore Platform Energy Management," *Computer*, vol. 49, no. 10, pp. 47–55, Oct. 2016.

[83] Intel Support Community, "CPU Throttling Broken for Atom BayTrail CPUs under Windows 10," https://communities.intel.com/thread/78086, 2015, Intel Support Community.

[84] K. Tiri and I. Verbauwhede, "Design Method for Constant Power Consumption of Differential Logic Circuits," in *Design, Automation and Test in Europe*, Mar. 2005, pp. 628–633 Vol. 1.

[85] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang, "Masking the Energy Behavior of DES Encryption [Smart Cards]," in *Design, Automation and Test in Europe Conference and Exhibition*, Mar. 2003, pp. 84–89.

[86] S. Yang, W. Wolf, N. Vijaykrishnan, D. N. Serpanos, and Y. Xie, "Power Attack Resistant Cryptosystem Design: A Dynamic Voltage and Frequency Switching Approach," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 64–69.

[87] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen, "High Efficiency Power Side-channel Attack Immunity using Noise Injection in Attenuated Signature Domain," in *International Symposium on Hardware Oriented Security and Trust*, May 2017, pp. 62–67.

[88] S. Yang, P. Gupta, M. Wolf, D. Serpanos, V. Narayanan, and Y. Xie, "Power Analysis Attack Resistance Engineering by Dynamic Voltage and Frequency Scaling," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 3, pp. 62:1–62:16, Sep. 2012.

[89] N. D. P. Avirneni and A. K. Somani, "Countering Power Analysis Attacks UsingReliable and Aggressive Designs," *IEEE Trans. Comput.*, vol. 63, no. 6, pp. 1408–1420, Jun. 2014.

[90] D. Real, C. Canovas, J. Clediere, M. Drissi, and F. Valette, "Defeating Classical Hardware Countermeasures: A New Processing for Side Channel Analysis," in *Design, Automation and Test in Europe*, Mar. 2008, pp. 1274–1279.

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

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

[93] R. P. Pothukuchi and J. Torrellas, *A Guide to Design MIMO Controllers for Processors*, http://iacoma.cs.uiuc.edu/iacoma-papers/mimoTR.pdf, Apr. 2016.

[94] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: Multilayer Resource Controllers to Maximize Efficiency," in *International Symposium on Computer Architecture*, 2018.

[95] ——, "Structured Singular Value Control for Modular Resource Management in Multi-layer Computers," in *IEEE Conference on Decision and Control*, Dec. 2018.

[96] R. P. Pothukuchi, S. Y. Pothukuchi, P. G. Voulgaris, and J. Torrellas, "Control Systems for Computing Systems: Making computers efficient with modular, coordinated, and robust control," *IEEE Control Syst. Mag.*, vol. 40, no. 2, pp. 30–55, 2020.

[97] R. P. Pothukuchi, J. L. Greathouse, K. Rao, C. Erb, L. Piga, P. G. Voulgaris, and J. Torrellas, "Tangram: Integrated Control of Heterogeneous Computers," in *International Symposium on Microarchitecture*, Oct. 2019.

[98] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Maya: Falsifying Power Sidechannels with Dynamic Control," *arXiv preprint arXiv:1907.09440, https://arxiv.org/abs/1907.09440*, 2019.

[99] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "A Distributed and Self-calibrating Model-Predictive Controller for Energy and Thermal Management of High-Performance Multicores," in *Conference on Design, Automation and Test in Europe*, Mar. 2011.

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

[101] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained Power Control for Chip Multiprocessors with Online Model Estimation," in *International Symposium on Computer Architecture*, 2009.

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

[103] F. Zanini, C. Jones, D. Atienza, and G. De Micheli, "Multicore Thermal Management using Approximate Explicit Model Predictive Control," in *International Symposium on Circuits and Systems*, May 2010.

[104] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable Power Control for Many-core Architectures Running Multi-threaded Applications," in *International Symposium on Computer Architecture*, 2011.

[105] M. Athans, "The Role and Use of the Stochastic Linear-Quadratic-Gaussian Problem in Control System Design," *IEEE Trans. Autom. Control*, vol. 16, no. 6, pp. 529–552, Dec. 1971.

[106] J. C. Doyle, J. E. Wall, and G. Stein, "Performance and Robustness Analysis for Structured Uncertainty," in *IEEE Conference on Decision and Control*, Dec. 1982.

[107] S. Skogestad, M. Morari, and J. C. Doyle, "Robust Control of Ill-conditioned Plants: High-purity Distillation," *IEEE Trans. Autom. Control*, vol. 33, no. 12, pp. 1092–1105, Dec. 1988.

[108] D.-W. Gu, P. H. Petkov, and M. M. Konstantinov, *Robust Control Design with MATLAB*, 2nd ed. Springer, 2013.

[109] L. Ljung, *System Identification : Theory for the User*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[110] R. P. Pothukuchi, A. Ansari, B. Gopireddy, and J. Torrellas, "Sthira: A Formal Approach to Minimize Voltage Guardbands under Variation in Networks-on-Chip for Energy Efficiency," in *International Conference on Parallel Architectures and Compilation Techniques*, 2017.

[111] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann, "Automated Control of Multiple Software Goals Using Multiple Actuators," in *Joint Meeting on Foundations of Software Engineering*, 2017.

[112] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems," in *International Symposium on Computer Architecture*, 2013.

[113] V. Hanumaiah, D. Desai, B. Gaudette, C.-J. Wu, and S. Vrudhula, "STEAM: A Smart Temperature and Energy Aware Multicore Controller," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 5s, pp. 151:1–151:25, Oct. 2014.

[114] C. Dubach, T. Jones, E. Bonilla, and M. O'Boyle, "A Predictive Model for Dynamic Microarchitectural Adaptivity Control," in *International Symposium on Microarchitecture*, Dec. 2010.

[115] H. Jung, P. Rong, and M. Pedram, "Stochastic Modeling of a Thermally-Managed Multi-Core System," in *Design Automation Conference*, Jun. 2008.

[116] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *International Symposium on Microarchitecture*, Dec. 2006.

[117] A. Dhodapkar and J. Smith, "Managing Multi-configuration Hardware via Dynamic Working Set Analysis," in *International Symposium on Computer Architecture*, 2002.

[118] N. Beck, S. White, M. Paraschou, and S. Naffziger, ""Zeppelin": An SoC for Multichip Architectures," in *International Solid-State Circuits Conference*, 2018.

[119] T. Karkhanis and J. Smith, "A First-Order Superscalar Processor Model," in *International Symposium on Computer Architecture*, Jun. 2004.

[120] B. C. Lee and D. Brooks, "Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[121] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[122] B. C. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable Performance Regression for Scalable Multiprocessor Models," in *International Symposium on Microarchitecture*, 2008.

[123] L. Ljung, "Black-box Models from Input-output Measurements," in *IEEE Instrumentation and Measurement Technology Conference*, May 2001.

[124] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "ECOSystem: Managing Energy As a First Class Operating System Resource," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[125] J. Flinn and M. Satyanarayanan, "Energy-aware Adaptation for Mobile Applications," in *ACM Symposium on Operating Systems Principles*, 1999.

[126] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A Platform for OS-level Power Management," in *ACM European conference on Computer systems*, 2009.

[127] K. Yan, X. Zhang, and X. Fu, "Characterizing, Modeling, and Improving the QoE of Mobile Devices with Low Battery Level," in *International Symposium on Microarchitecture*, 2015.

[128] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," in *International Symposium on Microarchitecture*, 2001.

[129] E. K. Ardestani and J. Renau, "ESESC: A Fast Multicore Simulator Using Time-Based Sampling," in *International Symposium on High Performance Computer Architecture*, 2013.

[130] V. Spiliopoulos, A. Bagdia, A. Hansson, P. Aldworth, and S. Kaxiras, "Introducing DVFS-Management in a Full-System Simulator," in *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug. 2013.

[131] MATLAB, *MATLAB and Simulink Release 2015a.* Natick, Massachusetts: The MathWorks Inc., 2015.

[132] W. Wang, P. Mishra, and S. Ranka, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems," in *Design Automation Conference*, 2011.

[133] V. Hanumaiah, "Unified Framework for Energy-proportional Computing in Multicore Processors: Novel Algorithms and Practical Implementation," Ph.D. dissertation, Arizona State University, 2013.

[134] "MATLAB and Robust Control Toolbox Release 2016a," The MathWorks Inc., Natick, Massachusetts, 2016.

[135] Taylor IoT Kidd, "Power Management States: P-States, C-States, and Package C-States," https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states, 2014, Intel Developer Zone.

[136] Microsoft, "Processor power management in Windows 7 and Windows Server 2008 R2," https://msdn.microsoft.com/en-us/library/windows/hardware/dn613983(v=vs.85).aspx, 2012, Microsoft Developer Network.

[137] HardKernel, "ODROID-XU3," http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127.

[138] ARM$^{®}$, "big.LITTLE Technology: The Future of Mobile," https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, 2013, White Paper.

[139] B. Jeff, "big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling," https://www.arm.com/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf, Nov. 2013, White Paper.

[140] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE™ Technology," https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf, 2013, White Paper.

[141] U. Rezki and V. Wool, "Doing big.LITTLE right: little and big obstacles," in *Embedded Linux Conference*, Mar. 2015.

[142] M. Anderson, "Implementation of the Global Task Scheduler in big.LITTLE Android Platforms," in *Embedded Linux Conference*, Mar. 2015.

[143] D. Kim and A. Daniel, "Kernel driver exynos_tmu," https://www.kernel.org/doc/Documentation/thermal/exynos_thermal, Online Documentation.

[144] "Samsung Exynos TMU Implementation," https://github.com/hardkernel/linux/blob/odroidxu3-3.10.y/drivers/thermal/exynos_thermal.c, Source Code.

[145] "Samsung Exynos TMU Header," https://github.com/hardkernel/linux/blob/odroidxu3-3.10.y/arch/arm/mach-exynos/include/mach/tmu.h, Source Code.

[146] P. Newbold, "The Principles of the Box-Jenkins Approach," *J. Oper. Res. Soc.*, vol. 26, no. 2, pp. 397–412, 1975.

[147] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/.

[148] C. Imes and H. Hoffmann, "Bard: A Unified Framework for Managing Soft Timing and Power Constraints," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2016.

[149] D. Brodowski and N. Golde, "Linux CPUFreq Governors," https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt, Online Documentation.

[150] "Intel Dynamic Platform and Thermal Framework (DPTF) for Chromium OS," https://01.org/intel%C2%AE-dynamic-platform-and-thermal-framework-dptf-chromium-os/documentation/implementation-design-and-source-code-organization, Intel Corporation, 2015, accessed: 2019.

[151] T. Rosedahl, "OCC Firmware Code is Now Open Source," https://openpowerfoundation.org/occ-firmware-code-is-now-open-source/, 2014, Code:https://github.com/open-power/docs/blob/master/occ/OCC_overview.md.

[152] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar, "The Need for Speed and Stability in Data Center Power Capping," in *International Green Computing Conference*, 2012.

[153] M. A. Prospero, "AMD Announces New Low-Power APUs for Tablets and Notebooks," https://www.laptopmag.com/articles/amd-mullins-beema-apu, Apr. 2014.

[154] SKYMTL, "AMD Mullins & Beema Mobile APUs Preview," http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/66162-amd-mullins-beema-mobile-apus-preview.html, Apr. 2014.

[155] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar, "Performance Directed Energy Management for Main Memory and Disks," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[156] GIGA-BYTE Technology Co., Ltd., "GIGABYTE," http://www.gigabyte.us/Motherboard/GA-AX370-Gaming-5-rev-10#kf, 2019, accessed: 2019.

[157] Advanced Micro Devices, Inc, "AMD Ryzen," http://www.amd.com/en/ryzen, 2019, accessed: 2019.

[158] Micro-Star Int'l Co.,Ltd., "MSI Graphics Cards," https://www.msi.com/Graphics-card/Radeon-RX-580-8G, 2019, accessed: 2019.

[159] Advanced Micro Devices, Inc, "AMD Radeon RX 580 Graphics," https://www.amd.com/en/products/graphics/radeon-rx-580, 2019, accessed: 2019.

[160] Intel Corporation, "Voltage Regulator Module (VRM) and Enterprise Voltage Regulator-Down (EVRD) 11.1," https://www.intel.ie/content/www/ie/en/power-management/voltage-regulator-module-enterprise-voltage-regulator-down-11-1-guidelines.html, 2009, accessed: 2019.

[161] J. Gómez-Luna, I. El Hajj, V. Chang, Li-Wen Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2017.

[162] J. Mathews and K. Fink, *Numerical Methods Using MATLAB.* Pearson Education, Limited, 2006. [Online]. Available: https://books.google.com/books?id=-DpDPgAACAAJ

[163] Kenneth Mitchell and Elliot Kim, "Optimizing for AMD Ryzen CPU," http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2017/03/GDC2017-Optimizing-For-AMD-Ryzen.pdf, 2017, accessed: 2019.

[164] NASA Advanced Supercomputing Division, "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html, 2003.

[165] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2008.

[166] Advanced Micro Devices, Inc, "AMD FX Processors Unleashed — a Guide to Performance Tuning with AMD OverDrive and the new AMD FX Processors," https://www.amd.com/Documents/AMD_FX_Performance_Tuning_Guide.pdf, Oct. 2011, Advanced Micro Devices, Inc.

[167] S. S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current Events: Identifying Webpages by Tapping the Electrical Outlet," in *Computer Security – ESORICS 2013*, J. Crampton, S. Jajodia, and K. Mayes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 700–717.

[168] M. A. Islam, S. Ren, and A. Wierman, "Exploiting a Thermal Side Channel for Power Attacks in Multi-Tenant Data Centers," in *Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2017, pp. 1079–1094. [Online]. Available: http://doi.acm.org/10.1145/3133956.3133994

[169] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397.

[170] E. Vasilakis, "An Instruction Level Energy Characterization of Arm Processors," https://www.ics.forth.gr/carv/greenvm/files/tr450.pdf, 2015.

[171] Y. S. Shao and D. Brooks, "Energy Characterization and Instruction-level Energy Model of Intel's Xeon Phi Processor," in *International Symposium on Low Power Electronics and Design*, Sep. 2013, pp. 389–394.

[172] R. Callan, N. Popovic, A. Daruna, E. Pollmann, A. Zajic, and M. Prvulovic, "Comparison of Electromagnetic Side-channel Energy Available to the Attacker from Different Computer Systems," in *2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, Aug. 2015, pp. 219–223.

[173] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay, "How TrustZone Could Be Bypassed: Side-Channel Attacks on a Modern System-on-Chip," in *Information Security Theory and Practice*, G. P. Hancke and E. Damiani, Eds. Cham: Springer International Publishing, 2018, pp. 93–109.

[174] "Power Profiles for Android," https://source.android.com/devices/tech/power/, android Open Source Project.

[175] H. Hlavacs, T. Treutner, J. Gelas, L. Lefevre, and A. Orgerie, "Energy Consumption Side-Channel Attack at Virtual Machines in a Cloud," in *International Conference on Dependable, Autonomic and Secure Computing*, Dec. 2011, pp. 605–612.

[176] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels," in *Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1626–1638. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978353

[177] D. Genkin, I. Pipman, and E. Tromer, "Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs," in *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 242–260. [Online]. Available: https://doi.org/10.1007/978-3-662-44709-3_14

[178] M. Zhao and G. E. Suh, "FPGA-based Remote Power Side-channel Attacks," in *IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 229–244.

[179] S. Trimberger and S. McNeil, "Security of FPGAs in Data Centers," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 117–122.

[180] N. Chawla, A. Singh, M. Kar, and S. Mukhopadhyay, "Application Inference using Machine Learning based Side Channel Analysis," in *International Joint Conference on Neural Networks*. IEEE, 2019, pp. 1–8.

[181] M. A. Islam, L. Yang, K. Ranganath, and S. Ren, "Why Some Like It Loud: Timing Power Attacks in Multi-tenant Data Centers Using an Acoustic Side Channel," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 1, pp. 6:1–6:33, Apr. 2018. [Online]. Available: http://doi.acm.org/10.1145/3179409

[182] S. Pandruvada, "Running Average Power Limit – RAPL," https://01.org/blogs/2014/running-average-power-limit--rapl, Published: June, 2014.

[183] A. van de Ven and J. Pan, "Intel Powerclamp Driver," https://www.kernel.org/doc/Documentation/thermal/intel_powerclamp.txt, Last modified: April, 2017.

[184] X. Wang, Q. Zhou, J. Harer, G. Brown, S. Qiu, Z. Dou, J. Wang, A. Hinton, C. A. Gonzalez, and P. Chin, "Deep Learning-based Classification and Anomaly Detection of Side-channel Signals," in *Proc. SPIE 10630, Cyber Sensing*, 2018. [Online]. Available: https://doi.org/10.1117/12.2311329

[185] FFmpeg Developers, "ffmpeg tool," http://ffmpeg.org/, 2016.

[186] Xiph.org Video Test Media, "derf's collection," https://media.xiph.org/.

[187] MathWorks, "Find Abrupt Changes in Signal," https://www.mathworks.com/help/signal/ref/findchangepts.html, Accessed: April, 2019.

[188] John M, "Intel Digital Random Number Generator (DRNG) Software Implementation Guide," https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-\implementation-guide, Oct. 2018.

[189] Intel, "Random Number Generator IP Core User Guide ," https://www.intel.com/content/www/us/en/programmable/documentation/dmi1455632999173.html, Feb. 2017.

[190] C. J. Hughes and S. V. Adve, "A Formal Approach to Frequent Energy Adaptations for Multimedia Applications," in *International Symposium on Computer Architecture*, 2004.

[191] M. C. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," in *International Symposium on Computer Architecture*, 2003.

[192] R. Sen and D. A. Wood, "Reuse-based Online Models for Caches," in *ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2013.

[193] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, and G. Magklis, "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," in *International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2002.

[194] A. Mishra, S. Srikantaiah, M. Kandemir, and C. Das, "CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010.

[195] K. Ma, X. Wang, and Y. Wang, "DPPC: Dynamic Power Partitioning and Control for Improved Chip Multiprocessor Performance," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1736–1750, Jul. 2014.

[196] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. Clark, "Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors," in *International Symposium on Low Power Electronics and Design*, Aug. 2005.

[197] C. Dubach, T. M. Jones, and E. V. Bonilla, "Dynamic Microarchitectural Adaptation Using Machine Learning," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 31:1–31:28, Dec. 2013.

[198] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization Power Management for Chip Multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[199] W. Wang and P. Mishra, "Leakage-Aware Energy Minimization Using Dynamic Voltage Scaling and Cache Reconfiguration in Real-Time Systems," in *International Conference on VLSI Design*, Jan. 2010.

[200] R. Bitirgen, E. İpek, and J. F. Martínez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach," in *International Symposium on Microarchitecture*, 2008.

[201] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher, "Integrating Adaptive Components: An Emerging Challenge in Performance-Adaptive Systems and a Server Farm Case-Study," in *International Real-Time Systems Symposium*, 2007.

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

[203] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and H. Ye, "Application-Specifc Performance-Aware Energy Optimization on Android Mobile Devices," in *International Symposium on High Performance Computer Architecture*, Feb. 2017.

[204] M. H. Santriaji and H. Hoffmann, "GRAPE: Minimizing Energy for GPU Applications with Performance Requirements," in *International Symposium on Microarchitecture*, 2016.

[205] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi, "Dynamic GPGPU Power Management Using Adaptive Model Predictive Control," in *International Symposium on High Performance Computer Architecture*, 2017.

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

[207] S. S. Jha, W. Heirman, A. Falcón, T. E. Carlson, K. Van Craeynest, J. Tubella, A. González, and L. Eeckhout, "Chrysso: An Integrated Power Manager for Constrained Many-core Processors," in *ACM International Conference on Computing Frontiers*, 2015.

[208] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout, "Maximizing Heterogeneous Processor Performance Under Power Constraints," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 29:1–29:23, Sep. 2016.

[209] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, "Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors," in *International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[210] T. Baruah, Y. Sun, S. Dong, D. Kaeli, and N. Rubin, "Airavat: Improving Energy Efficiency of Heterogeneous Applications," in *Conference on Design, Automation and Test in Europe*, Mar. 2018.

[211] B. Munger, D. Akeson, S. Arekapudi, T. Burd, H. R. Fair, J. Farrell, D. Johnson, G. Krishnan, H. McIntyre, E. McLellan, S. Naffziger, R. Schreiber, S. Sundaram, J. White, and K. Wilcox, "Carrizo: A High Performance, Energy Efficient 28 nm APU," *IEEE J. Solid-State Circuits*, vol. 51, no. 1, pp. 105–116, 2016.

[212] S. Sundaram, S. Samabmurthy, M. Austin, A. Grenat, M. Golden, S. Kosonocky, and S. Naffziger, "Adaptive Voltage Frequency Scaling Using Critical Path Accumulator Implemented in 28nm CPU," in *International Conference on VLSI Design*, 2016.

[213] Advanced Micro Devices, Inc, "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 70h-7Fh Processors," http://developer.amd.com/resources/developer-guides-manuals/, Jun. 2018, Advanced Micro Devices, Inc.

[214] X. Wang, "Intelligent Power Allocation: Maximize performance in the thermal envelope," ARM White Paper, Mar. 2017.

[215] N. Almoosa, W. Song, Y. Wardi, and S. Yalamanchili, "A Power Capping Controller for Multicore Processors," in *American Control Conference*, 2012.

[216] X. Chen, Y. Wardi, and S. Yalamanchili, “Power Regulation in High Performance Multicore Processors,” in *IEEE Conference on Decision and Control*, 2017.

[217] K. Rao, W. Song, S. Yalamanchili, and Y. Wardi, “Temperature Regulation in Multicore Processors using Adjustable-gain Integral Controllers,” in *IEEE Conference on Control Applications*, 2015.

[218] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps,” in *International Symposium on Microarchitecture*, 2011.

[219] C. Hankendi, A. K. Coskun, and H. Hoffmann, “Adapt&Cap: Coordinating System-and Application-Level Adaptation for Power-Constrained Systems,” *IEEE Des. Test*, vol. 33, no. 1, pp. 68–76, 2016.

[220] M. Conti, M. Nati, E. Rotundo, and R. Spolaor, “Mind The Plug! Laptop-User Recognition Through Power Consumption,” in *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, ser. IoTPTS ’16. New York, NY, USA: ACM, 2016, pp. 37–44. [Online]. Available: http://doi.acm.org/10.1145/2899007.2899009

[221] T. Kasper, D. Oswald, and C. Paar, “EM Side-Channel Attacks on Commercial Contactless Smartcards Using Low-Cost Equipment,” in *Information Security Applications*, H. Y. Youm and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 79–93.

[222] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative Instructions and Software Model for Isolated Execution,” in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP ’13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[223] ARM, “ARM Security Technology Building a Secure System using TrustZone Technology,” http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, Apr. 2009.

[224] E. M. Benhani and L. Bossuet, “DVFS as a Security Failure of TrustZone-enabled Heterogeneous SoC,” in *International Conference on Electronics, Circuits and Systems (ICECS)*, Dec. 2018, pp. 489–492.

[225] G. B. Ratanpal, R. D. Williams, and T. N. Blalock, “An On-Chip Signal Suppression Countermeasure to Power Analysis Attacks,” *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 3, pp. 179–189, Jul. 2004.

[226] W. Yu, O. A. Uzun, and S. Köse, “Leveraging On-chip Voltage Regulators As a Countermeasure Against Side-channel Attacks,” in *Design Automation Conference*, ser. DAC ’15. New York, NY, USA: ACM, 2015, pp. 115:1–115:6.

[227] J. Yang, F. Dai, J. Wang, J. Zeng, Z. Zhang, J. Han, and X. Zeng, “Countering Power Analysis Attacks by Exploiting Characteristics of Multicore Processors,” *IEICE Electronics Express*, vol. advpub, 2018.

[228] A. Althoff, J. McMahan, L. Vega, S. Davidson, T. Sherwood, M. Taylor, and R. Kastner, “Hiding Intermittent Information Leakage with Architectural Support for Blinking,” in *International Symposium on Computer Architecture*, Jun. 2018, pp. 638–649.

[229] K. Baddam and M. Zwolinski, “Evaluation of Dynamic Voltage and Frequency Scaling as a Differential Power Analysis Countermeasure,” in *International Conference on VLSI Design*, Jan. 2007, pp. 854–862.