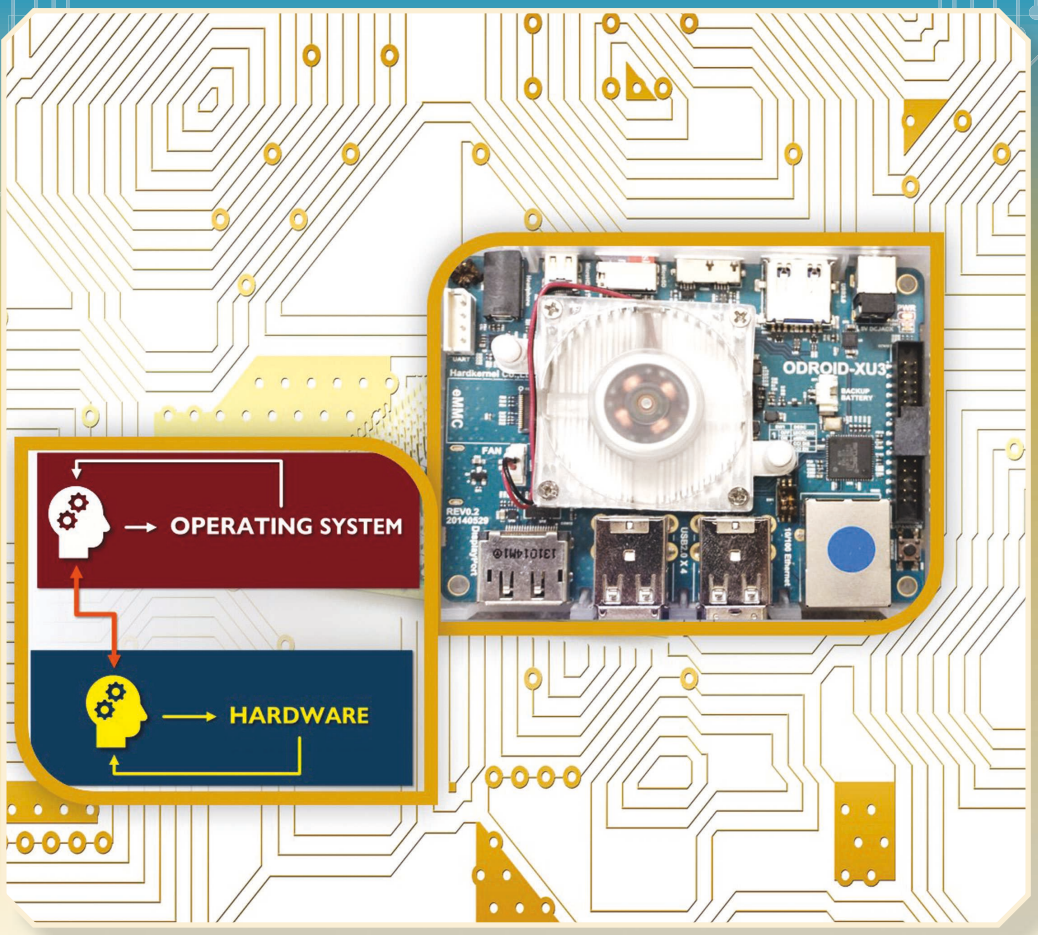

Control Systems for Computing Systems

MAKING COMPUTERS EFFICIENT WITH MODULAR, COORDINATED, AND ROBUST CONTROL

RAGHAVENDRA PRADYUMNA POTHUKUCHI,
SWETA YAMINI POTHUKUCHI,
PETROS G. VOULGARIS, and JOSEP TORRELLAS

Digital Object Identifier 10.1109/MCS.2019.2961733
Date of current version: 17 March 2020

Computing is taking a central role in advancing science, technology, and society, facilitated by increasingly capable systems. Computers are expected to perform a variety of tasks, including life-critical functions, while the resources they require (such as storage and energy) are becoming increasingly limited. To meet expectations, computers use control algorithms that monitor the requirements of the applications they run and reconfigure themselves in response [1]–[5].

1066-033X/20©2020IEEE

As computer systems grow more complex, they are being built with multiple modular layers that are designed independently and interact with standardized interfaces (see "Multilayer-Computer Organization" for an overview). These layers, such as the hardware and the operating system (OS), have separate functions and are developed by different expert teams that are often from separate companies. The designs in one layer must interoperate with several variants in the other layer. For example, an OS must work with many processor designs and vice versa. Moreover, a computer system with given hardware and OS layers must efficiently run several types of user applications.

Naturally, the inputs, outputs, control goals, and control logic are different in each layer. For example, hardware controllers typically measure the application throughput, power consumption, and temperature. The controllers change the processor's operating speed, called the clock frequency (or simply, frequency), the number of cores, and the amount of available storage. In the OS, the scheduling logic decides which applications run on each core at any time. The goals for the process scheduling vary by OS designs but generally include some of the following: high core utilization, high throughput, low power consumption, and fairness. However, the metrics of an application's execution on the computer (that is, the time the application takes to complete, the energy it consumes, and the task throughput) is a function of all of the layers.

Fully decoupled and centralized control architectures are inappropriate to manage modularly designed computers. A centralized controller that can access systemwide outputs and inputs is not practical because the layers are designed by different companies that tightly protect their subsystems' internal designs. Moreover, centralized control does not allow a layer to interoperate with other layers and must be redesigned, even if a single layer in the system changes [6]–[8]. Consequently, centralized controllers that operate across system layers do not fit the design of modern computer systems and are seldom found in actual products.

The other alternative is to use fully decoupled controllers in each layer. This design is modular. However, the separate controllers miss the interlayer interaction and cannot efficiently manage the full system [7], [9]. Controllers that appear to manage the individual layers effectively in isolation fare poorly when deployed in the full system. However, due to lack of alternatives, this is the most common design in existing computers.

New designs are required in which each layer is regulated by a modular resource controller, and the different controllers coordinate only with interfaces for overall efficiency. Furthermore, it is important to design these controllers with formal methods, such as control theory, instead of heuristics (see "Summary"). As computers have grown in complexity, it has become increasingly difficult to design, tune, and verify the heuristics even for a single layer. Despite intense design efforts, these heuristics can behave unexpectedly due to a lack of robustness [10]–[12]. When such heuristics operate in multiple layers, the overall efficiency can be poor. See "Lack of Coordination With Heuristic Control" to learn more about how heuristics in different layers can destructively interfere with each other in an IBM processor. Indeed, it is unfortunate that heuristics remain the popular choice for computer control in production and research.

This work proposes a modular design to manage multilayer computers using structured singular value (SSV) control, or μ synthesis. This approach differs from earlier designs [13]–[15] that focus on particular applications, such as a database server, or a single system layer, such as the hardware processor. The key idea in this article is that modeling limitations and interlayer interactions are considered to be uncertain when designing a controller for a computer layer. Furthermore, each controller reads signals from other layers to better coordinate under uncertainty. The design also has "optimizer" modules that provide changing targets to the μ controllers to maximize the computer-output combinations. This design is a significant advance beyond the state of the art in computer control.

Summary

Computers are operating in increasingly constrained environments and being equipped with intelligent controllers for resource management. Resource controllers keep computers efficient by customizing the usage of limited resources like energy and storage to match application requirements. However, the operation of computers is structured in multiple layers, such as the hardware, operating system, and networking layers. Each of those layers is built, run, and controlled independently. It is desirable to manage the overall system by coordinating the operation of the different controllers and relying on a systematic control methodology rather than heuristics.

This article presents a new approach to build coordinated multilayer controllers for computers. As opposed to the existing practice of heuristic computer control, the proposed approach is based on standard techniques from linear robust control theory. The key idea is that interlayer interaction, among others, is considered an uncertainty when designing a controller for a computer layer. We prototype this design on a real computer to demonstrate its effectiveness over existing methods. This is the first work to use linear robust control methods for resource efficiency in computers.

The article calls the attention of control systems researchers to the topic of building formal controllers for computer systems. Despite significant progress in control theory, computer resource control today is predominantly heuristic. There is a need for novel contributions that computer designers can take up. As an initial step, we present several challenges we faced in our design to solicit advanced solutions from the control systems community.

Multilayer-Computer Organization

A computer system can be conceptually divided into layers: the hardware and the operating system (OS). The hardware contains one or more processor chips, storage, and other circuitry. Figure S1 shows a computer with one processor chip in the hardware and an OS running as software. The system is running several user applications.

The processor chip, which is the key component in the hardware, has multiple processing units called cores and the storage necessary to run applications. Figure S1 shows two types of cores in the processor chip. Having different types of cores is useful to provide various levels of throughput (a measure of the computer's performance) and power consumption. For example, a complex core can generally deliver more throughput but consumes more power, while a simpler core has a relatively lower throughput and power consumption. Therefore, it may be more energy efficient to use the simpler core in certain cases.

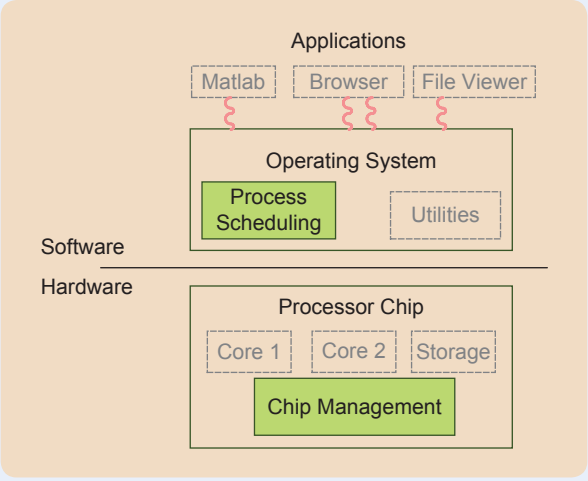

FIGURE S1 The multilayer organization of computer systems.

The OS runs as software on the processor and sets up the environment for user applications. It schedules applications on the processor and provides utilities such as storage management. Process scheduling is an important part of the OS that decides which applications can run on each core at any time. The process-scheduling goals vary by OS but generally include some of the following: high core utilization, high throughput, low power use, and fairness.

Finally, in the environment provided by the OS, there are multiple applications, such as Matlab, browsers, and file viewers, that users run on the computer. Typically, each application can launch multiple tasks to perform its work. As an example, Figure S1 has three applications that run as one task, two tasks, and one task, respectively. These applications (or application tasks) are scheduled by the OS to execute instructions on the hardware. Each task runs on one hardware core, and a multitasking application can use multiple cores simultaneously.

The hardware and OS layers are the common layers found in nearly all systems. Some systems have more software layers on top of the OS layer, with each providing a different abstraction for the immediately higher layer. Figure S1 highlights the components responsible for dynamic control of the system, which are the hardware chip management units and OS process-scheduling logic. The decisions of these components determine a given application's behavior, such as its throughput, energy consumption, and temperature. When these decisions are aligned with the nature of the application's work, the system can efficiently process the application, completing the task quickly and consuming only the minimum energy necessary.

This article makes two contributions. First, it describes the application of μ synthesis for modular, coordinated computer control. We consider several challenges in the practice of computer control so that the proposed architectures are more easily adopted in industrial computer system design. As opposed to examining specific user applications running on a computer or individual layers, we examine how multilayer computers can be better designed and propose how the controllers in the layers must be built to efficiently run many applications. The effectiveness of our design is demonstrated on a prototype computer. We show how standard, robust control methods provide significant benefits to computer systems, making this application of μ synthesis to computing the first of its kind.

Second, this article calls attention to the problem of building distributed and modular coordinated controllers for computer systems. This problem is especially important as computer systems are becoming increasingly modular and distributed. Despite decades of progress in control theory, few control-theoretic concepts have been applied to computer system control. More theoretically solid and practical contributions are needed. Therefore, we present several design challenges for the general problem of building resource-efficient multilayer computers to solicit advanced solutions from the control systems community.

This article has more breadth and depth than our recent work [16], with the goal of drawing the attention of the IEEE Control Systems Society researchers. We have expanded the discussion of multilayer-computer organization, interlayer interaction, the modular-control problem, the suitability of control theory, design challenges, and additional results from system identification and overall evaluation. We also provide the full details of our prototype design that were not present in [16] so that researchers can test new approaches to the problem. Finally, we bring our insights from prior work [12], [17] to the control systems community. Next, we present a representative computer system and describe

The processor chip management unit optimizes the chip for the user applications and protects the hardware from hazardous operating conditions, such as high temperatures. A common optimization goal is to reduce the power use and temperature while maximizing the throughput. The chip management unit monitors some of the hardware outputs, including the chip's throughput, power, and temperature. It sets the configurable inputs such as the processor's operating speed (called the clock frequency or simply, frequency), number of cores, and available storage. It is common in modern computers to have multiple processors and storage chips, each with its own management units.

The outputs and inputs in each layer are tightly coupled. For example, in the hardware, the frequency and number of cores can significantly affect the power, temperature, and throughput of the processor. However, their impact is different across applications. Poor input choices can result in a high power consumption without generating a high throughput. Similarly, the outputs and inputs in the OS are also tightly coupled.

MODULAR DESIGN AND INTERACTION BETWEEN LAYERS

Due to the complexity of computer systems, each layer is designed independently by expert teams, possibly from different companies. The names of some of these independently designed products are listed in Figure S2. With this modular design, a layer can be reused in many variants of the other layers. For example, the same hardware can run multiple OSs, and the same OS can be used on many hardware configurations. This is made possible by specifying interface standards that the layers must conform to, irrespective of their internal design.

Unfortunately, the modular design of computers makes it challenging to achieve system-wide efficiency with dynamic control. The controllers, with their layers, are designed independently. Their internal details are proprietary and not shared by designers. However, a controller influences the dynamics of the other layers and full system. For example, the power consumed by the computer depends on the processor frequency and assignment of applications to cores by the OS. However, the hardware chip management unit can control only frequency, not the OS's scheduling policy, and vice versa. This can result in inefficient operation. As an example, when the OS schedules application tasks on certain cores and expects a high throughput, the actual throughput may be poor if the hardware controller reduced the frequency of those cores, possibly to lower the power consumption. See "Lack of Coordination With Heuristic Control" for a description of such a case, as identified on an IBM computer. The immediate need for building efficient computer systems is to design a control system that conforms to the computer's modular structure and coordinates decisions across layers for system-wide efficiency.

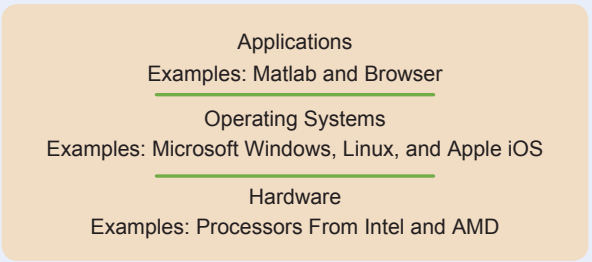

FIGURE S2 The modular design of computer layers. Proprietary names are registered properties of the respective companies.

the problem of modular, coordinated resource control in computers.

REPRESENTATIVE PROTOTYPE SYSTEM

We use an ODROID-XU3 computer board [18] as a prototype system. As is common to most computers, this system has two layers: the hardware and OS. The processor in the hardware is an Exynos 5422 with eight cores built using ARM big.LITTLE technology [19] by Samsung. The OS is Ubuntu 15.04, which is based on Linux. Figure 1 shows the system. We run several applications from the PARSEC 2.1 application suite [20] and SPEC06 suite [21].

Figure 2 is a schematic diagram of the system with the inputs and outputs of each layer. The processor in the hardware layer has eight units, or cores. Four of them, the big cores, can generate a high throughput and consume a significant amount of power. The remaining ones, the little cores, generate a lower throughput but consume much less power. The little and big cores are organized as separate clusters. Applications on this computer can create multiple software tasks (called threads) that execute in parallel to speed up the system's performance. Hence, there can be many applications and threads running simultaneously. We refer to each schedulable entity (an application or application thread) as a task.

HARDWARE- AND OPERATING SYSTEM LAYER INPUTS AND OUTPUTS

Four outputs in the hardware layer are considered, as shown in Figure 2: the power consumed by the little cluster ($\text{Power}_{\text{little}}$, measured in watts), the power consumed by the big cluster ($\text{Power}_{\text{big}}$, measured in watts), the temperature of the hottest location (measured in degrees Celsius), and the throughput of the processor [also called performance, measured in billions of instructions per second (BIPS)]. Among these, the power consumed by either cluster and the temperature are critical for system integrity.

Lack of Coordination With Heuristic Control

Vega et al. discuss how heuristics destructively interfere in a production IBM POWER7 computer system [10]. In this machine, there is a hardware controller that changes the speed (frequency) of each processor core to maintain a high utilization. Utilization is defined as the percentage of the clock cycles during which the application actively uses the processor. Reducing the processing speed decreases the number of clock cycles. Hence, there are fewer cycles during which the application does not use the processor at all; therefore reducing the frequency increases the processor utilization. The lower speed also reduces the power consumed by the computer. Thus, the hardware controller aims to improve the utilization and reduce the power consumption when applications are not using the hardware.

In the operating system, a task scheduler consolidates tasks onto cores and turns off the remaining cores to save power and improve the utilization. When tasks are not fully using a core, there are many idle cycles. By consolidating tasks from several low-utilization cores onto a subset of cores, fully idle ones can be shut down, and the utilization of the active cores improves. Therefore, when the system's utilization decreases, it is expected that the scheduler will consolidate tasks to reduce power without hurting performance.

Unfortunately, in scenarios of low utilization, the hardware controller immediately reduces the speed of the cores to increase utilization, preventing the scheduler from consolidating tasks and power-gating the cores. On identifying cores with high utilizations, the scheduler turns on more cores and moves tasks to them. This behavior alternates until all of the cores in the system are active and set to the lowest speed. The result is poor performance and wasted energy.

FIGURE 1 The Odroid-XU3 used as a prototype in this article. The processor is a Samsung Exynos 5422 with eight cores, and it was built using ARM big.LITTLE technology. The operating system is Ubuntu 15.04.

The performance counters and temperature sensors can provide accurate measurements at the millisecond timescale. However, the onboard power sensor has an update interval of nearly 300 ms. This time span is long because the sensor accumulates energy measurements and divides them by the update period to provide accurate power values. Therefore, we chose the sampling period in this layer to be 0.5 s. This is comparable to other works that use real system power measurements for control. For example, the sampling interval in an IBM processor controller is 1 s [10]. The power-sensing latency can be drastically lower when the sensor is directly located in the processor chip instead of on the computer board.

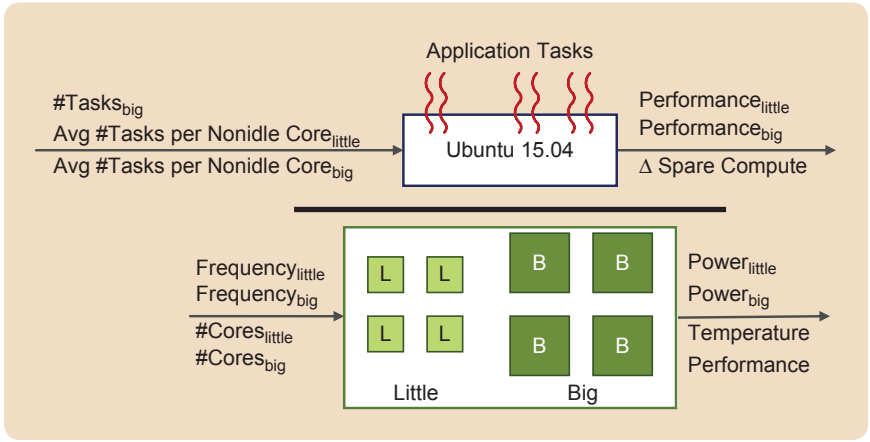

FIGURE 2 A schematic of the Odroid-XU3 showing the hardware and operating system layers, with the input and output signals considered. The hardware layer includes the processor, which is made of eight units, or cores. Four of the cores are high-performance, high-power units (called *big cores*) that are organized as one cluster, while the others are low-performance, low-power units (known as *little cores*) that form another cluster. There can be many application tasks running simultaneously. The outputs in one layer are influenced by the behavior of the other layer. Avg: average.

Four inputs in the hardware layer are considered: the operating speed of all of the cores in the little cluster (Frequency$_{little}$, measured in gigahertz), the operating speed of all of the cores in the big cluster (Frequency$_{big}$, measured in gigahertz), the number of little cores that are powered on (#Cores$_{little}$), and the number of big cores that are powered on (#Cores$_{big}$). Changing the frequency of each cluster takes fewer than 10 ms, while turning a core on/off takes twice as long. The available discrete values of each input are given in Table 1.

We consider three controlled outputs in the OS layer: the throughput of the little-cluster tasks (Performance$_{little}$, measured in BIPS), the throughput of the big-cluster tasks (Performance$_{big}$,

measured in BIPS), and the difference in spare compute capacity (ΔSpare compute) between the big and little clusters. We define a cluster's spare compute in (1), from [17]:

$$\text{spare compute} = \#\text{idle cores on} - (\#\text{tasks} - \#\text{cores on}). \quad (1)$$

The term (#tasks − #cores on) measures how many more tasks have been assigned to a cluster than the number of available cores. The spare compute measures how much of this difference can be accommodated within the idle cores of the cluster. A large value for the spare compute indicates that the cluster can accept more tasks, while low and negative values indicate an overloaded cluster. The output ΔSpare compute is the difference between the spare compute in the big and little clusters. Intuitively, the higher the difference in the spare compute, the more tasks the controller will move from the little to the big cluster.

The process scheduler in the OS assigns the application tasks to cores. Ignoring the differences between tasks, the scheduler must 1) decide how to divide the tasks between the big and little clusters and 2) map tasks to cores in each cluster. Some cores can be left idle without any tasks so that the hardware controller can power them down. Therefore, there are three inputs in this layer: 1) the number of tasks assigned to the big cluster ($\#\text{Tasks}_{\text{big}}$), leaving the rest for the little cluster; 2) the average number of tasks running on each nonidle little core (avg #tasks per nonidle $\text{core}_{\text{little}}$); and 3) the average number of tasks running on each nonidle big core (avg #tasks per nonidle core_{big}). Changing each OS input has nearly the same overhead (a few milliseconds) because it involves moving a task from one core to another.

The available discrete values of each input are given in Table 2. In it, $\#\text{Tasks}_{\text{max}}$ is the number of tasks in the application, which is decided by the application and changes dynamically. The minimum value of the inputs, avg #tasks per nonidle $\text{core}_{\text{little}}$ and avg #tasks per nonidle core_{big}, is zero when the tasks assigned to the respective clusters are zero; otherwise, it is one. The maximum value of these inputs occurs when all of the tasks assigned to those clusters are assigned to a single core. There is a nonlinear dependence between the inputs because the values accepted by avg #tasks per nonidle $\text{core}_{\text{little}}$ and avg #tasks per nonidle core_{big} depend on the value of the first input ($\#\text{Tasks}_{\text{big}}$).

INTERACTION BETWEEN LAYERS AND THE VARIABILITY OF APPLICATIONS

The prototype demonstrates how one system layer influences the other and how applications exhibit large variations in their behavior. Consider two applications from the PARSEC benchmark suite [20], *blackscholes* and *vips*, that generate a variable number of tasks through time and run on our prototype. We monitor the hardware-layer outputs and fix the hardware inputs to remain at the lowest values.

TABLE 1 The available values for the hardware-layer inputs.

Input	Range	Step Size
$\text{Frequency}_{\text{little}}$	0.2–1.4 GHz	0.1 GHz
$\text{Frequency}_{\text{big}}$	0.2–2 GHz	0.1 GHz
$\#\text{Cores}_{\text{little}}$	1–4	1
$\#\text{Cores}_{\text{big}}$	1–4	1

TABLE 2 The available values for the operating system–layer inputs.

Input	Range	Step Size
$\#\text{Tasks}_{\text{big}}$	$0–\#\text{Tasks}_{\text{max}}$	1
Avg #tasks per nonidle $\text{core}_{\text{little}}$	$\min(1,\#\text{Tasks}_{\text{little}})–\#\text{Tasks}_{\text{little}}$	1
Avg #tasks per nonidle core_{big}	$\min(1,\#\text{Tasks}_{\text{big}})–\#\text{Tasks}_{\text{big}}$	1

Avg: average.

Each application is run twice, and a different policy is used to change the OS inputs each time. In the first policy, the OS assigns an equal number of tasks to each cluster, and within each cluster, tasks are distributed on as many cores as possible. In the second policy, the OS randomly sets its inputs.

Figure 3 shows how two of the hardware outputs, $\text{Power}_{\text{big}}$ and Performance, vary over time for *blackscholes* with the two OS policies. The *blackscholes* application starts with one task and, after some time, abruptly launches eight more. When these tasks complete, the first task runs for a while before the application concludes. With the first OS policy [Figure 3(a)], three phases (corresponding to one task, eight new tasks, and one task) can be easily identified. Except for abrupt changes at transitions, there is little variation within each phase. However, with the second OS policy [Figure 3(b)], there is intense variation in all phases, even though the hardware inputs are constant. The variation in the phase with eight new tasks is particularly large due to the impact of the OS-layer inputs. The figure shows that the hardware-layer outputs are significantly affected by the OS inputs.

Figure 4 relays the behavior of *vips*. In this application, the number of tasks varies dynamically between one and eight, and some of the tasks may not use the processor, even when scheduled. Consequently, even with the first OS policy [Figure 4(a)], the hardware outputs display much variation. Figures 3 and 4 show that the behavior of a layer is significantly affected by the decisions from other layers. As a result, modularly designed controllers in each layer must be robust against the influence from other

layers. Comparing Figures 3(a) and 4(a) illustrates how two applications with the same hardware inputs (and the same policy to set the OS inputs) can present drastically different behavior. A computer system must work well for many such applications, further strengthening the need for robustness.

GOALS FOR COMPUTER-RESOURCE CONTROL

The objective of computer efficiency typically involves two types of goals: tracking a set of output targets and optimizing a combination of the measured outputs [12]. In applications such as video processing engines, it is necessary to deliver a constant level of power consumption and throughput. An advanced scenario occurs when the desired output levels and targets are changed dynamically based on real-time conditions. For example, in a battery-powered mobile device, the desired throughput (or quality of service) and power consumption change as the battery energy is depleted [22]. When the battery charge is above a certain level, the system must deliver a high throughput and can have a tolerably high power consumption. As the battery level decreases, the system can choose lower pairs of throughput and power to conserve battery life.

Optimizing a combination of outputs, which is the second type of goal, is more prevalent in computer control. Common optimization goals seek to minimize objectives of the form Energy $\times$ Delay^n (ED^n), subject to constraints. ED^n is the product of the application's entire energy consumption and the total execution time (Delay) raised by a factor n. A larger n prioritizes higher performance over lowering the energy consumption. Usually, the constraints are in terms of temperature, power consumption, and utilization (see [23], for example). Similar to the tracking goal, the metric to be optimized can change through time.

In a multilayer computer, controllers in each layer must achieve tracking and optimization goals using only

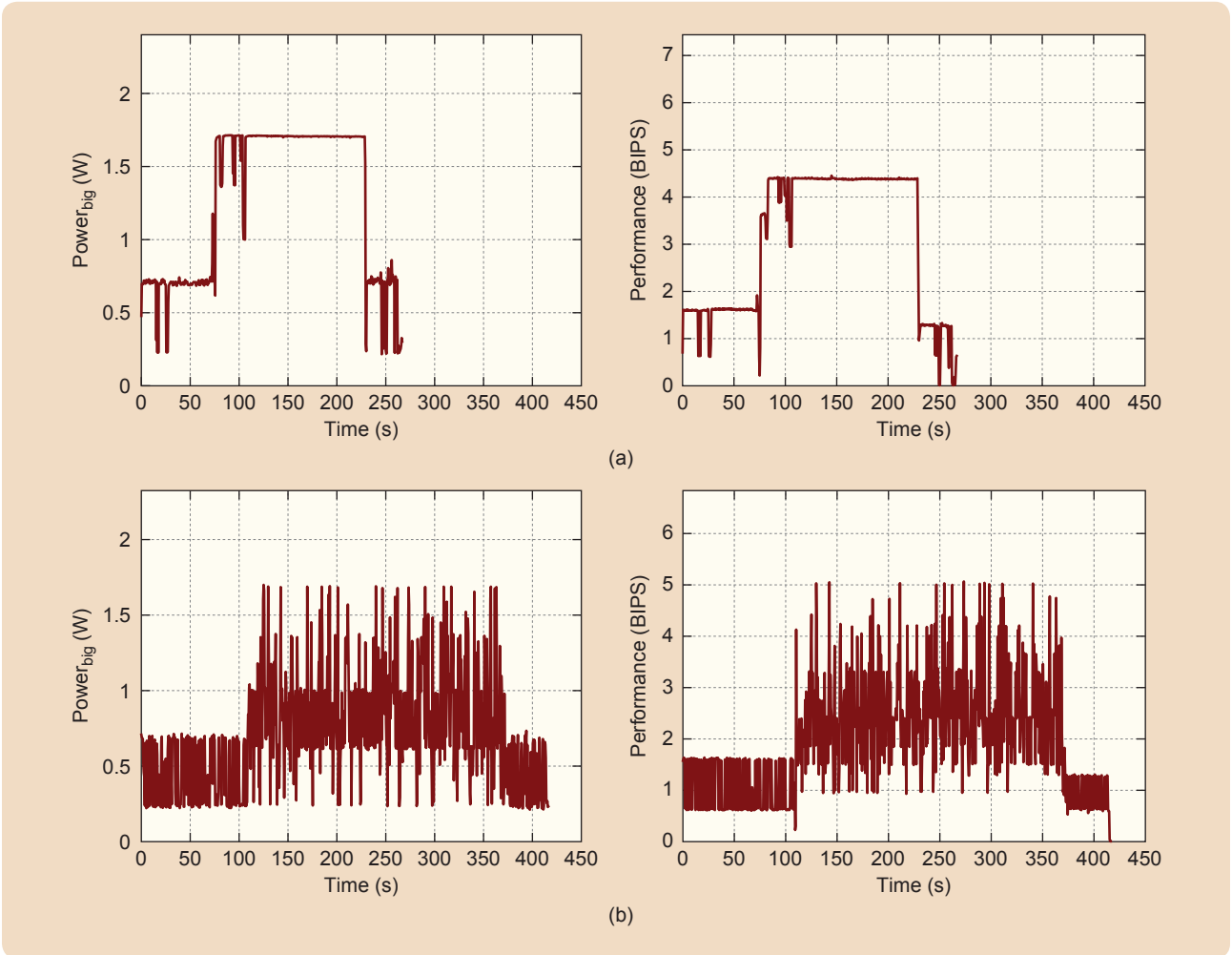

FIGURE 3 A demonstration of how the hardware-layer outputs (the power of the big cluster and performance) vary with different policies in the operating system (OS) layer. In both cases, the same application (*blackscholes*) is run, and all of the hardware inputs are held constant. Only the OS inputs are changed through time. (a) In the first OS policy, the OS distributes threads equally among all available cores in the system, while (b) the second OS policy sets the OS layer inputs randomly. BIPS: billions of instructions per second.

the sensors and actuators available to them. For example, a hardware controller may only change the processor frequency based on throughput and power, while the OS controller handles scheduling based on utilization. When they are optimizing the combined measures, such as ED^n, the resource controllers can know only the total energy and application duration after the application completes its execution. Therefore, they use other derived metrics for dynamic control during the execution. For example,

$$\text{Energy} = \text{Power} \times \text{Delay}, \tag{2}$$

$$\text{Delay} = \frac{\text{Instructions}}{\text{Throughput}}, \tag{3}$$

where Power is the average power, Instructions is the total number of instructions in the application, and Throughput is the average rate at which the instructions are processed by the computer. As the total number of instructions is fixed,

$$\mathrm{ED}^n \propto \frac{\text{Power}}{\text{Throughput}^{n+1}}. \tag{4}$$

At the kth controller invocation, Power$[k]$ and Throughput$[k]$ can be defined as

$$\text{Power}[k] = \frac{\text{Energy}(k) - \text{Energy}(k-1)}{T}, \tag{5}$$

$$\text{Throughput}[k] = \frac{\text{Instructions}(k) - \text{Instructions}(k-1)}{T}, \tag{6}$$

where Energy(k) and Instructions(k) are the values of the energy consumed and instructions processed by the system from the beginning of the application until the instant k. The instantaneous values, Power$[k]$ and Throughput$[k]$, are used to calculate $\mathrm{ED}^n[k]$. This value must be minimized

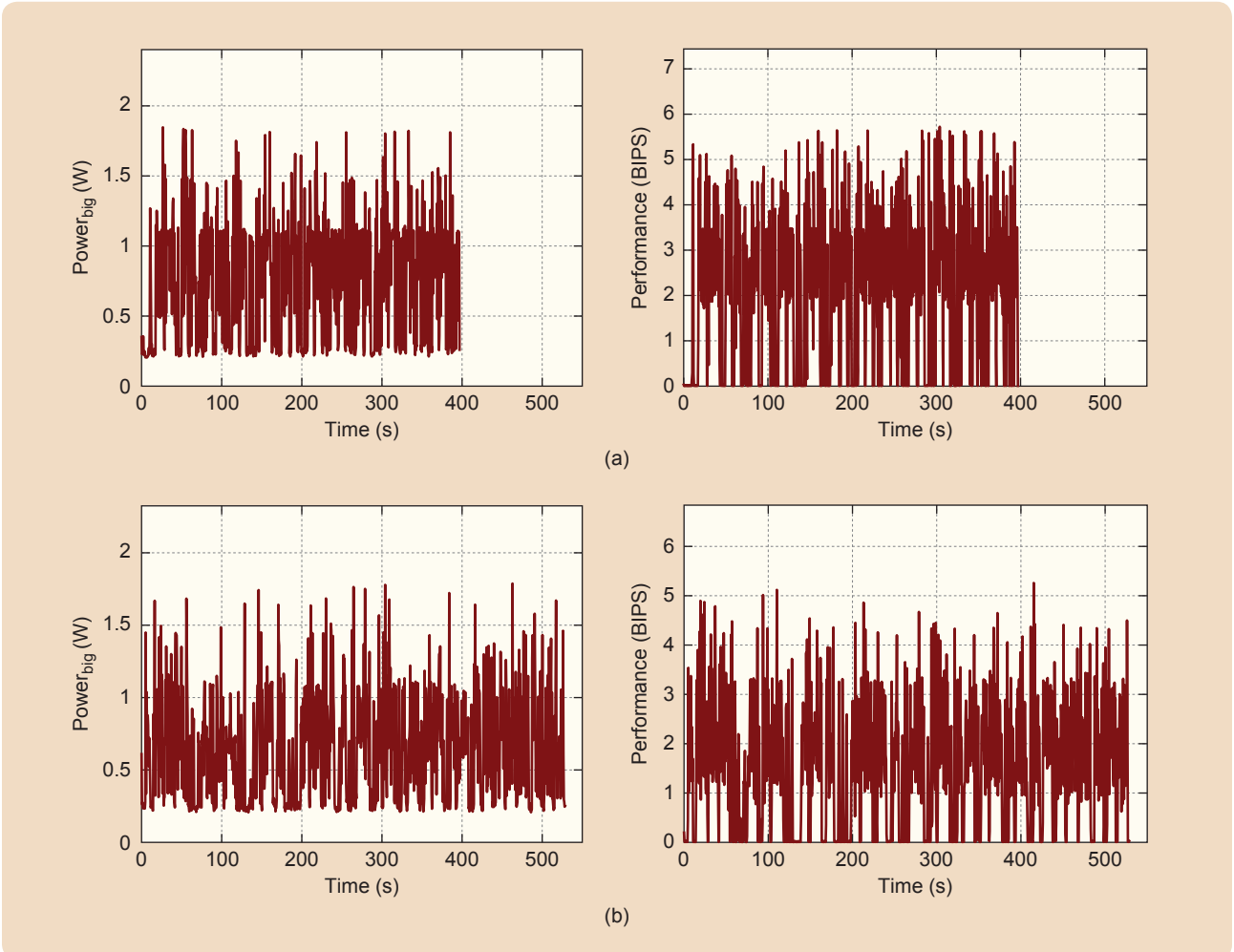

FIGURE 4 A demonstration of how the hardware-layer outputs (the power of the big cluster and performance) vary with different policies in the operating system (OS) layer. In both cases, the same application (*vips*) is run, and all of the hardware inputs are held constant. Only the OS inputs are changed through time. (a) In the first OS policy, the OS distributes threads equally among all available cores in the system, while (b) the second OS policy sets the OS layer inputs randomly.

by the resource controllers during successive invocations. A conventional practice is to maximize the inverse (Throughput$^{n+1}[k]$/Power$[k]$).

CONTROL OBJECTIVES FOR THE PROTOTYPE

The goal for the prototype computer is to develop a control system that achieves both tracking and optimization goals. For optimization, the goal is to minimize Energy × Delay under constraints. Therefore, the controller in each layer dynamically maximizes the same metric, Throughput^2/Power. This optimization has soft constraints to preserve the physical integrity of the system, including maintaining the power of the big and little clusters and holding the temperature below certain limits. The hardware controller's goal can be stated as

$$\begin{aligned} \underset{u_{\mathrm{HW}} \in U_{\mathrm{HW}}}{\text{maximize}} \ & \frac{\text{Throughput}^2}{\text{Power}} \\ \text{Power}_{\text{little}} &< \text{Power}_{\text{little}}^{\text{limit}}, \\ \text{Power}_{\text{big}} &< \text{Power}_{\text{big}}^{\text{limit}}, \\ \text{Temperature} &< \text{Temperature}^{\text{limit}}, \end{aligned} \tag{7}$$

where U_{HW} is the set of values that the inputs in the hardware can accept. These inputs are listed in the "Hardware- and Operating System Layer Inputs and Outputs" section. For our board, the maximum powers of the little cluster ($\text{Power}_{\text{little}}^{\text{limit}}$) and big cluster ($\text{Power}_{\text{big}}^{\text{limit}}$) and the limiting temperature ($\text{Temperature}^{\text{limit}}$) are 0.33 W, 3.3 W, and 79 °C, respectively.

The goal for the OS controller is

$$\underset{u_{\mathrm{OS}} \in U_{\mathrm{OS}}}{\text{maximize}} \ \frac{\text{Throughput}^2}{\text{Power}}, \tag{8}$$

where U_{OS} is the set of values accepted by the inputs in the OS layer, as listed in the "Hardware- and Operating System Layer Inputs and Outputs" section. The controllers must meet these goals by using imperfect models of their layers and any available interlayer interface information.

RESOURCE-CONTROL DESIGN REQUIREMENTS

Computer systems such as our prototype are difficult environments to control. We list some important features that the control system design should include to be effective.

- *Modularity*: Because the layers are developed separately, each one requires a modular controller that can be independently designed. A controller must use an interface to communicate with its counterparts in other layers and coordinate for global efficiency.
- *Robustness against modeling limitations*: The layers are too complex to model from first principles. Therefore, it is infeasible to obtain accurate models of a layer or interlayer interaction except in limited contexts (see [13] for examples). Consequently, robustness is paramount when designing the controllers.
- *Managing application variability*: The applications that run on computers are numerous and vary their behavior through time. Such variation can be inherent in the application itself. For example, an application may run heavy computations for some time (generating a high throughput and using significant power) and transition suddenly into an idle period that has a low throughput and consumes little power. In another case, the number of tasks used within the application can quickly increase, elevating the power use, temperature, and throughput. The control system must detect changes in application behavior and find the best operating points for the outputs and suitable values of the inputs during each phase. The controllers must also be robust against unanticipated changes in the application behavior.
- *Handling nonlinearities in the inputs and objectives*: The controllers must handle tracking and optimization goals involving metrics that can have a nonconvex and nonlinear relationship with the inputs. The inputs are also finite and discrete valued. Controllers must be robust against such actuator nonlinearities.
- *Intuitive design and tool support*: There is a significant gap between the tools, knowledge, and perspectives of computer designers and those in the theoretical research of disciplines such as optimization or control theory. For the mainstream adoption of new systematic methods, it is necessary to develop computer-centric abstractions and design methodologies. The design methodologies should be supported mostly by standard tools with intuitive tuning processes.

PRIOR WORK ON CONTROLLING COMPUTERS

Many works in research and industry use heuristics to control computers (for example, [8], [24]–[26]). Prior research demonstrates the design and runtime limitations of heuristic designs [10]–[12], [27]–[30]. Several studies use control-theoretic designs. Among these, most use single-input, single-output (SISO), proportional-integral-derivative (PID) controllers [14], [15], [23], [31]–[33], but these designs are too limited to control even a single layer that has many goals. Some researchers proposed using collections of SISO controllers in a layer to tackle multiple goals [27], [34]. Computer layers have many coupled outputs (such as power and throughput) strongly influenced by multiple inputs [known as multiple-input, multiple-output (MIMO)] systems [12], [28]. Decoupled SISO controllers cannot manage the interaction between these outputs, resulting in poor performance [12], [28]. Some designs employ heuristics to manage controller interaction [34]. However, this defeats the purpose of using control-theoretic methods. Some works use a multiple-input, single-output approach with model predictive controllers (MPCs) [35], [36]. However, these are also insufficient to handle the MIMO nature of computer layers.

Prior work includes MIMO designs with linear quadratic (LQ) controllers or MPCs [12]–[14], [28]. Since each computer layer is a MIMO system, MIMO control is best suited to computers. However, the proposed designs are intended for centralized use and do not prioritize robustness against the uncertainty in modularly designed multicontroller environments. The successful application of control theory for computers has also been limited to specific contexts where accurate models could be developed [13], [14]. Such methods do not directly apply to the general problem of modular-computer resource management.

Some control designs use machine learning [37]–[40] and matrix-completion methods for resource management [41]. Mishra et al. [42] employ an adaptive SISO PID controller that generates a generic speedup signal and a machine-learning-based algorithm that searches for the best input configuration by using the signal. When machine-learning-based control algorithms find different output conditions at runtime than those that they were trained on, they provide a lower-quality solution unless they complete an expensive retraining phase. The other difference is in the guarantees they provide. Usually, guarantees from machine learning are valid for the average case without considering uncertainty.

Some works formulate the ED^n minimization as a convex optimization problem settled by using linear programming solvers [43]–[45]. Solver-based approaches require more time to generate a decision. Since the system is dynamic, this expensive process must be repeated frequently. Finally, some designs use market theory [46], [47] and game theory [48] to manage resources in specific contexts. However, there is no work that addresses the design of modular coordinated controllers for multilayer computers.

CHOOSING AN APPROACH FOR MODULAR COORDINATED CONTROL

Prior work used several approaches to achieve tracking and optimization goals in selected computing scenarios [25], [26], [29], [36], [37], [45], [49]–[52]. We analyze these methods to determine the approach to achieve computer-resource management goals. We begin by classifying the existing work into five domains: static optimization, machine learning, control theory, model-based heuristics, and rule-based heuristics. Table 3 compares these approaches, outlining their problem formulation, design and tuning method, advantages, and shortcomings.

Static optimization is a common choice in several designs. One limitation of this approach is the lack of accurate models and convex objective functions. Another shortcoming is that it ignores the dynamic nature of computer systems. Therefore, controllers must update their models periodically at runtime. Lastly, the solvers used by most optimization algorithms can have high overheads in time and computation requirements. Table 3 does not mention approaches that use game [48] and market theory [46], [53], which have advantages and limitations similar to those of the optimization-based methods.

Machine learning is another approach to obtain the best configuration for optimization goals. In this case, the controller can be a neural network trained on several benchmarks to associate some preferred input choices with a set

TABLE 3 Comparing the approaches for architecture tuning.

Approach	Formulation	Example	Advantages	Shortcomings
Static optimization [45], [49]	$\min_{\text{inputs}} J(\text{outputs})$ with constraints, outputs = f(inputs)	$\min_{\text{frequency}} \frac{\text{Throughput}^2}{\text{Power}}$, Temperature $< T_o$	+ Natural choice for systems optimization + Formal approach	– Accurate and convex models required – No feedback – Slow solvers a possibility
Machine learning [37], [50]	inputs = f(outputs) Optionally update f() f() can be a neural network that minimizes J(outputs)	Input: frequency Outputs: power, memory accesses	+ Data driven + Formal approach	– Hard to add feedback – Only probabilistic guarantees – Needs extensive training
Control theory [29], [36]	inputs = G Δoutputs G: $[A\ B\ C\ D]_{LTI}$ that minimizes Δoutputs = outputs – targets	Input: frequency Outputs: power, throughput. Targets: $power_o$, quality of service	+ Gives worst-case guarantees + Learns from feedback + Formal approach	– Hard to obtain model – Specifying targets not obvious – Suits mostly tracking goals
Model-based heuristics [51], [52]	Uses a model to guide heuristic decisions The model relates outputs, inputs, and auxiliary outputs	Output: power Auxiliary output: memory accesses Input: frequency	+ Decision making simplified by model + Easy to develop for simple systems	– No guarantees – Ad hoc design – Prone to errors – Hard to add learning – Too difficult to design for complex systems
Rule-based heuristics [25], [26]	Algorithmic decisions to choose inputs based on outputs and auxiliary outputs		+ Easy to develop for simple systems	

of observed outputs. This training updates the weights of the neural network so the network can later predict good input configurations with runtime data. It is possible to dynamically modify the neural network through reinforcement-learning methods, although this incurs a higher implementation complexity. The data-driven approach for machine-learning methods is attractive for computer control because it overcomes the lack of models. The downside is that the guarantees of convergence and optimality are probabilistic and not necessarily valid for worst-case scenarios. Since computer applications are diverse, it is possible to encounter unexpected application patterns. Moreover, training controllers with supervised learning usually requires a large number of application runs and extensive data processing.

Control theory natively addresses dynamic systems; uses feedback to improve control; and provides guarantees for stability, convergence, optimality, and robustness. Despite control theory's advantages, at least two issues have limited its use in systems. First, it requires a dynamic model, which 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 targeting a throughput requirement. However, that is not so when trying to optimize metrics, including ED^n. It is also not straightforward to specify goals, such as "be less/more than this value." It is easier to specify objectives such as "attain this value."

Finally, heuristics-based approaches (with or without a model) are responsible for the majority of decision making in computers. While they are easy to develop and understand for simple systems, difficulties arise with complex structures. There are high overheads to design, tune, and verify the ad hoc algorithms. Despite this, heuristics have often failed to perform well after deployment.

Among these approaches, we select control theory because it can address the dynamic nature of computers. Next, we describe the selections from the choices available in control theory to build the design. Then, we introduce the control system that includes an optimizer module to provide suitable targets for the controllers, addressing the limitation of a control theory-based approach.

TABLE 4 The design choices from control theory. Italic text indicates the choices that were selected.

Category	Choices
Modeling	White box (analytical), *black box (data driven)*, gray box
Mode	Single-input, single-output; multiple-input, single-output; single-input, multiple-output; *multiple-input, multiple-output*
Organization	Decoupled, centralized, cascaded, *collaborative*
Approach	Classical, *robust*, gain-scheduling, adaptive
Type	Proportional-integral-derivative; linear quadratic Gaussian; model predictive controller, H_∞ or μ *synthesis*

NAVIGATING THE CONTROL THEORY CHOICES

Table 4 presents the taxonomy of design choices from control theory. First, the model of the system can be obtained by using analytical principles (white box), experimental data (black box), or a combination of both (gray box). Black-box models are the best choice when the system internals are unknown or too complicated to describe, as in computers (see "Black-Box System Identification" for an overview). Next, among the different modes of control, we use MIMO controllers because (as previously indicated) we target multiple tightly coupled goals that depend on numerous inputs.

The different organizations of MIMO controllers for multilayer systems are then considered. Decoupled and centralized designs cannot achieve modularity and coordination simultaneously. In a cascaded design [6], controllers are organized as a nested loop, where each one sets the targets for the immediately inner one. Only the innermost controller changes the system inputs. This method, too, is not well suited for our goal. The multilayer controllers do not always have a hierarchy between them and must be designed independently. To respond to dynamic conditions, the controllers must wait until their sensor data propagate to higher controllers and new targets cascade down. This is a long latency process. Instead, we identify the best choice to be a collaborative architecture, where independent controllers communicate to attain coordination.

Several approaches are used to ensure that the controller works correctly during uncertainty and through rapidly changing conditions. The classical one is to design controllers with additional stability margins [33]. This works for simple systems. Robust control explicitly optimizes controllers for significant uncertainty and is applicable to computer environments. The controllers have a low complexity and modest overheads. In gain scheduling, multiple controllers are used, each suited for a particular type of execution [54]. At runtime, logic determines when each one is active, based on the execution. This approach requires additional modeling efforts and may necessitate expensive selection logic at runtime. Lastly, adaptive control synthesizes a new controller online whenever changing conditions are detected [55]. It has a higher runtime overhead and design complexity.

Finally, for the controller type, PID controllers are commonly used for their simplicity. However, they are not adequate to control MIMO systems. For MIMO systems, LQ Gaussian (LQG) controllers [12] and MPCs [28] have been proposed. However, with modular-controller design, the interaction between layers can manifest as a large uncertainty

that can degrade the behavior of LQG control and MPC. Moreover, these controllers do not have channels to communicate among themselves.

The properties of μ controllers make them attractive to meet the needs of computer management.

- μ controllers can be designed to natively work with uncertainty. We can formulate model limitations, interlayer interaction, and input discretization as uncertainty. Independent teams use models of only their layers to design controllers that are robust to this uncertainty. This approach enables a modular and robust design.
- μ controllers can take disturbance measurements for improved control. In computer systems, these are called *external signals* and used to pass information from one layer to the controller of another layer. For example, the OS controller passes the number of currently running tasks as an external signal to the hardware controller.
- μ synthesis is well supported by standard tools [57]. These controllers do not require online solvers to compute decisions (unlike MPCs, for example), which is essential for fast decision making.

MODULAR COORDINATED CONTROL WITH ROBUST CONTROL THEORY

The design was developed using μ synthesis from robust control theory [58]. Black-box system identification [59] was employed for the modeling, since it is difficult to model computer systems using first principles. The μ controllers can handle model limitations and unknown interlayer interaction. Finally, additional optimizer modules were incorporated with each μ controller to meet the optimization goals.

Control System Architecture

Figure 5 diagrams the architecture of the proposed multilayer control system that achieves the goals in (7) and (8). There is a robust μ controller and an optimizer in each layer. The μ controller is used to keep the outputs close to the received targets. This accomplishes the basic tracking goal. For optimizing a metric, the optimizer issues various targets that progressively improve the metric. The μ controller responds by changing the inputs to make the outputs follow the changing targets. The μ controllers also read the inputs in other layers as measured disturbances (external signals) for improved decisions. This is a modular architecture in which the controller and optimizer in a layer can be

Black-Box System Identification

System identification is an area associated with control theory that concerns the modeling of systems by using empirical data. Because real-world systems include many phenomena that are difficult to model from first principles, system identification plays an important role in capturing the relationship between the systems' inputs and outputs.

Modeling can be classified into three types. The first is white-box modeling, where the system dynamics are described from physical equations. Aerospace control has many models that are obtained in this manner. The second is gray-box modeling, where designers specify a generic model structure (for example, a polynomial model) based on their insight, and the model parameters are derived from experimental information. This is the prevalent approach in many domains. Finally, black-box modeling assumes no prior knowledge of the model structure. Therefore, a suitable model structure and its parameters are derived from experimental data. This approach is particularly relevant for computer systems, as we do not have a priori knowledge of their dynamics.

In system identification, the system (or a computer running applications, in this case) is excited with special input signals. They are designed to easily bring out the dependence of outputs on the inputs. Popular choices include the pseudorandom binary sequence, random Gaussian, and sum-of-sinusoids methods. Choosing the input signals for multiple-input, multiple-output (MIMO) systems requires more attention. The input signals are chosen to minimize their cross correlation. One approach is to excite one input channel with a signal during each run and record the data. Another is to use uncorrelated random signals for each channel simultaneously. Therefore, MIMO systems may require multiple identification runs. In each identification run, the input–output data are collected through time.

After the identification runs are complete, there are multiple logs of input–output data through time, one log for each experimental run. These data are usually normalized so that the different inputs and outputs have similar ranges of values. With the normalized data, linear-regression methods are commonly used to identify the parameters of a model structure. With black-box identification, designers may have to experiment with different model structures and dimensions to find the one that explains their data well and aligns with any insights.

For a given model structure, using larger dimensions (for example, using a longer history with polynomial models) results in overfitting the model to the training data. As a result, the obtained model parameter values are said to have a low confidence, and the variance of the values given by the model is high. On the other hand, if the model dimension is too small, the model cannot explain all of the non-noise dynamics, resulting in larger systematic errors in the values it gives. The model is said to have a high bias. Tools such as Matlab [56] make it easy to choose between different model structures and the right dimensions for each model based on the training data. See [59] for details on the different steps in system identification.

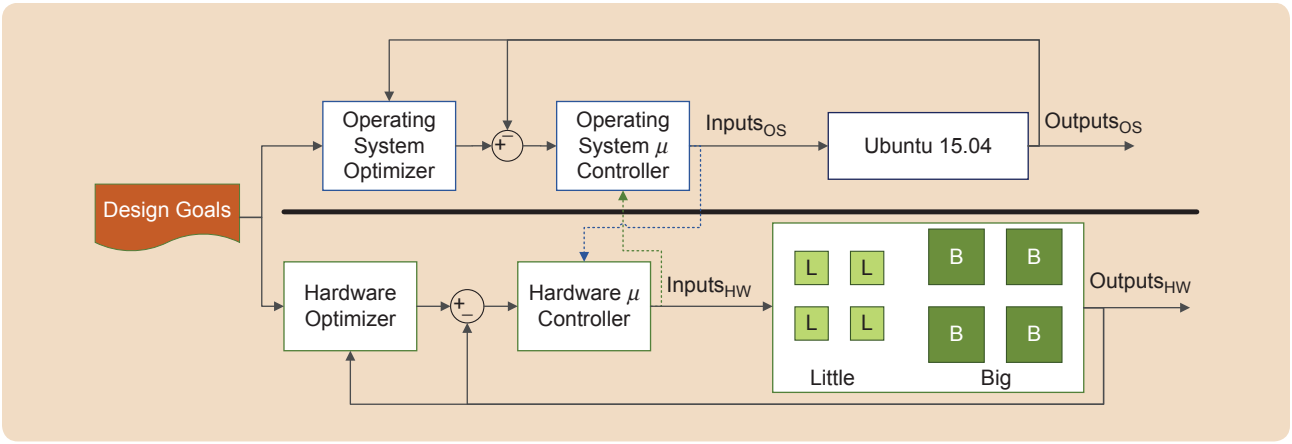

FIGURE 5 The multilayer μ-control architecture. Each layer uses a cascaded control system to meet the optimization and tracking goals. The optimizer searches for the targets that best optimize the execution under constraints. The μ controller tracks the given targets robustly and optimally according to a conventional cost function. The μ controllers receive inputs from the other layer as external signals.

designed independently. The design teams must exchange only the interface information for sharing the external signals and bounds for commonly monitored outputs.

The hardware optimizer generates the output targets that optimize the objective under constraints. This function can be stated as

$$\begin{aligned} &\max_{y_{HW_o}} \frac{\text{Throughput}^2}{\text{Power}} \\ &\text{Power}_{\text{little}} < \text{Power}_{\text{little}}^{\text{limit}}, \\ &\text{Power}_{\text{big}} < \text{Power}_{\text{big}}^{\text{limit}}, \\ &\text{Temperature} < \text{Temperature}^{\text{limit}}, \end{aligned} \tag{9}$$

where y_{HW_o} represents the target values for the outputs. This search for the best output targets is implemented using simple algorithms. For example, to optimize $\text{Throughput}^2[k]/\text{Power}[k]$, the search module progressively increases the performance targets by larger amounts than it increases the power targets, or it decreases the performance targets by smaller amounts than it decreases the power targets. This implementation requires low overheads.

The controller must robustly track the targets given by the optimizer. This goal is formulated as

$$\begin{aligned} &\min \| \Gamma_{zw} \|_\infty \\ &w : \begin{pmatrix} y_{\text{HW}^*} \\ d \end{pmatrix} \xrightarrow{\Gamma_{zw}} z : \begin{pmatrix} W_p (y_{\text{HW}^*} - y_{\text{HW}}) \\ W_u u_{\text{HW}} \end{pmatrix}, \end{aligned} \tag{10}$$

where u_{HW} represents the hardware inputs, y_{HW^*} denotes the targets for the hardware outputs y_{HW}, $\| \Gamma_{zw} \|_\infty$ is the induced H_∞ norm that captures the robustness to bounded disturbance d, W_p is the performance-requirement weight, and W_u is the weight for the inputs. This is a mixed-sensitivity, robust-control problem that ensures optimal tracking according to the designer requirements during uncertainty. Standard μ synthesis is used to achieve this subgoal. The formulations for the OS optimizer and μ controller are similar. For tracking-only objectives (see the "Goals for Computer-Resource Control" section), the optimizer simply issues the output targets that the designer/user needs. Even for advanced tracking where the targets change through time, the optimizer uses preset functions to generate the targets.

The principle behind the proposed architecture is to use the same control system for tracking and optimization. Furthermore, the optimizer searches the output-target space, which is simpler than searching the input space. For example, when optimizing $\text{Throughput}^2[k]/\text{Power}[k]$, the optimizer can progressively increase the performance targets by a larger amount than for the power targets or decrease the performance targets by a smaller amount than for the power targets. This results in a simple implementation. The alternative method of searching the input space has a higher complexity because the exact magnitude of an input's impact on each output is difficult to ascertain. This difficulty increases when there are many inputs and outputs and when other layers change the dynamics at runtime. Since it is important for the control system to consume as few resources as possible for decision making, this approach is not used.

Additionally, the optimizer does not have to explicitly address the uncertainty; instead, it relies on the μ controller. The μ controller ensures that the output tracking is effective under the uncertainty of unmodeled intra- and interlayer effects. Since the μ controller can quickly track the targets, the optimizer does not have to wait long to issue new ones. Consequently, the overall optimization is fast.

Black-Box System Identification and Models

Due to system complexity, there are no accurate dynamic models of computers built with first principles. Therefore, empirical black-box system identification [59] is the best

approach to model them. See "Black-Box System Identification" for a brief overview. Our identification experiments use two applications (*swaptions* and *vips*) from the PARSEC 2.1 application suite [20] and four applications (*astar*, *perlbench*, *milc*, and *namd*) from SPEC06 [21]. These suites are standard for evaluating the performance of computers. From them, applications for identification are randomly selected.

Two tests are used, and there are pseudorandom input sequences for each training benchmark. The value of each input is chosen randomly and remains unchanged for an arbitrary duration lasting between 1 and 16 sampling intervals. The data are normalized to a $[-1,1]$ range, and a polynomial model is identified. Figure 6 gives the input and output values during the identification experiments for swaptions. The figure shows how the four hardware inputs are changed and the three external signals received from the OS are set. Figure 7 presents the identification data following the normalization. The normalized data are used to obtain our models.

It is determined that a Box–Jenkins polynomial structure generates models that better fit our data compared to other polynomial, state-space, and transfer-function models. These models connect the outputs $y[k]$ at a discrete interval k, with the inputs $u[k]$ and noise $e[k]$ as

$$y[k] = \left(\frac{B}{F}u\right)[k] + \left(\frac{C}{D}e\right)[k]. \tag{11}$$

Note that B, C, D, and F are polynomial transfer functions in the z transform domain. The suitability of the Box–Jenkins model structure aligns with our assumption that the output values are related to prior values.

The hardware model has four inputs, three disturbance inputs, and four outputs; the OS model has three inputs, four disturbance inputs, and three outputs. The identified models can be represented as second-order transfer functions from inputs to outputs. For example, in the hardware model, the transfer function from $\text{Frequency}_{\text{little}}$ to $\text{Power}_{\text{little}}$ is given by $(0.38z^2/z^2 + 0.04z + 0.02)$. Similarly, the transfer function from $\text{Frequency}_{\text{big}}$ to $\text{Power}_{\text{big}}$ is given by $(0.34z^2/z^2 - 0.01z + 0.02)$, and the transfer function from $\text{Frequency}_{\text{big}}$ to Temperature is given by $(0.18z^2/z^2 - 0.21z + 0.12)$. The other transfer functions are similar.

The models are nominally stable as is the underlying system. Figure 8 shows the Bode magnitude plot for the

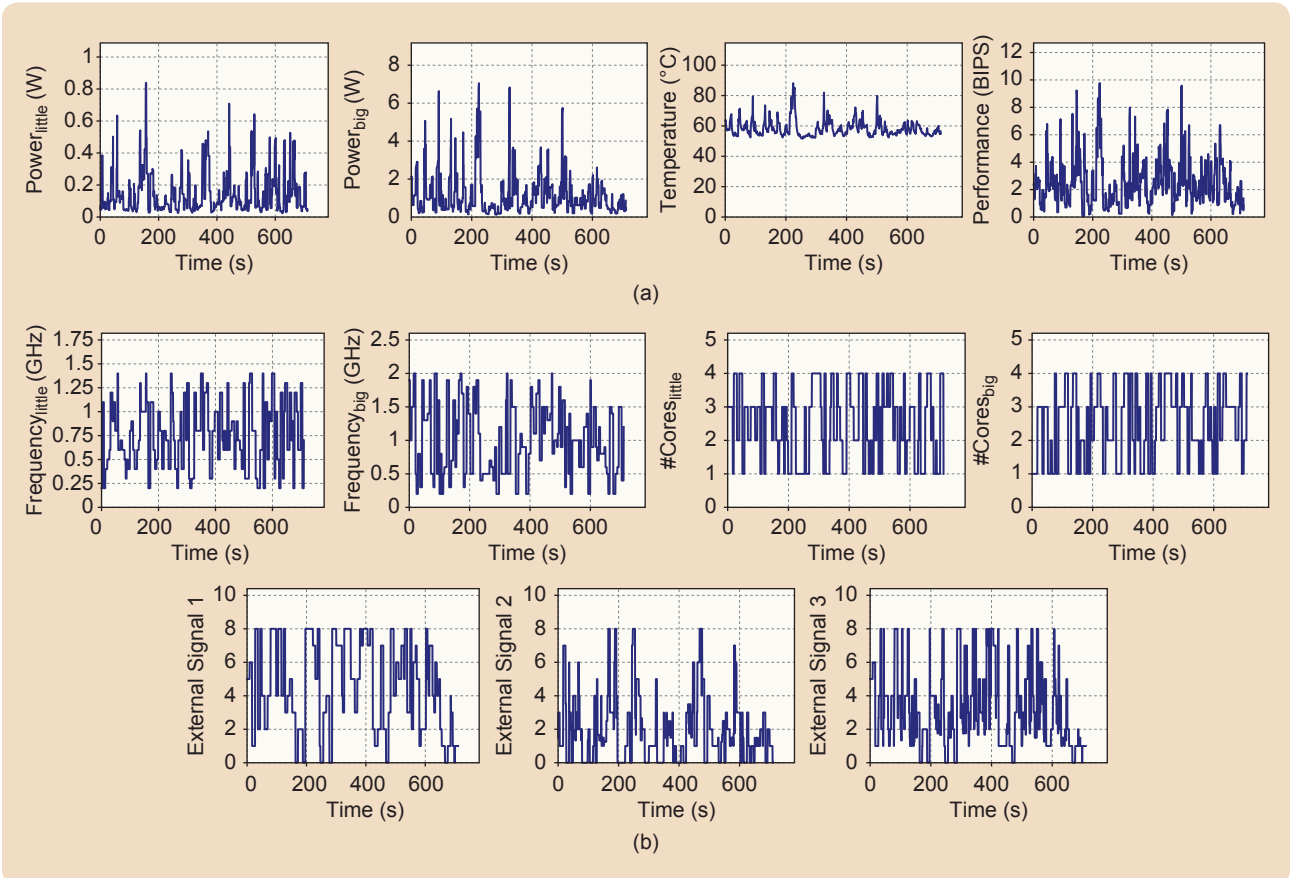

FIGURE 6 The identification data with the PARSEC benchmark swaptions. (a) Observed outputs of swaptions during system identification. The top row shows how each hardware output changes through time in response to the identification inputs. (b) Inputs used during system identification with swaptions. The bottom row of plots shows how we change the four hardware inputs (u_{HW1}–u_{HW4}) and the external signals received from the operating system (es_1–es_3). BIPS: billions of instructions per second.

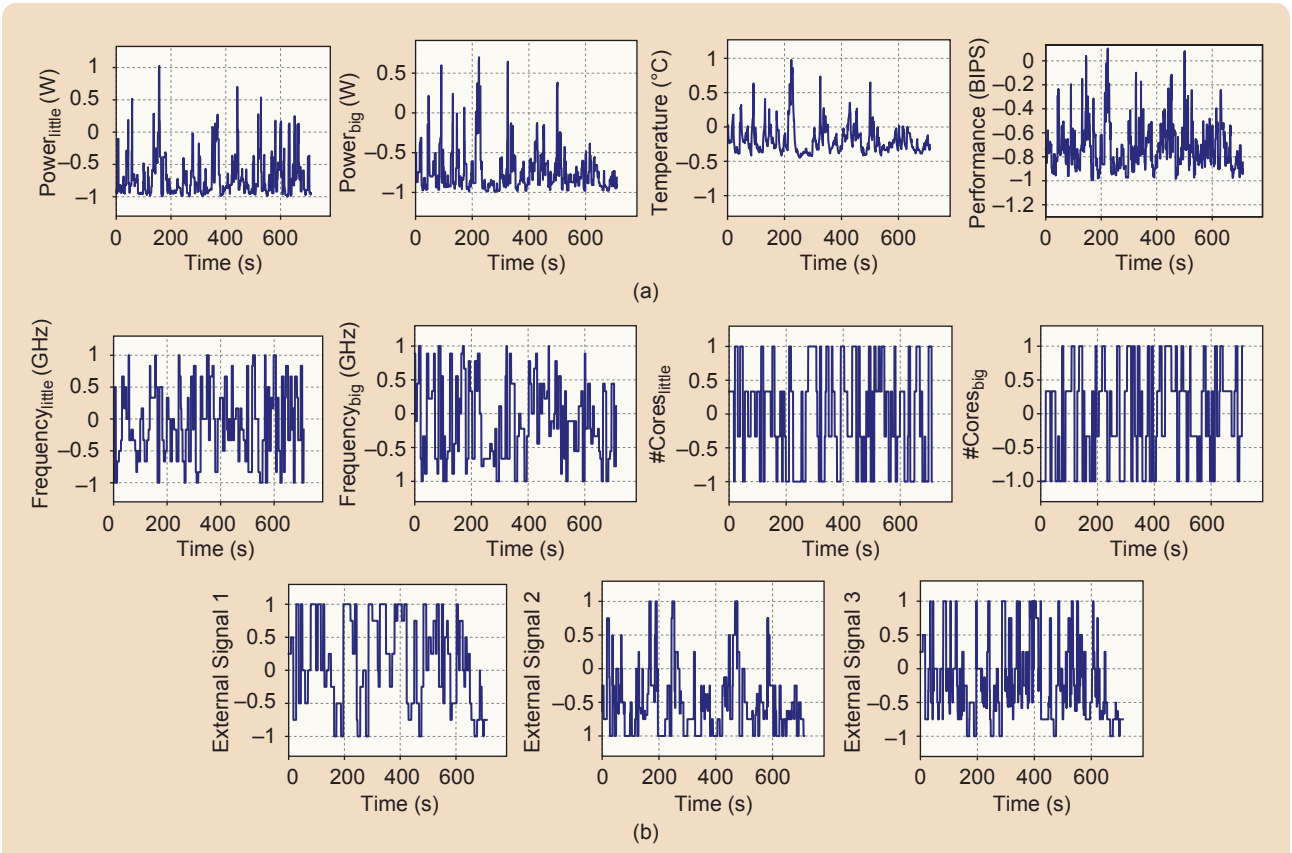

FIGURE 7 The normalized identification data with the PARSEC benchmark swaptions. (a) Normalized inputs used during system identification. The top row shows how the hardware outputs change through time in response to the identification inputs. (b) Normalized inputs used during system identification using swaptions. The bottom row of plots shows how we change the four hardware inputs (u_{HW1}–u_{HW4}) and the external signals received from the operating system (es_1–es_3). BIPS: billions of instructions per second.

hardware outputs from the external signals (*es*) that the OS sends to the hardware as well as the local inputs into the hardware. The frequency response of the model $H(z)$ is evaluated on the unit circle using the substitution $z = e^{j\omega T}$ with $0 < \omega < \omega_N = (\pi/T)$, where ω_N is the Nyquist frequency, and $T = 0.5$ s is the sampling interval [60]. This is the frequency range shown on the horizontal axis in Figure 8. ω_N is also shown as a vertical line in the subplots. Figure 8 includes the confidence regions that are given by the identification methods.

A feature of the models is that the frequency response of many outputs is nearly flat. This is expected because the sampling interval used in this system (0.5 s) is much longer than the physical timescales at which the outputs respond to the inputs. As a result, we capture only coarse-grained dynamics that are observed at a long time interval.

μ Synthesis

The structure in Figure 9(a) is used for the μ synthesis. $P_\circ$ is the identified nominal model. We consider two forms of uncertainty. The first (Δ_{op}) is the output multiplicative uncertainty to account for intra- and interlayer modeling limitations. This is bounded by W_{op}. The second form (Δ_{nl}) is the additive uncertainty used to model the input nonlinearity. The additive uncertainty setup lies inside a complex disk of radius 0.5, centered on the real axis at 0.5 [61]. W_p is the tracking-error bounds of the outputs, and W_u represents the input weights. Figure 9(a) does not show the scaling weights to normalize the signal measurements.

The structure in Figure 9(a) can be reorganized as a linear fractional transformation (LFT), shown on the left side of Figure 9(b), by pulling out the uncertain elements. Δ consists of block-diagonal uncertain elements,v including the fictitious Δ_{pe} used for enforcing robust performance (that is, W_p and W_u). The right side of Figure 9(b) displays the nominal closed loop $M = F_l(P_\circ, K)$ (that is, the lower LFT of $P_\circ$ and K). Suppressing the frequency dependence and interpreting M as $M(e^{j\omega T})$ for a particular frequency, ω (and similarly for the other systems involved), the SSV μ of M is defined as [58]

$$\mu_\Delta(M) = \frac{1}{\min\{\bar{\sigma}(\Delta): \det(I - M\Delta) = 0, \Delta \in \Delta\}} \tag{12}$$

where $\bar{\sigma}(\Delta)$ is the largest singular value of Δ. By the structured small-gain theorem [58], the system is internally stable and meets the performance criteria if and only if

$\mu_\Delta(M) \leq 1$ for all ω. The controller K is obtained by using the DK iteration of Matlab's Robust Control Toolbox [57].

The weights used in this structure are given in Table 5 and have the form $(k(z-a)/z-b)$. The weights to bound the uncertainty (W_{op}) are set from model validation experiments and confidence estimates from the identified model given by Matlab. For the hardware, $(0.8(z-0.55)/z-0.1)$ is used as the uncertainty bounds for all of the outputs, and for the OS, $(z-0.55/z-0.1)$ is used for every output. A higher uncertainty is set for the OS layer, as it is immediately affected by the changing number and nature of the application tasks.

For tracking the performance weights in the hardware (W_p), $(5(z-1)/z-0.55)$ is used for $\text{Power}_{\text{little}}$, $\text{Power}_{\text{big}}$, and the temperature, and $(2.5(z-1)/z-0.55)$ is used for the throughput. Outputs related to the power and temperature have tighter bounds, as they are vital to the system integrity. In the OS, $(2.5(z-1)/z-0.55)$ is used for all of the outputs, since they are all related to performance. Our weights emphasize the reference tracking, as the common case is to track the changing targets to optimize a metric.

Figure 10 displays the Bode magnitude plots of the weights used to represent the uncertainty bounds and inverse of the tracking-performance weight in the hardware layer. The OS

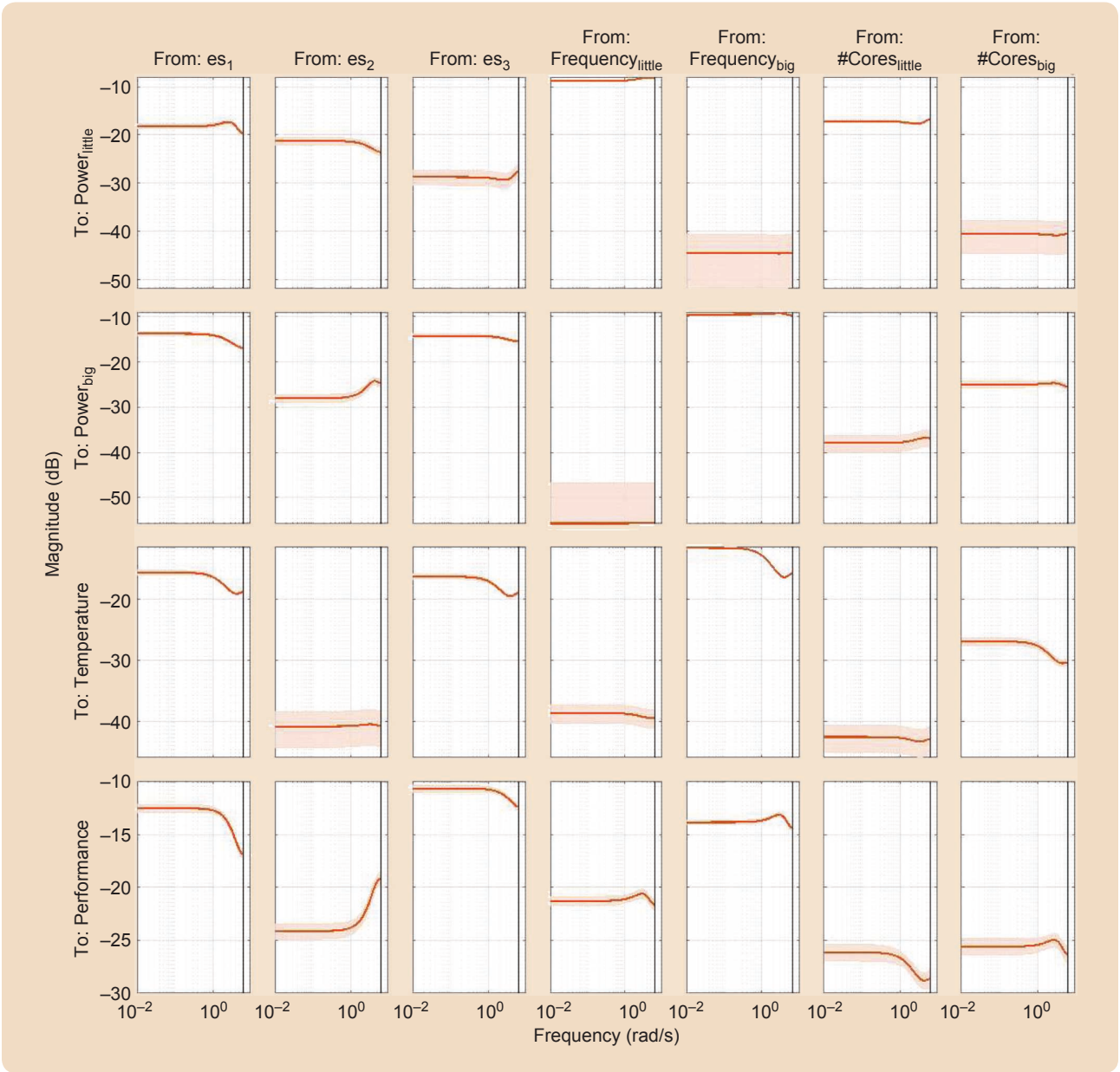

FIGURE 8 The Bode magnitude plot of the identified hardware model evaluated on the unit circle. The frequency range on the x-axis ω is based on the sampling interval ($T = 0.5$ s): $0 < \omega < \omega_N = \pi/T$, where ω_N is the Nyquist frequency. ω_N is shown as a vertical line the subplots.

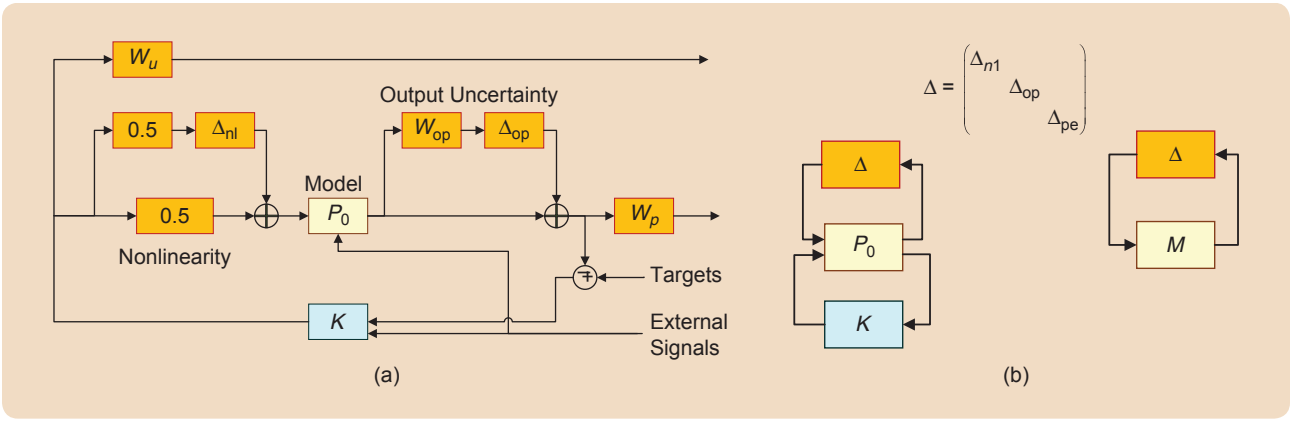

FIGURE 9 The structure specification for μ synthesis. P_0 is the identified nominal model. Δ_{op} is the output multiplicative uncertainty to account for intra- and interlayer modeling limitations. This is bounded by W_{op}. Δ_{nl} is the additive uncertainty used to model the input nonlinearity. W_p is the performance weight for the tracking-error bounds of the outputs, and W_u is the input weight. (a) The closed-loop structure for each layer. (b) The linear fractional transformation representation.

TABLE 5 The weights used in the structure specification.

Weight	Hardware	Operating System
$\mathbf{W_{op}}$	$\frac{0.8(z-0.55)}{z-0.1}$ for all outputs	$\frac{z-0.55}{z-0.1}$ for all outputs
$\mathbf{W_p}$	$\frac{5(z-0.1)}{z-0.55}$ for $\text{Power}_{\text{little}}$, $\text{Power}_{\text{big}}$, and temperature $\frac{2.5(z-0.1)}{z-0.55}$ for throughput	$\frac{2.5(z-0.1)}{z-0.55}$ for all outputs
$\mathbf{W_u}$	0.5 for $\text{frequency}_{\text{little}}$ and $\text{frequency}_{\text{big}}$ One for $\#\text{cores}_{\text{little}}$ and $\#\text{cores}_{\text{big}}$	2 for all inputs

weights are not shown, since they are similar to those of the hardware layer. The input weights (W_u) are set based on the relative overheads of changing the inputs in the hardware. We use 0.5 for $\text{Frequency}_{\text{little}}$ and $\text{Frequency}_{\text{big}}$ and one for $\#\text{Cores}_{\text{little}}$ and $\#\text{Cores}_{\text{big}}$. For the OS, all of the inputs have a weight of two, since they involve migrating a task from one core to another and have similar overheads. The OS has more conservative input weights, as it is closer to the application unpredictability.

Using the model and weights, separate μ controllers are obtained for the hardware and OS layers by using standard routines from Matlab [62]. Since the order of the controllers generated by Matlab's μ synthesis routines is large (68 for the hardware μ controller and 82 for the OS μ controller), the controller dimension is reduced through Hankel singular values. For each controller, the states whose Hankel singular-value contributions are lower than 0.01 are removed. As a result, the number of hardware and OS μ-controller states lowers to only 20 and 16, respectively. Figure 11 charts the μ bounds for the closed-loop system in each layer. It is shown that the controllers can provide robust stability and performance since $\mu_\Delta(M) \leq 1$ for all of the frequencies.

Optimizer Design

The optimizer searches for the output targets to minimize a metric in its layer. It is invoked periodically at 1-s intervals and runs an algorithm to generate the output targets. Since the controller runs every 0.5 s, each set of targets issued by the optimizer is used for two controller invocations. Intuitively, the optimizer's algorithm searches along two directions: a high-throughput region (Up) and a low-throughput region (Down). To move up in the high-throughput area, the algorithm increases the target of the throughput and that of the output, which is significantly below its maximum limit. Alternatively, to move down in the low-throughput zone, the algorithm decreases the target of the throughput and that of the output, which is close to its limit.

Algorithm 1 is the hardware optimizer that runs to maximize $\text{Throughput}^2/\text{Power}$ under the constraints in (7). It is based on the algorithm in [12] and modified to support search constraints and multiple outputs. The hardware optimizer reads the outputs in its layer, the limits in (7), convergence bound ϵ, and restart probability δ. It 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 begin again, 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 of the 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 $Output_{\text{agg}}$ (the output other than the throughput that has the smallest margin), $Output_{\text{laz}}$ (the output other than the throughput that has the largest margin), and $Output_{\text{lag}}$ (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 exceed their targets so that the lagging output is brought closer to the target. Since this can result in some of the outputs going beyond 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 beyond 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 compared to the previously achieved value (line 8). If the relative improvement is smaller than the convergence bounds ϵ, the targets are 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 zero and one. 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 target for $Output_{laz}$. It is expected that the controller will be able to increase the throughput much more than

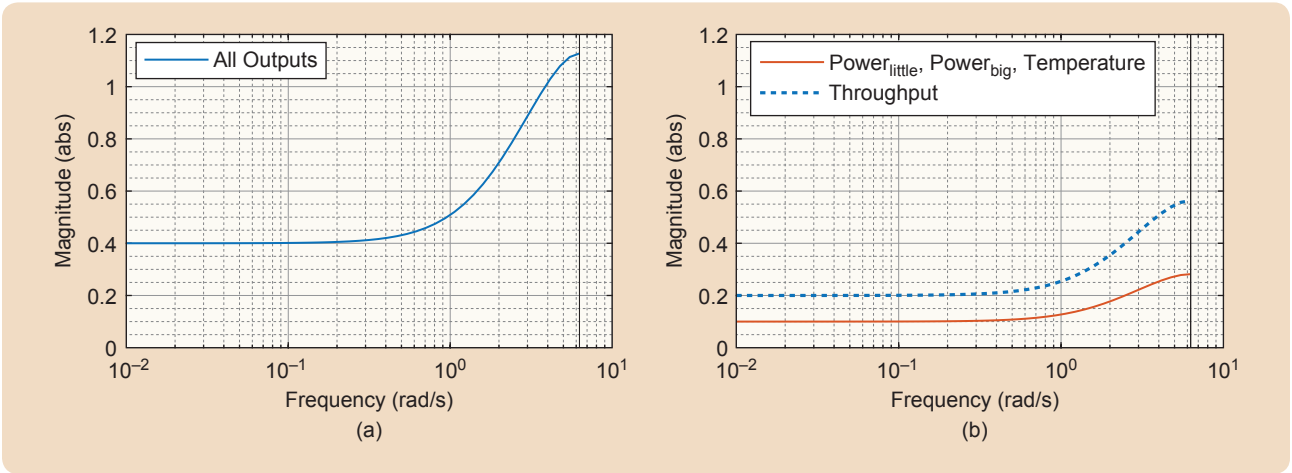

FIGURE 10 The Bode magnitude plots of the weight used to represent the uncertainty bounds $[(0.8(z-0.55)/z-0.1)$ for all outputs] and inverse of the tracking-performance weights $[(5(z-1)/z-0.55)$ for $Power_{little}$, $Power_{big}$, and the temperature and $(2.5(z-1)/z-0.55)$ for the throughput] in the hardware layer. (a) The output multiplicative uncertainty bounds. (b) The inverse tracking-performance weight. abs: absolute value.

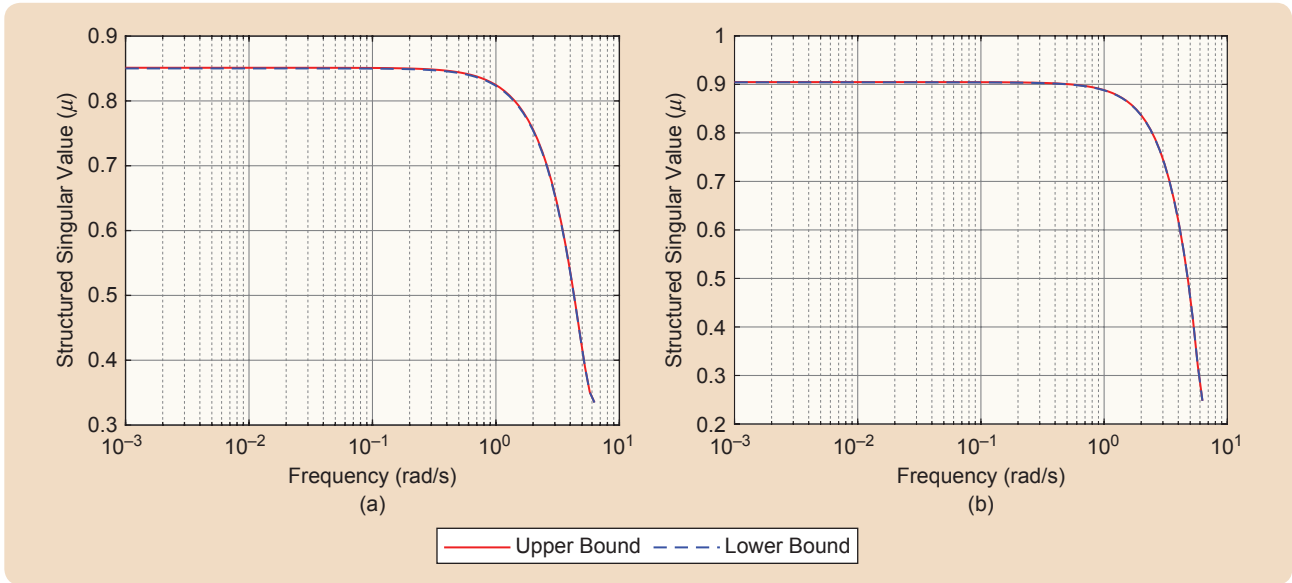

FIGURE 11 The structured singular-value (μ) bounds for each layer. The controllers can provide robust stability and performance since $\mu_{\Delta}(M) \leq 1$ for all of the frequencies on the unit circle. (a) The hardware subsystem. (b) The operating system subsystem.

ALGORITHM 1 The optimizer algorithm to generate targets for the hardware μ controller.

Input: outputs, targets, limits (7), convergence bound ϵ, restart probability δ
Output: New targets for hardware outputs
Initialize: direction ← Up, prev_metric ← 0, stop_search ← False

1 margins ← limits – outputs
2 errors ← 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[$Output_{agg}$] ← *decrease*(target[$Output_{agg}$])
5 target[$Output_{lag}$] ← *decrease*(target[$Output_{lag}$])
6 **else**
7 $\text{metric} \leftarrow \frac{\text{Throughput}^2}{\text{Power}}$
8 $\Delta metric \leftarrow abs\left(\frac{metric - prev\_metric}{prev\_metric}\right)$
9 **if** $\Delta metric > \epsilon$ **or** $rand() < \delta$ **then**
10 **if** *direction* = *Up* **then**
11 **if** *metric* > *prev_metric* **then**
12 target[Throughput] ← *large_increase*(target[Throughput])
13 target[$Output_{laz}$] ← *increase*(target[$Output_{laz}$])
14 **else**
15 dir ← Down
16 target[Throughput] ← *decrease*(target[Throughput])
17 target[$Output_{agg}$] ← *large_decrease*(target[$Output_{agg}$])
18 **end**
19 **else**
20 **if** *metric* > *prev_metric* **then**
21 target[Throughput] ← *decrease*(target[Throughput])
22 target[$Output_{agg}$] ← *large_decrease*(target[$Output_{agg}$])
23 **else**
24 dir ← Up
25 target[Throughput] ← *large_increase*(target[Throughput])
26 target[$Output_{laz}$] ← *increase*(target[$Output_{laz}$])
27 **end**
28 **end**
29 **end**
30 prev_metric ← metric
31 **end**

the power and 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 the power use and temperature rising.

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 target for $Output_{agg}$. The expectation for this move is that the controller will be able to reduce the power and temperature much more than the 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. Otherwise, it reverses to the Up direction.

On the prototype computer, the *increase()* and *decrease()* functions raise and lower 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 that the outputs can withstand. The convergence bound ϵ is 0.05, and the 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. It follows the intuition that the best value of the metric $\text{Throughput}^2/\text{Power}$ occurs in the high or low throughput region. Therefore, instead of searching for all of the output targets simultaneously, each decision of the algorithm changes the throughput target and the target of another output that is necessary to improve the throughput. Additionally, the algorithm does not have to explicitly account for changing system conditions because it relies on the μ controllers to robustly keep the outputs near the targets. Finally, implementing the algorithm requires only a little 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 the performance) and little-side (where the little cores contribute more to the performance). The algorithm finds the best targets for the OS outputs in this space.

Remarks on the Control System Design

In our design, the parameters for each μ controller and optimizer are set independently using only the knowledge of their respective layers. This is a key design requirement to accomplish (see the "Resource-Control Design Requirements" section). Our control system architecture can meet the tracking and optimization goals as needed. Finally, the controllers and optimizers can be synthesized using off-the-shelf design methods and tools that are easily accessed by computer designers.

EVALUATING THE MULTILAYER μ-CONTROL SYSTEM ON THE PROTOTYPE COMPUTER

We first present the storage and computation overheads of the controllers. Then, we evaluate the control system's effectiveness in managing the prototype computer running a variety of applications.

Implementation Overheads

Table 6 lists the implementation overheads for the μ controller in each layer. In the table, the number of operations includes the 32-b fixed-point additions and multiplications. The power consumed for the computation is measured on an ARM little core. For the hardware controller, there are four inputs, four outputs, three external signals from the OS, and 20 states. The controller must store nearly 2.6 KB of data. At every ms-level invocation, it performs nearly 700 32-b fixed-point operations (additions and multiplications). It was determined that performing these computations on an ARM Cortex A7 core consumes $\approx$20–25 mW and takes $\approx$28 μs. These values are small because the μ controllers must perform only matrix-vector calculations to generate decisions. The overheads are low enough to be easily used in computer hardware. The OS controller has similar overheads.

TABLE 6 The implementation overheads of μ controllers.

Parameter	Hardware μ	Operating System μ
Dimension	20	16
Required storage	2.6 KB	2.1 KB
Number of operations	$\approx$700	$\approx$600
Computation time	$\approx$28 μs	$\approx$25 μs
Power consumption	$\approx$20–25 mW	$\approx$20–25 mW

Overall Comparison With the State of the Art

We compare our multilayer μ-control design (called *Multilayer SSV*) with a state-of-the-art control system that was commercially developed for our computer. We use this design for comparison because 1) it has a controller for each layer that exchanges information with the other layer (similar to our design), 2) it is designed by industry experts, and 3) it is deployed in real computers. Other designs from research typically focus on controlling specific applications and managing only one layer. Moreover, such designs are usually specific to the computing platforms they consider. It is not clear how to compose these designs in different layers and port them on our computer. Therefore, we chose the industry-class control system for comparison.

In the control system used for comparison, the hardware and OS controllers are based on designs from ARM, Linaro, and Samsung [63], [64]. The OS controller is similar to the heterogeneous multiprocessing task scheduler from ARM, Linaro, and Samsung except that it is modified to optimize Energy × Delay. In this design, the hardware controller sets the number of cores and their frequency to the maximum values until the power of the big or little clusters or temperature exceed their limits; when that happens, it finds a lower, safe frequency value for that cluster [65]. Further, the OS controller reads the number, type, and frequency of the available cores from the hardware controller to schedule tasks. Similarly, the hardware controller reads how the tasks are distributed across all of the cores to determine the safe frequency. This design is called *Heuristics* because the state-of-the-art controllers in industry are built with heuristics [63]–[65].

We test our designs by running applications from PARSEC (*blackscholes, bodytrack, facesim, fluidanimate, raytrace, x264, canneal*, and *streamcluster*) and SPEC06 application suites (*h264ref, mcf, omnetpp, gamess, gromacs* and *dealII*) and their combination. Figure 12 displays the Energy × Delay of the applications with Multilayer SSV and Heuristics. The bars from left to right correspond to the SPEC applications, average of the SPEC applications, PARSEC applications, average of the PARSEC applications, and average across all of the applications. The applications are abbreviated to their first three characters. For each application, the bars are normalized to Heuristics.

Compared to Heuristics, Multilayer SSV reduces the Energy × Delay by 50%. The execution times and energy consumption (not shown) are reduced by 38 and 20%, respectively. The reason for this significant benefit is that the robust controllers in each layer were stable and could find the best settings for their layer under the influence from other layers. In Multilayer SSV, the actuation costs, output priorities, and uncertainty bounds are explicitly included in the controller design. The resulting controllers perform robustly during uncertainty. Heuristics incorporates similar information implicitly, using ad hoc rules and offering no stability or robustness guarantees. Hence, μ controllers result in a substantial advance from existing systems. We explain this improvement in detail by considering how these control systems manage the *blackscholes* application.

Analysis of a Specific Application: *blackscholes*

We present how Heuristics and Multilayer SSV differ by focusing on the *blackscholes* application. As discussed in the section "Interaction Between Layers and the Variability of Applications," this application begins with a single task and later launches eight new ones that run simultaneously. Once the eight tasks conclude, a single task performs some work, and the application concludes. Within a phase, the work performed by the application does not have large variations. Figure 13 documents how the control systems manage the outputs through time for this application.

Consider Heuristics. The application begins with one task that runs until 50 s. Then, it suddenly launches eight new tasks, resulting in a rapid increase in the power of the

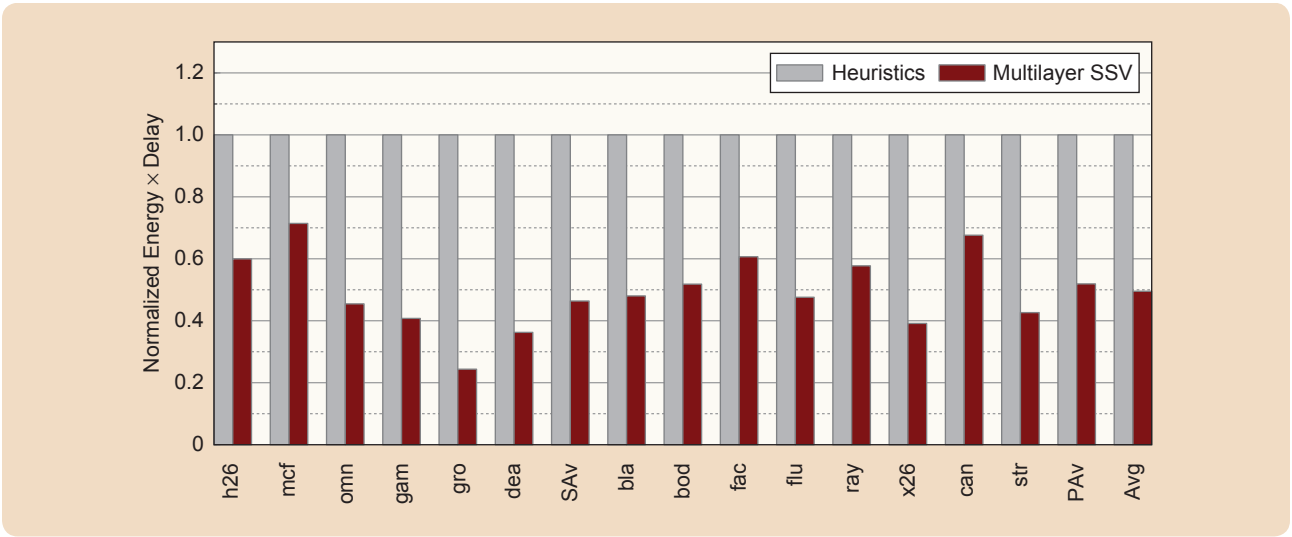

FIGURE 12 The comparison of Energy × Delay (lower is better). The bars from left to right correspond to SPEC applications, average of the SPEC applications (S*Av*), PARSEC applications, average of the PARSEC applications (P*Av*), and the average across all applications (*Avg*). For each application, the bars are normalized to Heuristics. SSV: structured singular value.

clusters, temperature, and performance. From that point, there are many oscillations in the power of both clusters and performance. Even though each layer's controller measures the inputs from the other layer for coordination, the ad hoc rules offer no robustness. The outputs continue to fluctuate, and the application takes 270 s to complete.

In contrast, Multilayer SSV has a significantly smoother behavior. Even though the hardware and OS controllers have been independently developed, they are robust against the interlayer interference. The outputs are kept within limits, and the application has a higher throughput. At approximately 50 s, when the application suddenly launches eight new tasks, the controllers quickly react to bring the power of each cluster and the temperature below their critical values. The controllers' fast response speeds up the search to optimize the Energy × Delay. Notice from Figure 13(d) that the instantaneous performance with Multilayer SSV is nearly 1.5 times that of Heuristics during the phase with eight tasks, with a comparable power consumption and temperature. As a result, the application completes in 180 s, much earlier than with Heuristics.

Evaluating Heterogeneous Application Combinations

We evaluate how the controllers would manage a completely new application behavior on our computer. Four application mixes are run on our computer (each with multiple tasks), and the control system's effectiveness is evaluated. Recall that the model identification experiments did not use any such data. In each mix, there is one application from PARSEC that can launch up to four tasks and one from SPEC that can generate four more. These mixes are *blmc* (*blackscholes*+mcf), *stga* (streamcluster+gamess), *blst* (*blackscholes*+streamcluster), and *mcga* (mcf+gamess). Figure 14 shows the Energy × Delay with Heuristics and Multilayer SSV, normalized to Heuristics. On average, Multilayer SSV achieves a 47% lower Energy × Delay than Heuristics. This demonstrates a Multilayer SSV robustness that could keep the computer efficient, even when encountering new application behavior.

Comparison With Decoupled Control Systems

We evaluate the importance of robustness and controller coordination by comparing Multilayer SSV against two decoupled control systems to minimize Energy × Delay on our computer. These systems have a controller running in the hardware and OS layers. However, one layer's controller does not read any information from the other's controller.

The first decoupled system is based on industry implementations and runs with heuristics. In this scheme, the OS controller assigns tasks to the cores in a round-robin manner without considering the type and frequency of each core. The hardware controller is similar to the performance-power governor in Linux [66]. It sets the number of cores in each cluster as well as the cluster's frequency to their maximum values whenever the power of each cluster and the temperature are below their limits. When the power or temperature exceeds its limits, the controller uses rules to temporarily reduce the frequency of each cluster first, followed by the number of cores in each cluster. It does not use information about how many threads are running on a core to actuate the inputs.

The second decoupled system uses an LQG servo controller (LQ integrators with Kalman estimators) in each layer instead of the SSV controllers. We reuse the optimizers from Multilayer SSV with the LQG controllers. To synthesize the LQG controllers, we use the weights for the inputs and outputs that are comparable to those employed for the corresponding μ controllers. For the

noise-covariance matrices, we use values that are comparable to the uncertainty bounds in the μ synthesis.

On average, decoupled heuristics result in 52% higher Energy × Delay values than Heuristics does. This is due to heuristics' lack of coordination and poor robustness. The ad hoc heuristics in each layer are aggressive and offer no guarantees by themselves. When used together, they destructively interfere with each other. The hardware controller increases the number of cores and their frequency to the maximum, while the OS controller spreads the application tasks on as many cores as available. This causes the power to exceed the limit, triggering the hardware controller to reduce the frequency of the cores and even shut down some of them. When the power and temperature are below their limits, the hardware controller again increases the number of cores and their frequency to the maximum. This cyclical pattern continues for a long time, resulting in inefficiency.

Decoupled LQG controllers have nearly the same Energy × Delay as Heuristics. This is much better than decoupled heuristics but far worse than Multilayer SSV. LQG controllers are more robust than decoupled heuristics. However, the separate LQG controllers do not communicate, while the controllers in Heuristics do. Therefore, Heuristics has an advantage in this aspect. Overall, decoupled LQG control outperforms decoupled heuristic control, but it is at the same level as Heuristics.

Compared to Multilayer SSV, the decoupled LQG has two disadvantages. LQG controllers do not communicate, and they are more conservative than the μ controllers for our computer. To achieve the same level of robustness as our μ controllers (against interlayer interaction, actuator nonlinearities, and model limitations), the LQG controller's response has become too slow compared to μ control. For example, when the targets were changed, the hardware LQG

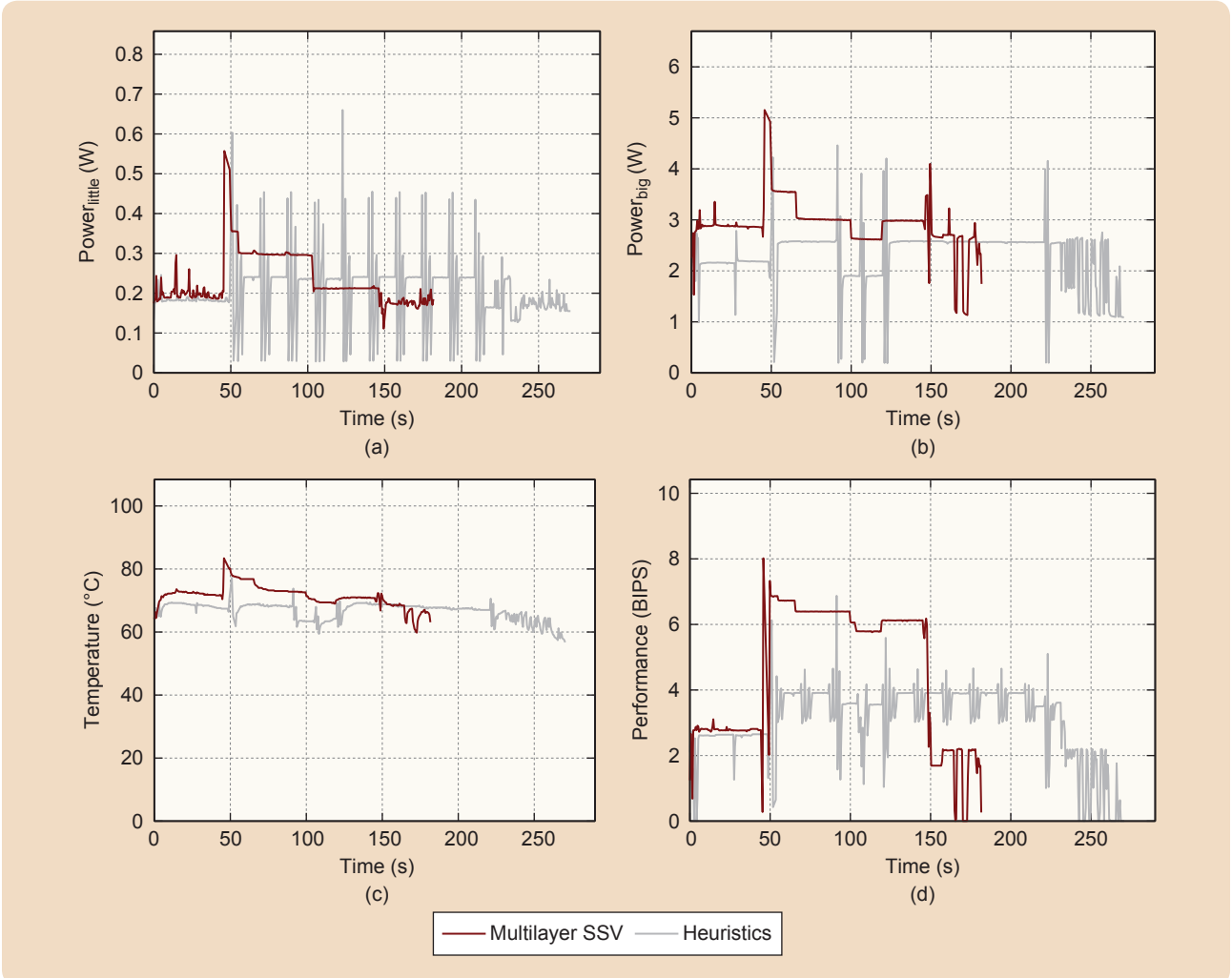

FIGURE 13 The regulation of *blackscholes* through the two control systems. This application begins with a single task and later launches eight parallel tasks. The work in the parallel phase does not have large variations. Finally, the parallel tasks complete their work, and the application terminates. (a) The power of the little cluster. (b) The power of the big cluster. (c) The temperature. (d) The application throughput. BIPS: billions of instructions per second.

There is no work that addresses the design of modular coordinated controllers for multilayer computers.

controller usually converged to the new targets in six sampling intervals, compared to only two for the μ. Consequently, the optimizer's search takes longer. As a result, the decoupled LQG performs poorly compared to Multilayer SSV. Overall, the proposed Multilayer SSV system has the required robustness, coordination, and fast response to keep computers efficient. Its modular design fits naturally with the multilayer organization of computers. This is a significant step beyond the existing designs for computer control.

RECENT DEVELOPMENTS AND FUTURE WORK

We recently collaborated with researchers from industry to develop a new hierarchical control network for managing modularly designed heterogeneous hardware [67], [68]. This recent work examines only the problem of composing control across heterogeneous systems in the hardware layer, unlike the present article (which considers the interaction between multiple system layers). Modern hardware has many independently designed heterogeneous subsystems (see "Multilayer-Computer Organization"). It is necessary to generate local decisions quickly in each subsystem and coordinate the different subsystems for global optimization. In practice, global coordination among subsystems is considered difficult, and current commercial systems use centralized controllers in the hardware. The result is a high response time and design cost due to the lack of modularity.

The problem of coordinating the resource control in heterogeneous hardware is different than the multilayer-computer control problem described in this article. Heterogeneous hardware is integrated hierarchically, such as cores into processors and multiple processors into heterogeneous nodes. There are resource constraints for each subsystem at each level of integration. In contrast, multilayer controllers do not have a hierarchy. Additionally, the outputs (that is, the system power) are hardware resources that must be distributed optimally between the subsystems. For example, if an application runs better on a graphics processor than a conventional processor, the graphics processor must be enabled to utilize as much power as it can, and the conventional processor should be disabled. In the multilayer-control problem, simply budgeting power and the throughput across layers is not meaningful. The OS and applications cannot consume power by themselves without running on the hardware.

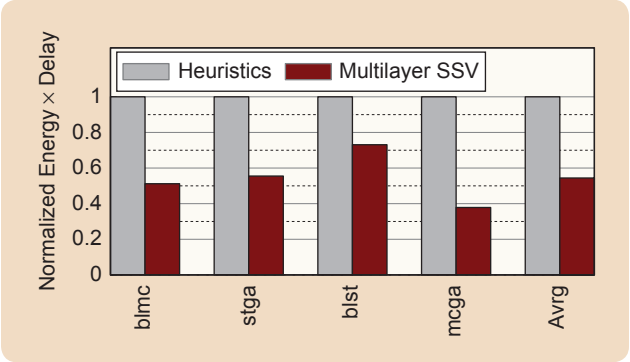

FIGURE 14 The comparison of Energy × Delay for heterogeneous application combinations. Each workload has a four-task PARSEC application plus four copies of a single-task SPEC application. The workloads are *blmc* (*blackscholes*+mcf), *stga* (streamcluster +gamess), *blst* (*blackscholes*+streamcluster), and *mcga* (mcf+ gamess). The models were not trained under such conditions. The Energy × Delay with Heuristics and Multilayer SSV are normalized to Heuristics.

An additional aspect of [67] is that it includes system safety as a resource control goal, one that is omitted from the present article. Real hardware has several mechanisms to protect computers from hazardous operating conditions such as high currents and temperatures. These are hard constraints, unlike the soft ones that are considered in the present work. Therefore, they are handled through engines that operate at faster timescales than those for optimization and tracking goals.

When optimizing engines operate without being aware of the interference from safety engines, the result is inefficiency. For example, when a performance controller increases the processor frequency to improve the throughput, it raises the processor temperature as a side effect. When the temperature exceeds a critical threshold, safety mechanisms will immediately lower the frequency. If the performance controller restores the frequency to the previous high value, it could repeatedly trigger the safety mechanisms, which would lower the frequency. Overall, this oscillatory behavior results in poor performance. The latest work [67] proposes a new controller for each subsystem that combines multiple engines for optimization and safety and has a standard interface. Building the controller for a subsystem requires knowing about only that subsystem. As a heterogeneous computer is assembled, the controllers in the different subsystems are connected hierarchically, exchanging standard coordination signals.

The controller design in the new work is also based on robust-control-theory principles and extends some of the ideas in the present article. The optimizing engine in a controller has a μ controller and planner. The planner provides targets to the controller, similar to the optimizer in the

The area of computer control is an exciting field with many opportunities for the control systems community to contribute.

present work. Additionally, it communicates with the parent and child controllers. The uncertainty bounds that are used to design the μ controllers capture the interaction between the subsystems and interference from the safety engines.

A natural path forward is to integrate the two ideas: develop distributed-control networks for all computer layers and connect them using standard interfaces. The networks may be organized differently in each layer. For example, the hardware control layer in [67] has a hierarchical nature because it matches the underling hardware-system structure. The OS does not have such a structure and may need a different organization. Overall, the area of computer control is an exciting field with many opportunities for the control systems community to contribute. For example, the lack of accurate, standard models is a significant challenge that merits novel designs from robust and adaptive control. It is possible to use ideas from extremum-seeking control to eliminate the use of a separate optimizer.

The problem of reconciling hard safety constraints with optimizing control can be likened to the control problem for self-driving cars and unmanned aerial vehicles. Switched and hybrid-control designs can be useful in this scenario. Further, the design of distributed-control networks can benefit from swarm and multiagent cooperative-control methods. Such distributed-control networks are particularly important for managing large data centers, cloud systems, and cyber-physical systems. Lastly, it is interesting to see if machine-learning techniques (specifically, reinforcement-learning methods) can be useful to control the system through online data. An example use of such learning modules would be to replace the optimizer in our design or redesign the controllers online. One important requirement for any control solution to be adopted as mainstream by computer system designers is that the control design must be supported with tools and abstractions that computer designers can easily use. Addressing this new application area is a great opportunity for the control systems community.

CONCLUSION

This article presented a novel control system to attain high resource efficiency in computers, outlined several challenges in using control theory for systematic computer control, and showed how our system meets these challenges. The proposed control system is based on linear robust control and provides modular coordinated control for modern multilayer computers. The scheme considers interlayer interactions as uncertainty and relies on modular μ controllers to be robust to this uncertainty. The controllers can be designed independently and are guaranteed to work in coordination. On a representative computer, our two-layer control system reduced the Energy × Delay of a set of programs by 50%, on average, beyond the state of the art. We hope that the insights from this article will stimulate more advanced work on building formal controllers for computers.

ACKNOWLEDGMENT

This work was supported, in part, by the National Science Foundation under grants CNS 17-63658, CCF 17-25734, and CCF 16-29431.

AUTHOR INFORMATION

Raghavendra Pradyumna Pothukuchi (pothuku2@illinois.edu) is a Ph.D. candidate at the University of Illinois at Urbana–Champaign (UIUC). He received the B.E. degree (with honors) in electrical and electronics engineering from the Birla Institute of Technology and Science, Pilani, India, where he was awarded the University Gold Medal upon graduation, and the M.S. degree in computer science from UIUC in 2014. He worked at Nvidia before beginning his graduate studies. His research focuses on building intelligent systems for security and extreme efficiency. His primary research area concerns computer systems architecture, which overlaps with control theory, machine learning, security, operating systems, and distributed systems. He received the W.J. Poppelbaum Memorial Award from the Department of Computer Science, UIUC, a UIUC Mavis Future Faculty Fellowship, and an Association of Computing Machinery Student Research Competition. He was chosen as a rising star in computer architecture in 2018. He interned at Advanced Micro Devices, and this collaborative work on modular control for heterogeneous systems resulted in a joint patent.

Sweta Yamini Pothukuchi is a Ph.D. candidate at the University of Illinois at Urbana–Champaign (UIUC). She received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, and the M.S. degree in computer science from UIUC in 2015. She previously worked at Microsoft. Her research focuses on efficient code generation for distributed-memory systems from high-level language constructs. Her work includes the areas of language design, compilers, parallel programming, and autotuning.

Petros G. Voulgaris received the Diploma in mechanical engineering from the National Technical University, Athens, Greece, in 1986 and the S.M. and Ph.D. degrees

in aeronautics and astronautics from the Massachusetts Institute of Technology, Cambridge, in 1988 and 1991, respectively. Since 1991, he has been with the Department of Aerospace Engineering at the University of Illinois at Urbana–Champaign, where he is currently a professor (with appointments in the Coordinated Science Laboratory and Department of Electrical and Computer Engineering). His research interests include optimal, robust, and distributed control and estimation; networked control; and applications of advanced control methods to engineering practices (including power systems and air-vehicle, nanoscale, robotic, and structural-control systems). He is a recipient of several awards, including the National Science Foundation (NSF) Research Initiation Award, Office of Naval Research (ONR) Young Investigator Award, and University of Illinois at Urbana–Champaign Xerox Award for research. He was also a visiting Abu Dhabi Gas Liquification Company Chair Professor of Mechanical Engineering at the Petroleum Institute, Abu Dhabi, United Arab Emirates (2008–2010). His research has been supported by several agencies, including the NSF, ONR, Air Force Office of Scientific Research, and NASA. He is a Fellow of the IEEE.

Josep Torrellas is a professor of computer science at the University of Illinois at Urbana–Champaign (UIUC). He is the director of the Center for Programmable Extreme Scale Computing and past director of the Illinois-Intel Parallelism Center. He received the Ph.D. degree from Stanford University, California. His research interests include multiprocessor computer architectures. He received the IEEE Computer Society 2015 Technical Achievement Award and the 2017 UIUC Campus Award for Excellence in Graduate Student Mentoring. He chairs the IEEE Technical Committee on Computer Architecture and has been a member of the Computing Research Association (CRA) board of directors as well as a council member of the CRA's Computing Community Consortium. He serves on the International Roadmap for Devices and Systems Team and is a member of the U.S. Board on Army Research and Development. He is a Fellow of the IEEE, Association for Computing Machinery, and American Association for the Advancement of Science.

REFERENCES

[1] N. Beck, S. White, M. Paraschou, and S. Naffziger, "'Zeppelin': An SoC for multichip architectures," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 40–42. doi: 10.1109/ISSCC.2018.8310173.

[2] B. Sinharoy et al., "Advanced features in IBM POWER8 systems," *IBM J. Res. Dev.*, vol. 59, no. 1, pp. 1:1–1:18, Jan.–Feb. 2015. doi: 10.1147/JRD.2014.2374252.

[3] T. D. Sherer, "Introduction to power management," Microsoft Docs, June 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-power-management

[4] 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. doi: 10.1109/MC.2016.309.

[5] X. Wang, "Intelligent power allocation: Maximize performance in the thermal envelope," ARM, Cambridge, U.K., White Paper, Mar. 2017.

[6] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No 'power' struggles: Coordinated multi-level power management for the data center," in *Proc. 13th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 48–59. doi: 10.1145/1346281.1346289.

[7] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," in *Proc. 16th ACM Symp. Operating Systems Principles*, 1997, pp. 276–287. doi: 10.1145/268998.266708.

[8] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," in *Proc. 21st Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 545–559. doi: 10.1145/2872362.2872375.

[9] V. Vardhan et al., "GRACE-2: Integrating fine-grained application adaptation with global adaptation for saving energy," *Intl. J. Embed. Syst.*, vol. 4, no. 2, pp. 152–169, 2009. doi: 10.1504/IJES.2009.027939.

[10] 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 *Proc. 46th Int. Symp. Microarchitecture*, 2013, pp. 210–221. doi: 10.1145/2540708.2540727.

[11] X. Li et al., "Performance directed energy management for main memory and disks," in *Proc. 11th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 271–283. doi: 10.1145/1024393.1024425.

[12] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Computer Architecture*, June 2016, pp. 658–670. doi: 10.1109/ISCA.2016.63.

[13] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, "Introduction to control theory and its application to computing systems" in *Performance Modeling and Engineering*, Z. Liu and C. H. Xia, Eds. Boston, MA: Springer-Verlag, 2008, pp. 185–215.

[14] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Hoboken, NJ: Wiley, 2004.

[15] P. K. Janert, *Feedback Control for Computer Systems*. Sebastopol, CA: O'Reilly Media, 2013.

[16] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Structured singular value control for modular resource management in multilayer computers," in *Proc. IEEE Conf. Decision and Control*, Dec. 2018, pp. 5121–5127. doi: 10.1109/CDC.2018.8619427.

[17] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: Multilayer resource controllers to maximize efficiency," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Computer Architecture*, June 2018, pp. 505–518. doi: 10.1109/ISCA.2018.00049.

[18] "ODROID-XU3," HardKernel, Anyang, South Korea. Accessed on: 2018. [Online]. Available: https://www.hardkernel.com/shop/odroid-xu3/

[19] "big.LITTLE technology: The future of mobile," ARM, Cambridge, U.K., White Paper, 2013. [Online]. Available: https://www.arm.com/zh/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures and Compilation Techniques*, Oct. 2008, pp. 72–81. doi: 10.1145/1454115.1454128.

[21] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sept. 2006. doi: 10.1145/1186736.1186737.

[22] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "ECOSystem: Managing energy as a first class operating system resource," in *Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 123–132. doi: 10.1145/605397.605411.

[23] E. Rotem, "Intel Architecture, Code Name Skylake Deep Dive: A new architecture to manage power performance and energy efficiency," presented at the Intel Developer Forum, San Francisco, Aug. 2015.

[24] M. Broyles, C. J. Cain, T. Rosedahl, and G. J. Silva, "IBM EnergyScale for POWER8 processor-based systems," IBM, Somers, NY, Tech. Rep., Nov. 2015. [Online]. Available: https://www.ibm.com/downloads/cas/JB3VDKZQ

[25] 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 *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2006, pp. 347–358. doi: 10.1109/MICRO.2006.8.

[26] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proc. Int. Symp. Computer Architecture*, 2002, pp. 233–244. doi: 10.1109/ISCA.2002.1003581.

[27] A. Filieri, H. Hoffmann, and M. Maggio, "Automated multi-objective control for self-adaptive software design," in *Proc. 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 13–24. doi: 10.1145/2786805.2786833.

[28] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann, "Automated control of multiple software goals using multiple actuators," in *Proc. 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 373–384. doi: 10.1145/3106237.3106247.
[29] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *Proc. 36th Annu. Int. Symp. Computer Architecture*, 2009, pp. 314–324. doi: 10.1145/1555754.1555794.
[30] S. Adve et al., "The Illinois GRACE Project: Global resource adaptation through CoopERation," in *Proc. Workshop on Self-Healing, Adaptive and self-MANaged Systems*, June 2002, pp. 1–8.
[31] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, "Formal online methods for voltage/frequency control in multiple clock domain microprocessors," in *Proc. 11th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 248–259. doi: 10.1145/1024393.1024423.
[32] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *Proc. 38th Annu. Int. Symp. Computer Architecture*, 2011, pp. 449–460. doi: 10.1145/2000064.2000117.
[33] 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 *Proc. 26th Int. Conf. Parallel Architectures and Compilation Techniques*, 2017, pp. 260–272. doi: 10.1109/PACT.2017.23.
[34] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. 50th ACM/EDAC/IEEE Design Automation Software*, 2013, pp. 1–9. doi: 10.1145/2463209.2488949.
[35] F. Zaninit, C. Jones, D. Atienza, and G. De Micheli, "Multicore thermal management using approximate explicit model predictive control," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 2010, pp. 3321–3324. doi: 10.1109/ISCAS.2010.5537891.
[36] 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 *Proc. Design, Automation and Test in Europe*, Mar. 2011, pp. 1–6. doi: 10.1109/DATE.2011.5763141.
[37] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. 41st IEEE/ACM Int. Symp. Microarchitecture*, 2008, pp. 318–329. doi: 10.1109/MICRO.2008.4771801.
[38] Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & Cap: Adaptive DVFS and thread packing under power caps," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 175–185. doi: 10.1145/2155620.2155641.
[39] C. Hankendi, A. K. Coskun, and H. Hoffmann, "Adapt&Cap: Coordinating system-and application-level adaptation for power-constrained systems," *IEEE Design Test*, vol. 33, no. 1, pp. 68–76, 2016. doi: 10.1109/MDAT.2015.2463275.
[40] 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. doi: 10.1145/2541228.2541238.
[41] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. 19th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 127–144. doi: 10.1145/2541940.2541941.
[42] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning control for predictable latency and low energy," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 184–198. doi: 10.1145/3173162.3173184.
[43] 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 *Proc. IEEE Int. Real-Time Systems Symp.*, 2007, pp. 227–238. doi: 10.1109/RTSS.2007.48.
[44] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong, "Application-specific performance-aware energy optimization on android mobile devices," in *Proc. IEEE Int. Symp. High Performance Computer Architecture*, Feb. 2017, pp. 169–180. doi: 10.1109/HPCA.2017.32.
[45] 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. doi: 10.1145/2661430.
[46] X. Wang and J. F. Martínez, "ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment," in *Proc. 21st Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 19–32. doi: 10.1145/2872362.2872382.
[47] X. Wang and J. F. Martínez, "XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proc. IEEE 21st Int. Symp. High Performance Computer Architecture*, 2015, pp. 113–125. doi: 10.1109/HPCA.2015.7056026.
[48] S. Fan, S. M. Zahedi, and B. C. Lee, "The computational sprinting game," in *Proc.21st Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 561–575. doi: 10.1145/2872362.2872383.
[49] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and A. Shoemaker, "Flicker: A dynamically adaptive architecture for power limited multicore systems," in *Proc. 40th Annu. Int. Symp. Computer Architecture*, 2013, pp. 13–23. doi: 10.1145/2485922.2485924.
[50] C. Dubach, T. Jones, E. Bonilla, and M. O'Boyle, "A predictive model for dynamic microarchitectural adaptivity control," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 485–496. doi: 10.1109/MICRO.2010.14.
[51] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and memory system DVFS in server systems," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 143–154. doi: 10.1109/MICRO.2012.22.
[52] H. Jung, P. Rong, and M. Pedram, "Stochastic modeling of a thermally-managed multi-core system," in *Proc. 45th ACM/IEEE Design Automation Software*, June 2008, pp. 728–733. doi: 10.1145/1391469.1391657.
[53] M. Guevara, B. Lubin, and B. C. Lee, "Navigating heterogeneous processors with market mechanisms," in *Proc. Int. Symp. High Performance Computer Architecture*, 2013, pp. 95–106. doi: 10.1109/HPCA.2013.6522310.
[54] A. M. Rahmani et al., "SPECTR: Formal supervisory control and coordination for many-core systems resource management," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 169–183. doi: 10.1145/3173162.3173199.
[55] H. Hoffmann, "JouleGuard: Energy guarantees for approximate applications," in *Proc. 25th Symp. Operating Systems Principles*, 2015, pp. 198–214. doi: 10.1145/2815400.2815403.
[56] The MathWorks, Inc., Natick, MA. *MathWorks*. (2015). System Identification Toolbox: R2015a, [Online]. Available: https://www.mathworks.com/help/ident/index.html?s_cid=doc_ftr
[57] The MathWorks, Inc., Natick, MA. *MathWorks*. (2015). Robust Control Toolbox: R2015a https://www.mathworks.com/help/robust/
[58] S. Skogestad and I. Postlethwaite, *Multivariable Feedback Control: Analysis and Design*. Hoboken, NJ: Wiley, 2005.
[59] L. Ljung, *System Identification: Theory for the User*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1999.
[60] *Matlab Control System Toolbox Reference*. Natick, MA: MathWorks, 2018. [Online]. Available: https://www.mathworks.com/help/pdf_doc/control/reference.pdf
[61] R. Y. Chiang and M. G. Safonov, *MATLAB: Robust Control Toolbox User's Guide*. Natick, MA: MathWorks, 1984.
[62] D.-W. Gu, P. H. Petkov, and M. M. Konstantinov, *Robust Control Design with MATLAB*, 2nd ed. New York: Springer-Verlag, 2013.
[63] B. Jeff, "big.LITTLE technology moves towards fully heterogeneous global task scheduling," ARM, Cambridge, U.K., White Paper, Nov. 2013. [Online]. Available: https://www.arm.com/zh/files/pdf/big_LITTLE_technology_moves_towards_fully_heterogeneous_Global_Task_Scheduling.pdf
[64] H. Chung, M. Kang, and H.-D. Cho, "Heterogeneous multi-processing solution of Exynos 5 Octa with ARM big.LITTLE technology," ARM, Cambridge, U.K., White Paper, 2013. [Online]. Available: https://www.arm.com/zh/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf
[65] D. Kim and A. Daniel, "Kernel driver exynos_tmu," The Linux Foundation, San Francisco. Accessed on: 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/thermal/exynos_thermal
[66] D. Brodowski and N. Golde, "Linux CPUFreq governors," The Linux Foundation, San Francisco. Accessed on: 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[67] P. Pothukuchi et al., "Tangram: Integrated control of heterogeneous computers," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 384–398. doi: 10.1145/3352460.3358285.
[68] R. P. Pothukuchi, J. L. Greathouse, and L. Piga, "Distributed multi-input multi-output control theoretic method to manage heterogeneous systems," U.S. Patent Application 15/950 172, Apr. 11, 2018.