Yukta: Multilayer Resource Controllers to Maximize Efficiency

Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

***Abstract*—**
Since computers increasingly execute in constrained environments, they are being equipped with controllers for resource management. However, the operation of modern computer systems is structured in multiple layers, such as the hardware, OS, and networking layers—each with its own resources. Managing such a system scalably and portably requires that we have a controller in each layer, and that the different controllers coordinate their operation. In addition, such controllers should not rely on heuristics, but be based on formal control theory.

This paper presents a new approach to build *coordinated multilayer formal controllers* for computers. The approach uses Structured Singular Value (SSV) controllers from Robust Control Theory. Such controllers are especially suited for multilayer computer system control. Indeed, SSV controllers can read signals from other controllers to coordinate multilayer operation. In addition, they allow designers to specify the discrete values allowed in each input, and the desired bounds on output value deviations. Finally, they accept uncertainty guardbands, which incorporate the effects of interference between the controllers. We call this approach *Yukta*. To assess its effectiveness, we prototype it in an 8-core big.LITTLE board. We build a two-layer SSV controller, and show that it is very effective. Yukta reduces the $E \times D$ and the execution time of a set of applications by an average of 50% and 38%, respectively, over advanced heuristic-based coordinated controllers.

***Keywords*-Multilayer computer control, Energy efficiency, Resource management, Robust control theory.**

I. Introduction

Computing devices are increasingly operating in constrained environments, where resources such as energy, power, or memory bandwidth are limited, and measures such as temperature, Quality of Service (QoS), or throughput need careful control. This has prompted computers to include controllers that manage multiple resources during execution [1], [2], [3], [4], [5], [6].

Modern computing systems are organized in multiple layers, each with its own resources and with partial information about the current execution — e.g., the hardware, Operating System (OS), and networking layers. To meet the multiple resource constraints in an execution, each layer has information available to it that can be used to manage resources from its perspective. There are many proposals on managing resources in a multilayer environment (e.g., [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]).

One approach is to have a single, *monolithic* controller that takes signals from all the layers and controls the resources in all the layers. This solution is complex to develop, difficult to maintain, cannot interoperate across systems, and is not scalable [7], [8], [10], [11], [14], [21]. Indeed, to build these controllers, designers have to understand the inner details of all the layers. Moreover, when one of the layers needs to be modified, these controllers require a complete re-design. Sometimes, this design is infeasible, as when hardware and OS come from different vendors.

An alternative is to have a controller at each layer, and have them operate in an uncoordinated, *decoupled* way. However, without coordination, the overall behavior can be greatly suboptimal [7], [10], [12], [13], [14], [22]. The different controllers may interfere destructively.

An example of such interference is described by Vega et al. [24] in an IBM POWER7. In this system, there is a per-core hardware DVFS controller that changes the core's frequency to maintain high utilization and meet power requirements. In the OS, there is a task scheduler that tries to consolidate threads onto cores and power-gate the resulting unused cores to save power. When the multicore's load goes down, it is expected that the scheduler will consolidate threads to reduce power without hurting performance. Unfortunately, the DVFS controller immediately reduces the frequency to increase utilization, preventing the scheduler from consolidating threads and power-gating cores.

There is a need to use modular controllers in each layer of a multilayer system that collaborate through a mutually agreed interface, without a monolithic or decoupled design. This is the position taken by industry, where hardware and software companies such as ARM and Linaro are working on coordinated hardware-software approaches [6]. Other examples where hardware and software power management modules coordinate with each other include IBM [27] and Intel systems [1], [5], [28]. In these designs, each layer performs its own resource management, and interacts with the other layers through well-defined interfaces.

Unfortunately, most existing coordinated, multilevel designs rely substantially on heuristics [8], [13], [14], [27]. Heuristic designs are highly specific to particular choices, and do not address the general problem. Their algorithms may become unusable when a different hardware or software platform is used. Moreover, even highly-tuned heuristics can perform poorly on application corner cases [12], [29]. The solution, then, is to use formal methodologies such as control theory, whose properties are well studied [30].

Currently, there are no formal control methods to develop coordinated multilayer controllers for computers. Popular formal control designs such as PID controllers [30] and similar Single Input Single Output (SISO) proposals [1],

[31], [32], [33] can only monitor one goal and change one parameter. Some designs [11], [12], [25], [26] use a collection of separate SISO controllers, but cannot manage the interaction between the goals [34], [35], [36]. There are controller designs that operate on Multiple Inputs and Single Output (MISO) [37], [38], [39], [40] or Multiple Inputs and Multiple Outputs (MIMO) [34], [35], [41]. However, all these controllers are intended for standalone use, and do not have channels for coordination between multiple controllers. Some designs employ heuristics to make up for this deficit [11], but this defeats the purpose of formal control methods. Importantly, existing designs are not natively robust to the large uncertainty that appears in the presence of multiple controllers, each acting with partial system information.

In this paper, we present a new approach to build coordinated multilayer formal controllers for computer systems. We consider Robust Control Theory [30], which focuses on uncertain environments, and pick the popular Structured Singular Value (SSV) controller [30], [42], [43], [44]. This is a MIMO controller with four traits that make it suitable for computer system control.

First, SSV controllers can read *External Signals*, which provide information that the controller cannot directly change, but can use to make better decisions; we use them to pass coordinating information between the controllers in different layers. Second, SSV control designers can specify the maximum bounds on the deviations of outputs from their goals, enabling more accurate computer controllers. Third, the design of SSV controllers accepts *uncertainty guardbands*, which are useful to incorporate the effects of interference between independently-designed controllers. Finally, SSV control designers can provide the discretized values allowed in each input — unlike with other controllers, where inputs are assumed to have continuous unlimited values.

We call this approach of using multilayer SSV controllers for computer system control *Yukta*. With Yukta, controllers at different layers can be built with little interaction. This modular design is essential for controlling complex systems like computers. To assess its effectiveness, we prototype it in an 8-core big.LITTLE processor board running Linux. We build a two-layer SSV controller, and show that it is very effective. Yukta reduces the $E{\times}D$ (Energy $\times$ Delay) and the execution time of a set of applications by an average of 50% and 38%, respectively, beyond what advanced heuristic-based coordinated controllers attain. Our contributions are:

1) Applying MIMO SSV control from robust control theory for systematic computer resource efficiency.
2) *Yukta*, an approach for independent teams to design coordinated multilayer controllers with MIMO SSV.
3) A prototype of Yukta on a big.LITTLE multicore board and its evaluation.

II. Background

A. Multilayer Control

There are many works on managing resources from different layers of a computer system (e.g., [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]). As indicated before, designs vary depending on whether they manage resources in a monolithic, decoupled, or coordinated manner. In addition, as we will see in Section VII, they also vary depending on whether they use heuristics, control-theoretic methods, machine learning, or optimization theory. To our knowledge, Yukta is the first approach that uses control-theoretic methods to build coordinated, modular multilayer controllers.

B. Robust Control for Uncertain Environments

Control theory has been used to design computer controllers (e.g., [1], [12], [25], [26], [31], [32], [33], [34], [35], [37], [38], [39], [40]). Most of these proposals use a SISO approach, where the controller only changes one parameter (input) to control one goal (output) [1], [31], [32], [33] — e.g., it changes a core's frequency to control the frame rate of the application. Some designs use a MISO approach, where the controller changes multiple parameters to control one goal [37], [38], [39], [40] — e.g., it changes a core's frequency and L2 cache size to control the core's utilization. Finally, other designs use a MIMO approach, where the controller changes multiple parameters to control multiple goals [34], [35] — e.g., it changes a core's frequency and issue queue to control the core's performance and power consumption. In this case, each of the outputs depends on each of the inputs. The MIMO approach is the most applicable to computer architecture, since multiple goals (performance, power) are typically coupled with each other.

Computers are complex, and program behavior is determined by many factors. As a result, controlling computer environments intrinsically involves dealing with uncertain dynamics and approximate models.

A branch of control theory that focuses on hard-to-predict environments is Robust Control Theory [30]. In this field, variability and uncertainty of the system dynamics at runtime is an integral part of the controller synthesis process. Among the robust controller methodologies, one of the most mature and better understood, with standard packages and tools, is *Structured Singular Value (SSV)* control [30], [42], [43], [44].

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

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

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

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

The designer sets the uncertainty guardband based on a combination of suggestions from theory, system insight, and actual experimentation. Guardbands enable the controller to work correctly in scenarios that are very different from modeled executions. Unlike non-robust controllers, SSV controllers do not become slow unless significantly large guardbands (e.g., over 400%) are used [43]. On the other hand, if the guardband is not large enough and is exhausted at runtime, the controller detects it dynamically, and may no longer provide all the guarantees expected.

Finally, robust controllers accept a description of the allowed input settings. The designer can specify the range of values taken by the inputs and their discrete values (Saturation and Quantization). This is in contrast to typical non-robust controllers, such as PID controllers [30], where each input is assumed to take values that are continuous and unbounded. This makes SSV controllers natively applicable to computing systems, which have discrete resources.

C. *Mathematical Theory of SSV Controllers*

Figure 1 shows the representation of a system when designing an SSV controller. *M* is the model of the system that we want to control. *M* describes how the inputs (*u*) and external signals generate the outputs (*y*). *K* is the controller we have to design. In practice, there are real world inaccuracies shown as Δ in the figure. One is due to the true system behavior deviating from the model (Δ_u), caused by any behavior of the system not captured by the model. This is the model uncertainty for which we specify guardbands. Another constraint is due to the inputs taking only a discrete (or quantized) and limited (or saturated) set of allowed values (Δ_{in}) instead of unlimited values. This is the input discretization. The external signals may also have such effects, but we omit them in this discussion.

The system inside the dotted line boundary in Figure 1 is called the nominal closed loop because it contains the

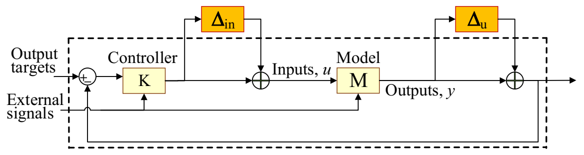


Figure 1: System representation when designing an SSV controller.

components without any imprecisions. Consolidating the individual Δ components into an overall Δ, and denoting the nominal closed loop of Figure 1 as *N*, we can represent the system as Figure 2. In this figure, signals generated from elsewhere (i.e., external signals and output targets or references) are called exogenous inputs (*w*). The outputs of the system that can actually be measured outside are called exogenous outputs (*z*). The Δ block interacts with the system through fictitious signals called perturbation inputs (*d*) and perturbation outputs (*f*) that capture the effects of model uncertainty and discrete inputs.

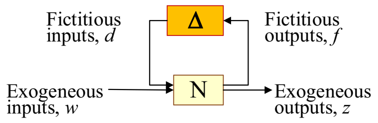


Figure 2: The Δ-*N* representation of the control problem.

The controller K in the closed loop N is robust if it: (i) keeps N stable, (ii) generates optimal inputs according to designer-specified input weights W, and (iii) keeps all visible outputs z within bounds B of the targets – for all possible model inaccuracies smaller than the specified Δ. Robust control theory [30] uses the Structural Singular Value (SSV) defined as follows to assess a controller's robustness:

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

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

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

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

Compare this approach to the design of a non-SSV controller such as a PID controller [30]. In such case, the designer can only specify the model *M* and obtains a controller *K*. There is no way to specify inaccuracies Δ, bounds *B*, and weights *W* in the controller design. As a result, such controllers are less useful in complex multilayer environments like computing systems.

D. Taxonomy of Controllers

Table I presents our taxonomy of designs from control theory. The choices that we select in this paper are italicized. First, the model of the system can be obtained with analytical principles (white box), experimental data (black box), or a combination of both (gray box). Black box models are best when the internals of the system are unknown or too complicated to describe, as in computers. Next, among the different modes of control, we use MIMO (Multiple Input Multiple Output) controllers because, as indicated before, we target multiple tightly-coupled goals that depend on multiple inputs [30], [34], [35].

Table I: Space of design choices from control theory.

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

Then, we consider the different organizations of MIMO controllers for multilayer systems. Decoupled or Centralized designs cannot achieve modularity and coordination simultaneously. In a Cascaded design [21], controllers are organized as a nested loop, where each controller sets the targets for the immediately inner one. Only the innermost controller changes the system inputs. This method is also suboptimal. Instead, we identify as the best choice a Collaborative architecture, where independent controllers communicate to attain coordination.

There are several approaches used to ensure that the controller works correctly under uncertainty or highly-changing conditions. The Classical one is to design controllers with additional stability margins [45]. This works for simple systems. Robust control explicitly optimizes controllers for large uncertainty, and is applicable to computer environments. The controllers have low complexity and low overheads. In Gain Scheduling, multiple controllers are used — each suited for a particular type of execution [46]. At runtime, some logic chooses when each of the controllers is active, based on the execution. This approach requires additional modeling efforts and expensive selection logic at runtime. Lastly, Adaptive control synthesizes a new controller online whenever changing conditions are detected [12]. It has higher runtime overhead.

Finally, for the controller type, PID controllers are popularly used for their simplicity, but are not useful to control MIMO systems. For MIMO systems, Linear Quadratic Gaussian (LQG) controllers [35] and Model Predictive Controllers (MPC) [34] have been proposed. However, these controllers are not natively optimized for uncertainty and, instead, trade-off optimality and fast response time for robustness. Moreover, they do not have channels to communicate among controllers. As indicated above, SSV controllers are suitable for computer system control.

III. Yukta: Multilayer SSV Control

A. Challenge: Controlling Multiple Layers

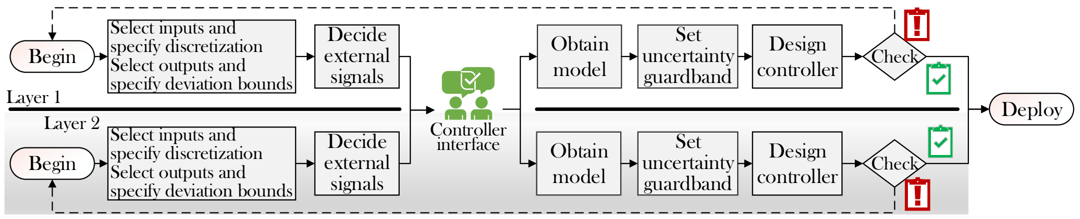
The operation of a computer involves interactions between multiple layers, including the hardware, OS, and networking layers. In such an environment, designing a single, unified formal controller that senses and actuates on signals from all the layers is both impractical and non-portable. This is because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

B. Solving the Problem with SSV Controllers

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

SSV control is suitable for multilayer computer control. To see why, compare the traits of SSV control as per Section II-B to the needs of computer systems.

First, SSV controllers take external signals. Traditionally, these signals were used by the controller to monitor "external disturbances". In computer systems, we use them to pass information from the controller of one layer to that of another layer at runtime. The second controller can use the signals to make better decisions, although it cannot control such


Begin
Select inputs and specify discretization Select outputs and specify deviation bounds
Decide external signals
Layer 1
Layer 2
Begin
Select inputs and specify discretization Select outputs and specify deviation bounds
Decide external signals
Controller interface
Obtain model
Set uncertainty guardband
Design controller
Check
Obtain model
Set uncertainty guardband
Design controller
Check
Deploy
Figure 3: Process to design a Yukta multilayer SSV controller.

signals. For example, an OS controller can pass the number of running threads as an external signal to a hardware DVFS controller.

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

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

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

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

Figure 4 shows the envisioned Yukta control system for a two-layer system. Each controller takes external signals from the other controller.

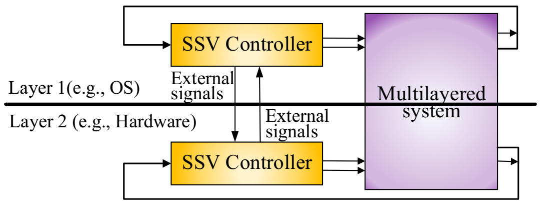


Figure 4: Yukta multilayer SSV controller.

C. *Designing SSV Controllers*

Figure 3 shows the process of designing a Yukta multilayer SSV controller. In each layer, a team initiates the design of the layer's controller by selecting the input signals and their discretized values, the output signals and their deviation bounds, and the external signals that the controller takes.

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

After this communication step, each team develops a model of the system according to their layer's perspective (possibly with the System Identification methodology [47]), sets its controller's uncertainty guardband, designs the SSV controller using MATLAB controller synthesis routines [48], and validates it. Finally, the designs of all the layers are combined, validated as a group, and deployed.

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

If the timelines of the two teams do not overlap, or close communication between teams is not desired, an approach like Figure 3 can still work, albeit less effectively. Teams can use historically-available or standard information from the other layer for their external signals. An example is how OS teams use the popular P-state interface of processors [49], [50]. Alternatively, a team can do without any extra information for their external signals. In this case, the team should increase their uncertainty guardband. This works because SSV controllers withstand inaccurate assumptions.

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

A second use is when we want some outputs (or combination thereof) to maximize or minimize their value, subject to other outputs to be within certain limits. An example is to minimize $E \times D$ subject to a power constraint. In this case, the controller needs to perform some search to find the best configuration. Hence, each SSV controller is augmented with an optimizer module (Figure 5). We discuss the operation of the optimizer in Section IV-D.

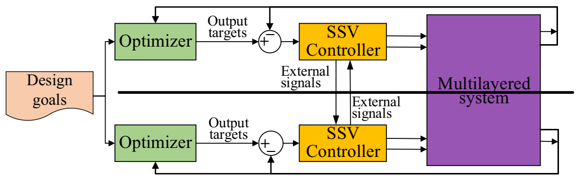


Figure 5: Yukta controller augmented with optimizers.

D. Scalability to Several Layers

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

IV. Prototyping Yukta

We prototype a multilayer SSV controller in a challenging environment: an ODROID XU3 board [51], which has an 8-core asymmetric processor running Linux. The processor is Samsung Exynos 5422, built using ARM big.LITTLE technology [52]. It has a cluster of four little cores (the in-order, low power Cortex A7), and a cluster of four big cores (the out-of-order, high performance Cortex A15). The multicore runs Ubuntu 15.04, which contains the HMP (Heterogeneous Multi-Processing) task scheduler [53], [54]. This scheduler was designed for ARM big.LITTLE platforms. The scheduler can turn cores on/off dynamically (called CPU hotplugging) based on requests from a thermal management module. Figure 6 shows a picture of our experimental platform.

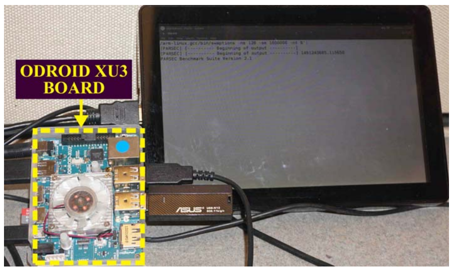


Figure 6: The ODROID XU3 used for our prototype.

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

Our choice of controllable parameters is limited by what is feasible on the board. We cannot actuate on internal structures of the processor, such as its pipeline configuration. Similarly, the HMP scheduler for big.LITTLE systems has dependencies on parts of the OS [55], [56], so we need to carefully choose what we modify. Since our goals involve minimizing $E \times D$, we also design optimizer modules for each of the controllers. The resulting system is shown in Figure 7. In the following sections, we consider each controller in turn.

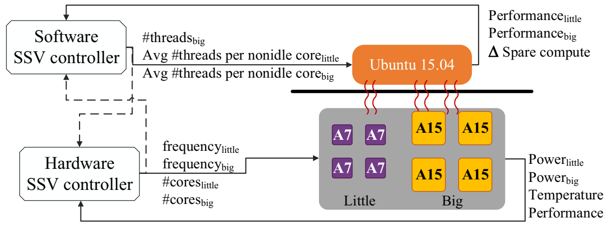


Figure 7: Prototyped controllers on the ODROID XU3.

A. Designing a Hardware Controller

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

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

Additionally, the absolute values of these weights determine the aggressiveness of the controller response. High absolute weights produce a sluggish controller, which changes the inputs only slowly when the outputs are perturbed from their target value. Low input weights produce an eager controller, which changes the inputs quickly. Neither extreme is desirable in processor control. We perform a sensitivity analysis of weight values in Section VI-E3. Based on that, we set all the weights to 1 (Table II).

The hardware controller monitors four system outputs: the performance of the workload measured in total billions of instructions committed per second (BIPS), the big cluster power, the little cluster power, and the hot-spot temperature. To set the bounds of the output deviations, we proceed as follows. When we characterize the processor with a training set of applications to build the system model (Section IV-C),

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

Goal	Inputs		Outputs		External Signals	Uncertainty
	Signals	Weights	Signals	Bounds		
Minimize $E \times D$ subject to $\text{Power}_{big} < \text{Power}_{big}^{max}$, $\text{Power}_{little} < \text{Power}_{little}^{max}$, and $\text{Temp} < \text{Temp}^{max}$	#big cores #little cores frequency_{big} $\text{frequency}_{little}$	1 1 1 1	Performance Power_{big} Power_{little} Temp	±20% ±10% ±10% ±10%	$\text{\#threads}_{big}$, avg #threads per non-idle core in cluster_{big}, and avg #threads per non-idle core in cluster_{little}	±40%

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

Goal	Inputs		Outputs		External Signals	Uncertainty
	Signals	Weights	Signals	Bounds		
Minimize $E \times D$	$\text{\#threads}_{big}$ Avg #threads per non-idle core in cluster_{big} Avg #threads per non-idle core in cluster_{little}	2 2 2	$\text{Performance}_{little}$ Performance_{big} Δ $\text{SpareCompute}_{big-little}$	±20% ±20% ±20%	#big cores, #little cores, frequency_{big}, and $\text{frequency}_{little}$	±50%

we record the range of values exhibited by each output. We then set the bounds to be a percentage of such range.

Of the four outputs considered, the power of both clusters and the temperature are critical for the integrity of the board. Hence, we assign them a bounds range that is ±10% of their maximum range; for the performance, since it is less critical, we assign a ±20% bounds range. The synthesis routines inform the designer when tighter bounds than those specified can be achieved. Alternatively, if any of these bounds is too tight, the MATLAB SSV controller synthesis routines will fail to build the controller. At runtime, the controller keeps the deviations of all outputs within these bounds for feasible targets. If it cannot, it keeps the deviations at least proportional to their relative bounds values as given by the designer. We perform a sensitivity analysis of bounds ranges in Section VI-E1.

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

Finally, as we generate the SSV controller, we need to provide an uncertainty guardband. Uncertainty is the result of limitations in how the model describes the real system, and of unpredictability in various system components. An example of the latter is aspects of the HMP scheduler, which sometimes packs multiple threads on a core while leaving another core idle. In Section VI-E2, we evaluate several uncertainty guardbands for the hardware controller. Based on that, we pick a guardband of ±40%. If the guardband is too large, MATLAB routines cannot build a controller that can deliver the output deviation bounds. If the guardband is too small, the controller will report so at runtime.

In contrast to these few intuitive parameters, industry-grade heuristic controllers have an order of magnitude more parameters. For example, in the Samsung Exynos 5422 hardware we use, to change the big cluster frequency based on the current temperature, there are many thresholds (each with its own rule) [57], [58], [59]. These rules are used to assess the impact of the temperature, detect whether temperature is rising or falling, and then change the big cluster frequency. Furthermore, to control all the four hardware outputs (i.e., performance, power of big and little clusters, and temperature), the Samsung Exynos 5422 uses several tens of interdependent settings that require tuning. Our approach eliminates the need for this extensive tuning.

B. Designing a Software Controller

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

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

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

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

$$SC = \#idle\_cores\_on - (\#threads - \#cores\_on) \quad (2)$$

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

To coordinate with the hardware controller, the software controller takes as external signals all the signals that the hardware controller actuates on. Finally, the uncertainty guardband used for the software controller should be higher than that of the hardware controller. This is because the main action of the software controller (i.e., assign threads to cores) is directly affected by an unpredictable event: dynamic changes in the number of application threads. After evaluating several uncertainty guardbands (not shown in the paper), we set the guardband value to ±50%.

C. Modeling the Controlled System

The process of designing a controller requires that we build a model of the controlled system — i.e., the ODROID board. To build the models for both controllers, we use the System Identification methodology [47]. This black-box methodology involves running a training set of applications on the ODROID board, while setting the signals that would be actuated by the controller (i.e., the inputs) and the external signals in a variety of ways, and recording the changes in the signals that would be observed by the controller (i.e., the outputs). Then, the input and output data is passed to MATLAB, which uses the Box-Jenkins polynomial model [60] to obtain a dynamic model of the system. The model generated for both controllers has dimension four — i.e., it predicts the value of an output at time *T* as a function of the values of all the outputs at times *T-1*, ... *T-4*, and the values of all the inputs at times *T*, ... *T-3*. The system identification methodology is widely used, and captures many subtleties of the input-output dependencies.

D. Designing Optimizers

The goal of each of the optimizers is to provide increasingly better targets for the output signals, so that the corresponding controller can tune the input signals (Figure 5). To see how they operate, consider the hardware controller. The optimizer reads the outputs of the system (Perf, Power$_{big}$, Power$_{little}$, and Temp), computes the resulting $E{\times}D$, and changes the output targets passed to the controller (Perf$_0$, Power$_{big\_0}$, Power$_{little\_0}$, and Temp$_0$) to attain a lower $E{\times}D$. This will trigger the controller to actuate on the input signals (#big cores, #little cores, freq$_{big}$, and freq$_{little}$) so the system converges to the new output targets. The optimizer will then read the new outputs and repeat the process, progressively generating better targets that produce lower $E{\times}D$ values.

Recall that $E{\times}D$ is proportional to Power/Perf^2. Hence, to lower $E{\times}D$, the optimizer keeps increasing Perf$_0$ a lot while increasing Power$_{big\_0}$ and Power$_{little\_0}$ a little. When the result is that $E{\times}D$ has increased, the optimizer discards the latest move, and moves in the opposite direction: it decreases Perf$_0$ a little while decreasing Power$_{big\_0}$ and Power$_{little\_0}$ a lot. Eventually, the optimizer settles into a desirable set of output targets.

V. Experimental Methodology

A. Infrastructure

The ODROID XU3 has on-board power sensors that measure the power drawn by the big and little clusters. These sensors update every 260 ms. There are on-chip sensors that measure temperature. We set up performance counters on all cores using the Linux *perf* API [61] to measure the number of instructions committed per second. The number of cores in each cluster and the cluster frequency can be changed by writing to appropriate *cpufreq* files. Thread scheduling is performed through *sched_setaffinity* system calls.

The controllers are invoked every 500 ms. This time is determined by the update period of the power sensors. Many prior works that use real systems use comparable sampling intervals (e.g., 0.5 s – 2 s in [10], [11], [24]). Both controllers are implemented as independent privileged processes, as we cannot add hardware modules to the board.

The power and temperature limits that we use in our evaluation are constrained by the emergency power and thermal heuristics of the board. These heuristics are automatically triggered when power or temperature increase beyond preset thresholds for extended periods of time [57], [58], [59]. We identify the minimum thermal threshold that triggers these heuristics and use it as the limit for temperature. Similarly, we set the limits for the power consumed by the little and big clusters to be below the emergency-triggering values. The limits we use are 0.33 W, 3.3 W, and 79 °C for the power of the little cluster, power of the big cluster, and temperature.

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

B. Schemes for Comparison

In our ODROID XU3 board, we implement the four two-level controllers shown in Figure 8. Table IV lists their names and describes them in detail.

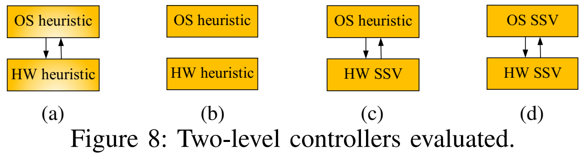
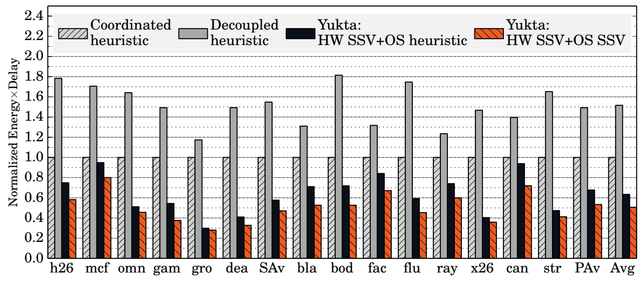


Figure 8: Two-level controllers evaluated.


(a) Normalized Energy×Delay

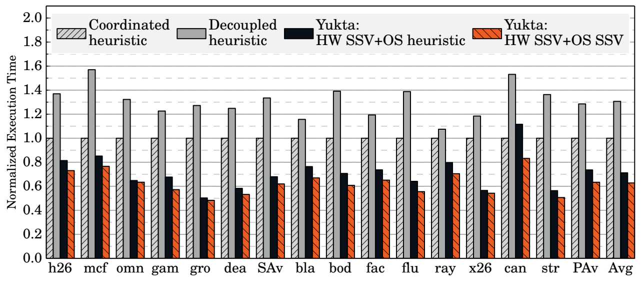

(b) Normalized execution time

Figure 9: Energy×Delay (a), and execution time (b) for the four two-layer controller schemes considered.

Table IV: Description of the controllers.

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

In the *Coordinated heuristic* scheme, the OS controller is similar to the HMP task scheduler from ARM, Linaro and Samsung [53], [54], except that it is modified to optimize E×D. The OS controller coordinates with the hardware controller in that it uses the number, type, and frequency of the available cores to schedule threads. The hardware controller sets the number of cores and their frequency to maximum values until the power or temperature exceeds the limits; when this happens, it finds a lower, safe frequency value for that cluster. It coordinates with the OS controller in that it uses how the threads are distributed across all the cores to determine the safe frequency. This OS-hardware scheme is representative of industry-standard controllers in big.LITTLE systems, and we use it as a baseline.

The *Decoupled heuristic* scheme takes uncoordinated decisions at each layer. The OS controller distributes threads on cores in a round-robin manner. The hardware controller is similar to the *Performance* power governor in Linux [62]. It sets the number of cores and their frequency to maximum values whenever temperature and power are within limits. When the limits are exceeded, it uses threshold-based rules to temporarily reduce frequency first, and then the number of cores — irrespective of the number of threads.

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

VI. Evaluation

A. Multilayer Controller Evaluation

Figure 9 compares our four two-layer controller schemes running our applications. Figure 9(a) shows the $E \times D$ of the applications, and Figure 9(b) the execution time. In each chart, the bars from left to right correspond to individual SPEC applications, the average of the SPEC applications (*SAv*), individual PARSEC applications, the average of the PARSEC applications (*PAv*), and the average of all the applications (*Avg*). Each application has a bar for each of the four controller schemes. The bars are normalized to *Coordinated heuristic*.

Figure 9(a) shows that *Decoupled heuristic* has higher $E \times D$ than *Coordinated heuristic*. On average, decoupling the controllers results in a 52% higher $E \times D$. On the other hand, using Yukta causes $E \times D$ to decrease. On average, *Yukta: HW SSV+OS heuristic* has a 37% lower $E \times D$ than *Coordinated heuristic*. Furthermore, having both SSV controllers as in *Yukta: HW SSV+OS SSV* results in an average $E \times D$ that is 50% lower than *Coordinated heuristic*. Thus, SSV controllers offer substantial improvements over existing systems.

The execution times in Figure 9(b) show similar results. *Decoupled heuristic* increases the execution time by 30% on average. On the other hand, *Yukta: HW SSV+OS SSV* reduces the time by 29% on average, and *Yukta: HW SSV+OS SSV* by even more, namely a substantial 38% on average.

To gain insight into the impact of the Yukta controllers, we focus on the execution of the blackscholes application (labeled *bla* in Figure 9). This application begins with a single thread and later executes 8 parallel threads. The work in the parallel phase does not have large variations. Figure 10 shows the power consumed by the big cluster in blackscholes as a function of time, for the four controller schemes. Recall that the limit in sustained power is 3.3W.

The figure shows that, under all schemes, the power fluctuates. At certain points, it goes over the limit but, immediately after, the system reacts and brings the power down again. What varies between the four schemes is the

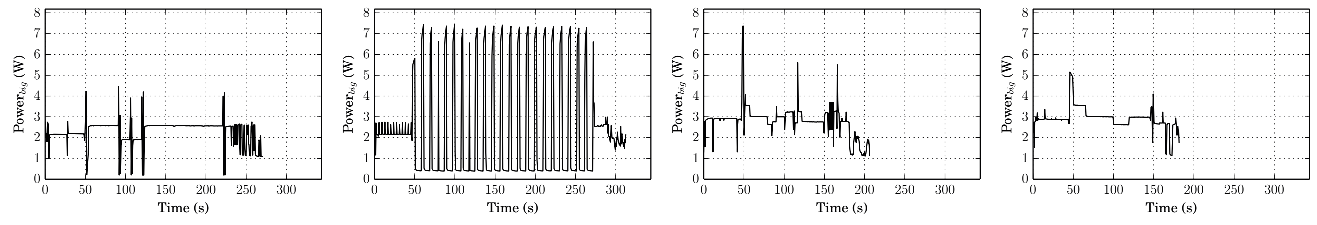

(a) *Coordinated heuristic* (b) *Decoupled heuristic* (c) *Yukta: HW SSV+OS heuristic* (d) *Yukta: HW SSV+OS SSV*

Figure 10: Power consumed by the big cluster in blackscholes as a function of time for the four controller schemes.

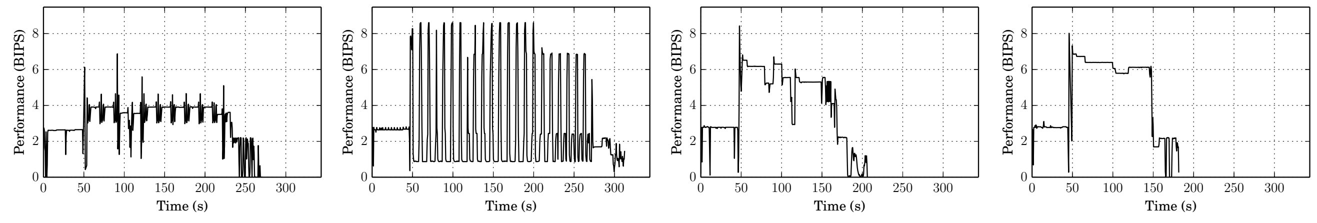

(a) *Coordinated heuristic* (b) *Decoupled heuristic* (c) *Yukta: HW SSV+OS heuristic* (d) *Yukta: HW SSV+OS SSV*

Figure 11: Performance of blackscholes in BIPS, as a function of time for the four controller schemes.

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

In *Decoupled heuristic* (Figure 10(b)), there are many oscillations. In this scheme, the hardware controller increases the number of cores and their frequency to the maximum, while the OS controller simply assigns threads round-robin. This causes the power to go over the limit and trigger the emergency system, which reduces the frequency of the cores and shuts off some cores. The power then drops to low values, and the hardware controller again increases the number of cores and their frequency to the maximum. The result is continuous power oscillation.

The *Coordinated heuristic* scheme (Figure 10(a)) drastically reduces the amplitude and number of these peaks and valleys. This is thanks to the coordination between the two controllers: the hardware controller knows the distribution of the active threads, and the OS controller knows the number, type, and frequency of the active cores.

As we move to *Yukta: HW SSV+OS heuristic* (Figure 10(c)) and, especially, *Yukta: HW SSV+OS SSV* (Figure 10(d)), the number of peaks and valleys decreases. Moreover, the power during the steady-state periods gets closer to 3.3W. The Yukta controllers control power much better.

These differences in power control translate directly into different performance. Figure 11 shows the performance of blackscholes in BIPS, as a function of time, for the four schemes. We see that the performance of the *Decoupled heuristic* scheme (Figure 11(b)) oscillates, and the application takes nearly 320 seconds to complete. In the *Coordinated heuristic* scheme (Figure 11(a)), the steady-state performance increases, and the application completes in 270 seconds. Finally, in *Yukta: HW SSV+OS heuristic* (Figure 11(c)) and *Yukta: HW SSV+OS SSV* (Figure 11(d)), the steady-state performance keeps increasing, and the application completes sooner, in 205 and 180 seconds, respectively.

B. Comparing to LQG Control

We compare Yukta to the recently-proposed state-of-the-art MIMO LQG (Linear Quadratic Gaussian) controller [35]. Like Yukta, such controller can change many inputs to meet many output targets, and accepts weights for inputs and outputs. Unlike Yukta, however, it does not accept External signals, deviation bounds for outputs, saturation/quantization of inputs, or design with uncertainty guardbands.

Since an LQG controller cannot use External Signals, we evaluate the two ways in which it can be used for multilayer control: one that has independent LQG controllers in the hardware and OS layers (*Decoupled HW LQG+OS LQG*), and one that has a single LQG controller that manages both layers (*Monolithic LQG*). The latter is the use in [35]. We use input and output weights comparable to our SSV controllers.

Figures 12 and 13 compare the $E \times D$ and execution time, respectively, of *Coordinated heuristic*, *Decoupled HW LQG+OS LQG*, *Monolithic LQG*, and *Yukta: HW SSV+OS SSV*. The bars are normalized to *Coordinated heuristic*. We see that, on average, *Decoupled HW LQG+OS LQG* delivers $E \times D$ and performance similar to *Coordinated heuristic*. This is because each controller works independently without coordination, making the system inefficient.

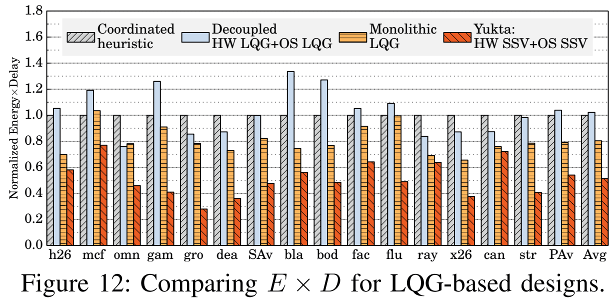


Figure 12: Comparing $E \times D$ for LQG-based designs.

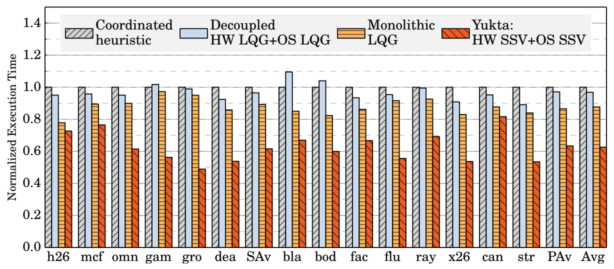


Figure 13: Comparing performance for LQG-based designs.

Monolithic LQG delivers better results, thanks to the centralized decisions taken by the controller. On average, it reduces $E \times D$ by 20% and execution time by 11% relative to *Coordinated heuristic*. This is consistent with the 16% $E \times D$ reduction reported in [35]. However, these gains are small compared to those of *Yukta: HW SSV+OS SSV*, which attains average reductions of 50% in $E \times D$ and 38% in execution time.

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

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

Finally, LQG controllers are not natively optimized for uncertainty. The framework that generates LQG controllers uses uncertainty guardbands to discard unstable designs [35], [41]. When this happens, it changes the output weights, which slows down the controller. In the framework that generates SSV controllers, instead, uncertainty is an explicit parameter. Hence, the resulting controller is optimal within the uncertainty guardband. Overall, LQG controllers are no match for Yukta designs.

C. Heterogeneous Workloads

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

Figure 14 compares the normalized $E \times D$ of these workloads under all the heuristic, LQG and Yukta-based designs we built. The results are similar to the homogeneous workloads, with the Yukta-based designs exhibiting the lowest $E \times D$, then *Monolithic LQG*, and then *Coordinated heuristic*. The reduction in *Yukta: HW SSV+OS SSV* is 47%, which is close to the 50% attained before. This demonstrates the robustness of the Yukta-based designs in diverse environments.

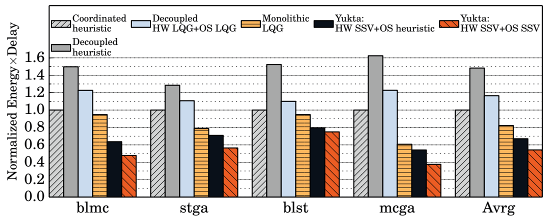


Figure 14: Comparing $E \times D$ for heterogeneous workloads.

D. Implementing a Hardware SSV Controller

A hardware implementation of our hardware SSV controller is a simple state machine. It is characterized by the dimensionality of its state (N), and the number of inputs (I), outputs (O), and external signals (E). It implements the following equations in hardware [30]:

$$x(T+1) = A \times x(T) + B \times \Delta y(T) \quad (3)$$

$$u(T) = C \times x(T) + D \times \Delta y(T) \quad (4)$$

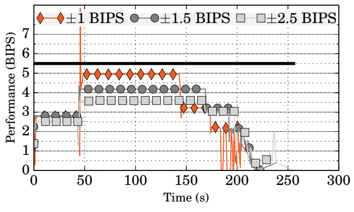
where x is the state of the controller (N-entry vector), Δy is the external signals and the deviation of outputs from their targets (vector of O+E entries), u is the new inputs (I-entry vector), A is the controller evolution matrix (N×N), B is the matrix of impact of output deviations on the state (N×(O+E)),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

E. Hardware SSV Controller Analysis

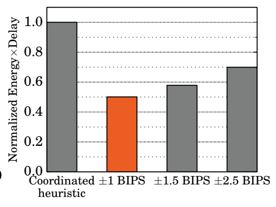
1) Analysis of Output Deviation Bounds: The hardware controller of *Yukta: HW SSV+OS SSV* in Section IV has deviation bounds of ±20% for its performance output (i.e., ±1 BIPS in absolute terms). In this section, we change them to ±30% (i.e., ±1.5 BIPS) and ±50% (i.e., ±2.5 BIPS). Since the OS controller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

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

Figure 15(a) shows the performance of the computer system as a function of time for the three output deviation bounds (absolute values of bounds are shown for convenience). The data is for blackscholes. Ignoring the initialization and termination stages, we see that the performance remains close to the target, and within the deviation bounds. The tighter the bounds are, the closer the performance is to the target.



(a) Tracking 5.5BIPS of performance.



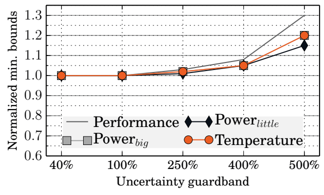
(b) Minimizing E×D.

Figure 15: Sensitivity to the output deviation bounds.

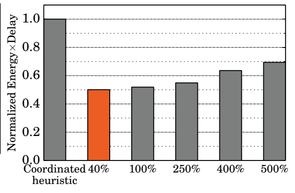
The second experiment is like the one in Section VI-A, where we minimize $E \times D$. Figure 15(b) shows the $E \times D$ of *Yukta: HW SSV+OS SSV* for the different output deviation bounds (absolute values of bounds shown for convenience), and of *Coordinated heuristic*. The bars are the average of all the applications, and are normalized to *Coordinated heuristic*.

We see that the $E \times D$ with deviation bounds of ±20% (±1 BIPS), ±30% (±1.5 BIPS), and ±50% (±2.5 BIPS) is 50%, 41%, and 30% lower than with *Coordinated heuristic*, respectively. As bounds grow wider, the execution is less optimal: output changes that would cause a controller with tight bounds to actuate, do not cause a controller with loose bounds to actuate.

2) Analysis of Uncertainty Guardband: We examine uncertainty guardbands from ± 40% to ± 500%. Figure 16(a) shows how the output deviation bounds guaranteed by the controller change with different uncertainty guardbands. These bounds are normalized to those in Section IV-A, namely ±20% for performance, and ±10% for the rest.



(a) Normalized output bounds.



(b) E×D.

Figure 16: Sensitivity to the uncertainty guardband.

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

Figure 16(b) shows E×D for different uncertainty guardbands, all normalized to *Coordinated heuristic*. For ± 40% guardband, E×D is 50% lower than the baseline. For large guardbands, E×D increases for two reasons. First, the controller is slower to respond to the optimizer-generated targets. Second, the output bounds grow larger, which causes the controller to work less effectively. Overall, we use ± 40% as our default guardband.

3) Analysis of Input Weights: We examine input weights from 0.5 to 2 for all the inputs. This results in controllers that respond at different speeds to output changes. In our experiment, we consider the big cluster power output and set its target value to 2.5W. Figure 17 shows the big cluster power as a function of time for the different input weights. The data corresponds to blackscholes.

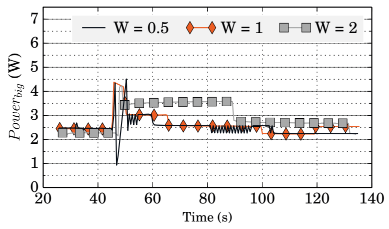


Figure 17: Big cluster power for different input weights.

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

VII. Other Related Work

Expanding on Sections I and II, we discuss additional aspects of prior work on multilayer control. We consider controller organization (Table I) and methodology.

Organization: Decoupled controllers are simple to design and modular, while monolithic controllers can achieve better results due to a global view of the system. However, decoupled techniques can exhibit destructive interference between controllers, even at a single layer [7], [14], [21], [24], [35], [63]. In turn, monolithic techniques have design complexity and are less maintainable, scalable, or portable [7], [8], [10], [11], [14], [21].

Some designs use decoupled controllers that are coordinated implicitly by the use of controller ranking [8], [11], [12], [25], [26]. In these designs, each controller runs at a slower timescale than its immediately higher-ranked one.

The highest-ranked controller adjusts one resource first, to attain the most important goal. Each subsequent lower-ranked controller modifies a different resource, to meet a different goal. Bhattacharya et al. [64] and Maggio et al. [34] show that such designs may have responsiveness and stability issues.

Graybox [16] creates middleware that exposes useful OS information to the application to coordinate software controllers. Other authors argue for collaborative control [14]. However, as we see next, most of the existing collaborative controllers (e.g., [13], [14], [27]) have limitations.

Methodology: Multilayer controllers can be based on heuristics, control theory, machine learning, or optimization theory. Many works rely on heuristics to modify parameters and coordinate controllers [8], [13], [14], [27]. However, researchers have shown how heuristics can fail [12]. Other designs use a combination of heuristics and control theory [11], [21], heuristics and optimization [65], or just optimization [66]. The xTune framework [18] uses Monte Carlo simulations at runtime to pick the statistically-best action from a list of designer-specified actions. Some designs use a combination of machine learning and PID control [12], [67]. Muthukaruppan et al. [10] use price theory for big.LITTLE systems.

VIII. Conclusion

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

Acknowledgments

This work was supported in part by NSF under grants CCF-1536795 and CCF-1649432.

References

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

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

[3] S. Jahagirdar, V. George, I. Sodhi, and R. Wells, "Power Management of the Third Generation Intel Core Micro Architecture formerly Codenamed Ivy Bridge," in *Hot Chips*, Aug. 2012.

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

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

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

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

[8] H. Zhang and H. Hoffmann, "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques," in *ASPLOS*, 2016.

[9] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration," in *ISCA*, 2006.

[10] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price Theory Based Power Management for Heterogeneous Multi-cores," in *ASPLOS*, 2014.

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

[12] H. Hoffmann, "JouleGuard: Energy Guarantees for Approximate Applications," in *SOSP*, 2015.

[13] J. Flinn, E. de Lara, M. Satyanarayanan, D. S. Wallach, and W. Zwaenepoel, "Reducing the Energy Usage of Office Applications," in *Middleware*, 2001.

[14] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile Application-aware Adaptation for Mobility," in *SOSP*, 1997.

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

[16] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Information and Control in Gray-box Systems," in *SOSP*, 2001, pp. 43–56.

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

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

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

[20] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated Power-performance Optimization in Manycores," in *PACT*, 2013.

[21] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center," in *ASPLOS*, 2008.

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

[23] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "A

Taxonomy of Compositional Adaptation," Michigan State University, Tech. Rep., 2004.

[24] 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 *MICRO*, 2013.

[25] A. Filieri, H. Hoffmann, and M. Maggio, "Automated Multi-objective Control for Self-adaptive Software Design," in *ESEC/FSE*, 2015.

[26] S. Shevtsov and D. Weyns, "Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems," in *FSE*, 2016, pp. 229–241.

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

[28] 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.

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

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

[31] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron, "Control-theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads," in *CASES*, 2002.

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

[33] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable Power Control for Many-core Architectures Running Multi-threaded Applications," in *ISCA*, 2011.

[34] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann, "Automated Control of Multiple Software Goals Using Multiple Actuators," in *ESEC/FSE*, 2017.

[35] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures," in *ISCA*, June 2016.

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

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

[38] F. Zanini, C. Jones, D. Atienza, and G. De Micheli, "Multicore Thermal Management using Approximate Explicit Model Predictive Control," in *ISCAS*, May 2010.

[39] 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 *DATE*, Mar. 2011.

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

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

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

[43] S. Skogestad, M. Morari, and J. C. Doyle, "Robust control of ill-conditioned plants: high-purity distillation," *IEEE Trans. Autom. Control*, vol. 33, no. 12, pp. 1092–1105, Dec. 1988.

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

[45] 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 *PACT*, 2017.

[46] A. M. Rahmani, B. Donyanavard, T. Müch, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt, "SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management," in *ASPLOS*, 2018.

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

[48] *MATLAB and Robust Control Toolbox Release 2016a*. Natick, Massachusetts: The MathWorks Inc., 2016.

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

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

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

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

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

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

[55] U. Rezki and V. Wool, "Doing big.LITTLE right: little and big obstacles," in *ELC*, Mar. 2015.

[56] M. Anderson, "Implementation of the Global Task Scheduler in big.LITTLE Android Platforms," in *ELC*, Mar. 2015.

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

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

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

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

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

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

[63] R. Bitirgen, E. İpek, and J. F. Martínez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach," in *MICRO*, 2008.

[64] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar, "The Need for Speed and Stability in Data Center Power Capping," in *IGCC*, 2012.

[65] C. J. Hughes and S. V. Adve, "A Formal Approach to Frequent Energy Adaptations for Multimedia Applications," in *ISCA*, 2004.

[66] 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 *RTSS*, 2007.

[67] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, "CALOREE: Learning Control for Predictable Latency and Low Energy," in *ASPLOS*, 2018.