Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures

Raghavendra Pradyumna Pothukuchi, Amin Ansari,* Petros Voulgaris, and Josep Torrellas
University of Illinois at Urbana-Champaign **Qualcomm Research*
http://iacoma.cs.uiuc.edu

***Abstract*—As processors seek more resource efficiency, they increasingly need to target multiple goals at the same time, such as a level of performance, power consumption, and average utilization. Robust control solutions cannot come from heuristic-based controllers or even from formal approaches that combine multiple single-parameter controllers. Such controllers may end-up working against each other. What is needed is control-theoretical MIMO (multiple input, multiple output) controllers, which actuate on multiple inputs and control multiple outputs in a coordinated manner.**

In this paper, we use MIMO control-theory techniques to develop controllers to dynamically tune architectural parameters in processors. To our knowledge, this is the first work in this area. We discuss three ways in which a MIMO controller can be used. We develop an example of MIMO controller and show that it is substantially more effective than controllers based on heuristics or built by combining single-parameter formal controllers. The general approach discussed here is likely to be increasingly relevant as future processors become more resource-constrained and adaptive.

***Keywords*-Architectural control; Control theory; Tuning.**

I. Introduction

There is an urgent need to make computer systems, and processors in particular, increasingly resource efficient. To be competitive, new systems need to consume less power, use the battery better, operate at lower temperatures, and generally avoid inefficient conditions. At the same time, processors are built to be more adaptive, with C and P states and extensive power management [1], [2], [3], [4], [5], and there are a myriad of reconfiguration capabilities under research (e.g., [6], [7], [8], [9]).

In this environment, as a program runs, we increasingly want to target multiple goals *at the same time* — e.g., a certain application frame rate, power consumption level, and average core utilization. To accomplish this, designers typically develop control algorithms that try to enforce all the constraints (e.g., [10]). Such algorithms use heuristics, have rules and thresholds, and are often organized in multiple loops — i.e., an inner loop that optimizes one measure, surrounded by a second loop that optimizes the next measure, and so on [2].

Unfortunately, this ad-hoc or heuristic-based approach is generally not robust. First, it is not obvious how to build the unified algorithm. For example, suppose that, to attain the targets above, we can move jobs across cores and change each core's frequency, issue width, and load/store queue size. It is unclear by how much and in what order we need to change each of these parameters. Second, this approach requires developing complex algorithms, which may end-up being buggy. As the algorithm executes, there is the danger that unexpected corner cases cause large deviations from the targets.

The alternative is to design software or hardware controllers using control theory [11]. With control theory, the designer has a systematic way to quantify the knowledge and expectations he has about the design. For example, he knows how important each of the objectives is relative to each other. He also knows the overhead of actuating on different inputs. Hence, he can provide this information, and then, proven methodologies generate a controller that actuates on the most convenient and fastest inputs to safely converge to the desired targets.

Control theory has been used to design controllers that tune architectural parameters in processors (e.g. [2], [9], [12], [13], [14], [15], [16], [17]). However, to the best of our knowledge, in all architectural proposals, a controller controls only a single output — e.g., the power, performance, or utilization. Hence, these designs do not address the general case where we want to control multiple outputs in a coordinated manner. As a result, in a processor with multiple controllers, each controlling a single output, we run the risk that the separate controllers end-up working against each other.

What is needed is what control theory calls MIMO control: a controller that actuates on multiple system inputs and controls multiple interdependent system outputs [11]. For example, a MIMO controller can actuate on a processor's frequency, issue width, and load/store queue size (system inputs) to ensure that both the frame rate and power (system outputs) attain certain target values. This approach ensures the effective control of interdependent measures and, we argue, will be key as processors become more resource-constrained and adaptive.

In this paper, we use MIMO control-theory techniques to develop controllers to dynamically tune architectural parameters in processors. To our knowledge, this is the first work in this area. We give architectural intuition for the different procedures involved. We discuss three ways in which a MIMO hardware controller can be used. Specifically, a basic use is to target a fixed set of reference values of multiple outputs. A second use is to target a set of reference values that change based on real-time conditions. For example, to maximize battery life, a high-level agent can change the target quality of service and the target power level as a battery is being depleted. A final use is to maximize or minimize a combination of the values of the outputs — e.g., the product of energy times delay squared ($E{\times}D^2$).

We develop an example MIMO controller that actuates on the cache size, frequency, and reorder buffer size of a processor, to control both the power and the performance of the processor. We apply this controller to the three uses. We show that it is substantially more effective than controllers based on heuristics or built by combining single-output formal controllers.

II. Problem Description

In control theory, a controlled system is represented as a feedback control loop as in Figure 1(a). The controller reads the output *y* of a system in state *x*, compares it to the target value y_0 and, based on the difference (or error), generates the input *u* to actuate on the system to reduce the error.

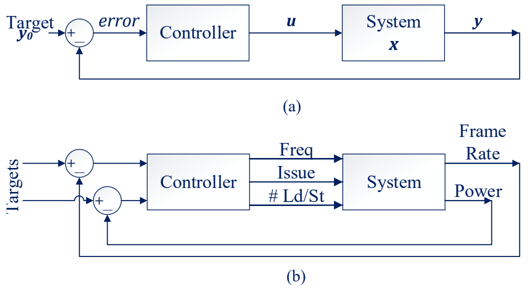


Figure 1: Typical feedback control loop.

Control theory has been used to design controllers that tune architectural parameters in processors (e.g., [2], [9], [12], [13], [14], [15], [16], [17]). To the best of our knowledge, in these proposals, a controller only controls a single output (*y*). Specifically, the large majority of these proposals use a SISO design: Single Input to the system (*u*) and Single Output (*y*) [2], [9], [15], [17]. For example, Lu et al. [9] control the frame rate of multimedia applications by changing the frequency. This is a limited approach.

The other designs use a MISO approach: Multiple Inputs and Single Output. Some of them combine multiple SISO models to generate a larger MISO controller. For example, Wang et al. [14] control the total power of a multicore by changing the frequencies of all the cores. The multicore's power is the sum of the powers of all the cores, and each core's power only depends on that core's frequency. Changing a core's frequency (input) only impacts the power of that core (output); it does not impact the power of all the other cores — at least, not directly. A similar approach is followed by others [12], [16].

One design that is intrinsically MISO is Fu et al. [13]. The authors control the utilization of a processor by changing its frequency and the size of its L2 cache. They embed this controller inside an outer loop that uses a linear programming solver to minimize power.

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

Figure 1(b) shows our goal, MIMO control: Multiple Inputs and Multiple Outputs [11]. The example controller senses the frame rate and power of a processor, compares them to their target values, and then actuates on multiple inputs (the processor's frequency, issue width, and load/store queue size) to ensure both frame rate and power targets are satisfied. Each of the inputs impacts each of the outputs.

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

III. Background

A. *Overview of MIMO Control*

We take a processor as our system, and abstract it as a controlled system as in Figure 1(a). The system is characterized by state *x*, inputs *u*, and outputs *y*. They are all a function of time *t*. The system state *x* is given as an *N*-dimensional vector. We assume we have *I* inputs (e.g., frequency, issue width, and ld/st queue size) and *O* outputs (e.g., power and frame rate). These measures are related as follows [11]:

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

$$y(t) = C \times x(t) + D \times u(t) \tag{2}$$

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

The characterization experiments also assess the unpredictability component of the system, represented as two matrices [11]. One encapsulates the non-determinism of the system, possibly caused by effects such as interrupts and unexpected program behavior changes. The other matrix encapsulates the noise in reading the outputs — e.g., due to inaccuracies in the power sensor. These two effects are shown in the augmented feedback control loop of Figure 2 (together with an uncertainty effect that we discuss later).

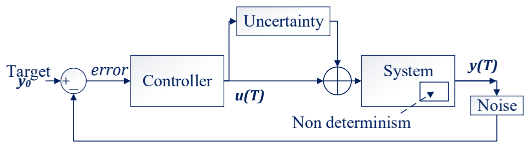


Figure 2: Augmented feedback control loop.

Table I: Comparing approaches for architecture tuning.

Approach	Problem formulation	Example	Design and tuning	Advantages	Shortcomings
Optimization [18], [19]	Minimize an objective subject to constraints. The objective is a f(inputs).	Objective: power. Constraint: IPC $>$ k.	Obtain a model of f(). Use solvers at runtime to obtain the inputs.	1) Natural choice for architecture. 2) Expressive.	1) Model needs to be close to reality and convex. 2) No feedback.
Machine learning [20], [21]	input$_i$ = max([weights]$_{v\times o}$ $\times$ [features]$_o$), where v is the # of values for input$_i$, and o is the # of features.	Input: frequency. Features: power, misses/Kinstr.	Tune weights by specifying the best set of input values and associated feature values.	1) Data driven identification of relationships. 2) Formal reasoning and methodology.	1) Hard to add feedback. 2) No guarantees. 3) Requires exhaustive enumeration during training.
Control theory [12], [14]	Change inputs to make outputs approach reference values, where outputs[t] = f(inputs[t,t-1,..],outputs[t-1,..])	Input: frequency. Output: IPC. Reference value: QoS$_0$.	Obtain the f() for outputs[t]. Specify several design requirements.	1) Provides guarantees. 2) Learns from feedback. 3) Formal reasoning and methodology.	1) Hard to obtain model. 2) Specifying reference values is not obvious.
Model-based heuristics [22], [23]	Use a model to guide decisions. The model relates outputs, inputs and auxiliary outputs.	Output: power. Aux output: misses/Kinstr. Input: frequency.	Find the model. Use insight to develop rules on top of the model. Train rules to set thresholds.	1) Model simplifies decision making.	1) No guarantees. 2) No formal methodology. 3) Hard to add learning. 4) Prone to errors. 5) Hard to deal with multiple inputs and/or outputs.
Rule-based heuristics [24], [25]	Encode in an algorithm decisions to choose inputs based on outputs and auxiliary outputs.	Output: power. Aux output: misses/Kinstr. Input: frequency.	Select rules. Train rules to set thresholds.	1) Lightweight.	1) No guarantees. 2) No formal methodology. 3) Hard to add learning. 4) Prone to errors. 5) Hard to deal with multiple inputs and/or outputs.

To control this system, we use a type of MIMO controller called Linear Quadratic Gaussian (LQG) controller [11], [26]. The LQG controller generates the system inputs, $u(t)$, based on the state of the system, $x(t)$, and the difference in the outputs of the system from their reference (i.e., target) values. However, as the system's true state is not known, the controller begins with a state estimate and generates the system inputs based on this estimate. The controller refines the estimate and learns the true state by comparing the output predicted using the state estimate and the true output. Both estimation and system input generation happen simultaneously and their accuracy increases with time. The design of the LQG controller guarantees that the estimated state converges to the unknown true state soon and, therefore, the appropriate input values are generated to stabilize the system's outputs at their target values quickly.

We use LQG control because it fits the requirements present in architectural control. Specifically, an LQG controller tries to minimize the sum of the squares of a set of costs (also called errors). Such errors are the differences between each output and its reference value, and between each input and the proposed new value of that input — the controller minimizes input changes to avoid quick jerks from steady state. These errors can be given architectural meanings. Moreover, the designer can add weights to each of these errors: the higher the weight, the more important is for that error to be small. For example, the designer can give a high weight to power errors. These weights are given by the designer in two positive diagonal matrices [11]: the Tracking Error Cost matrix (*Q*) for the outputs, and the Control Effort Cost matrix (*R*) for the inputs. We give the architecture insights in Section IV.

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

Since a processor is a very complex system, even models generated by experimenting with many applications will be incorrect in some cases. Hence, we add an Uncertainty factor to the model. In practice, this means adding an additional guardband to the parameters generated by the control theory calculations. This is shown in Figure 2 as an extra path that perturbs the system in a random way.

Then, we perform Robustness Analysis [11] to ensure that the controller will work correctly with this level of uncertainty. Robustness Analysis is a type of mathematical analysis that analyzes every type of uncertainty that is possible in the system (e.g., non-linearity or time variance) and, for a given bound on the size of this uncertainty, determines whether the system will be stable.

B. Comparing Approaches for Architecture Adaptation

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

Most of the entries in the table are self explanatory. From the advantages and shortcomings columns, we see that the control theory approach is the only one that: 1) uses feedback to learn automatically at runtime, and 2) provides three guarantees. We discuss the feedback difference more in Section IX. The guarantees are *Convergence*, *Stability*,

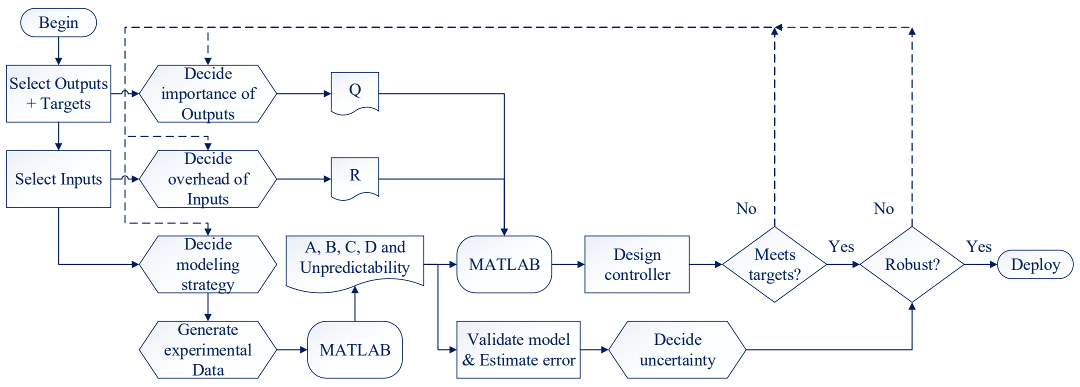


Figure 3: Proposed flowchart of the process of building a MIMO controller. Architecture-level insight is especially needed in the hexagonal steps.

and *Optimality* [11]. Informally, these guarantees mean the following. Convergence means that, if it is possible for the outputs to take the reference (i.e., target) values, then they will eventually reach them. Stability means that, once the outputs converge to their reference or to the closest they can get to them, then they exhibit no oscillatory behavior. Optimality means that the final state of the system is optimal according to the cost function specified by the architect.

In spite of control theory's advantages, at least two issues have limited its use in architecture. First, it needs a model that gives the current output values as a function of the current input values and some history of input and output values. This is hard to obtain analytically. Second, control theory approaches assume that the reference values for the outputs are specified by a higher entity. This might be easy in some cases, such as when targeting a Quality of Service (QoS) requirement, but it is not so when trying to optimize a metric — e.g., $E{\times}D^2$. We address these issues later.

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

IV. MIMO Controllers for Processors

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

A. Steps in Controller Design

Figure 3 shows the proposed process to build a MIMO controller, with hexagons showing steps that need architectural insight. First, we select the outputs to be controlled and the inputs that can be manipulated. Then, using architectural insights, we decide on the relative importance of the outputs (to generate the Q matrix), the relative overheads of the inputs (to generate the R matrix), and the strategy for modeling the system. The latter mainly involves choosing how to model the system (analytically or experimentally) and the number of dimensions of the system state x.

We model the system experimentally, performing the experiments for black-box system identification [27], which we describe later. We pass the experimental data to a least square solver for a dynamic environment (running in MATLAB) and generate the A, B, C, D, and two unpredictability matrices. These matrices constitute the model. We pass this model plus the Q and R matrices to a constraint optimization solver (also running in MATLAB) to generate the controller. The controller is encoded as a set of matrices that can produce the changes in the manipulated inputs on observing tracking errors in the controlled outputs.

Next, we validate the model by running additional programs on both the model and the real system. Based on the observed differences, we estimate the model error and, using architectural insights, we set the Uncertainty of this model. The uncertainty will be used in the validation step.

Finally, we proceed to validating the controller in two steps. First, we check if the controller meets the targets (i.e., it brings the outputs to the targets and does it fast enough). Then, we use Robust Stability Analysis [11] to see if, for the worst case of estimated uncertainty, the system is still stable. We describe this process later. If any of these two checks fail, we change the initial decisions and repeat the process. Otherwise, we have the final controller.

B. Detailed Controller Design

1) Modeling the System: While there are analytical models of processor performance and power [28], [29], [30], [31], they do not capture the dynamics of the system at the level we need, and are not amenable for formal controller design. To control the system, we need models that describe the relation between inputs and outputs as a function of time. Hence, we build an experimental model using the black-box identification technique of System Identification Theory [27], [32]. We apply waveforms with special patterns at the inputs of the system, and monitor the waveforms at the outputs. We then assume that the outputs at time t (i.e., $y(t)$) depend on the outputs at the previous k time steps ($y(t\text{-}1)$, ...$y(t\text{-}k)$), the inputs at the current and previous $l\text{-}1$ time steps ($u(t)$, ...$u(t\text{-}l\text{+}1)$), and a noise term. We pass the waveforms to a least square solver for a dynamic environment (running in MATLAB) and obtain the A, B, C, D, and unpredictability matrices. Since system identification is well known, we do not describe it further.

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

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

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

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

Type of Weight	Qualitative Weight Ranking (From High to Low)
System Outputs	Voltage guardband, Temperature, Power, Core Utilization, Energy, Frame rate, Instructions per Second (IPS)
System Inputs	Cache power gating, core power gating, frequency, issue width, ld/st queue entries

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

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

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

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

Figures 4(a) and (b) illustrate the two cases. Each figure shows how the output (top) and input (bottom) change with time, relative to the initial conditions. The time starts when the output suffers some positive noise. Figure 4(a) shows a ripply system: the input immediately reacts, causing the output to change course and get into negative values. After a few waves, the system stabilizes again. Figure 4(b) shows a system with inertia: the input does not react until much later, and with lower intensity. After a while, both output and input return to the stable values. Note that the figures are not drawn to scale.

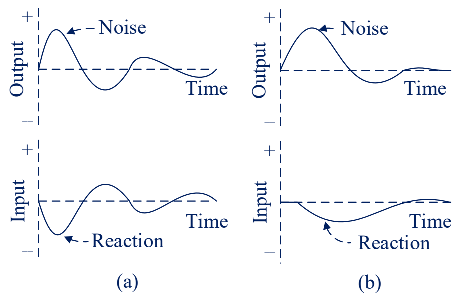


Figure 4: System behavior when input weights are low (a) or high (b) relative to output weights.

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

The absolute values of the weights are unimportant; only their relative value matters. The designer uses offline experimentation and his architectural intuition to set them appropriately. As an example, consider a weight of $100\times$ for one output (o_1) over the other (o_2). The relative quadratic cost of a tracking error Δ in these outputs becomes $100\Delta o_1^2$ and Δo_2^2. If they are to matter equally, then we have $100\Delta o_1^2 = \Delta o_2^2$, or $10\Delta o_1 = \Delta o_2$. This means that the controller will not deviate from the reference value for o_1

by 1% unless it can reduce the deviation for o_2 by 10%. Applying a similar analysis for inputs, a relative input weight of 100 means that the controller will change the lower-weight input by up to 10% before changing the higher-weight input by 1%.

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

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

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

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

We want to ensure that the controller, based on a model with this uncertainty, is still stable. Therefore, in the step labeled *Robust* in Figure 3, we perform Robust Stability Analysis (RSA) [11]. RSA checks whether a perturbation equal in magnitude to the uncertainty, if coming at the worst time and in the worst manner, can make the system unstable — e.g., prevent the output ripples in Figure 4(a) from dying down. If the system is unstable, we have to go back and change the inputs to the controller design. In particular, we can use more challenging applications in the initial design, or use lower Q weights relative to R weights, thereby making the system less ripply and more cautious to changes.

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

C. Adding Additional Inputs and/or Outputs

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

V. Uses of the Controller

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

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

Time-Varying Tracking: A more advanced use is when a high-level agent monitors real-time conditions and, based on those, changes the target values that it wants the outputs to track [33]. A typical example is a battery-powered mobile platform that runs a program as the battery is being depleted. The best tradeoff in performance (or quality of service) versus power consumed changes as the battery energy level decreases. For example, while the battery is above a certain level, the output targets are high performance and a tolerably-high power consumption. As the battery level decreases, the OS sets successively lower pairs of performance and power targets to conserve battery life [2], [33], [34], [35], [36].

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

Figure 5(a) shows the envisioned system. The outputs are IPS and P, and we are trying to maximize IPS^2/P. The search is driven by an extension to the original controller that we call *Optimizer*. It can be a part of the runtime system or a hardware module.

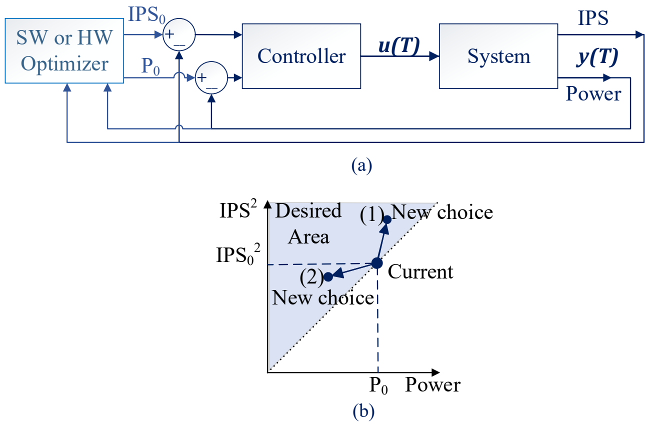


Figure 5: Using MIMO to optimize a combination of measures such as $E\times D$.

Initially, the optimizer sets a certain target IPS$_0$ and P$_0$. The base controller then generates the input configuration (e.g., frequency, issue width, and load/store queue size) that attains this target. After the system converges, the optimizer changes the target outputs so that IPS^2/P increases. Specifically, it either increases P a little and increases IPS much more, or decreases IPS a little and decreases P much more. This is shown in Figure 5(b), where the original point is called *Current*, and the two possible changes are (1) and (2), respectively. Based on the new (IPS$_1$, P$_1$), the base controller regenerates the input configuration.

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

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

VI. Example of MIMO Control System

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

A. Controller Design

1) Choosing Input/Output Weights: To assign the input and output weights, we proceed based on the discussion of Section IV-B2. Specifically, among the outputs, we assign to power a higher weight than to IPS, to minimize power tracking errors and power budget violations. As shown in Table III, we use weights of 1,000:1 for power:IPS, which makes power $\sqrt{1,000}\times$ (or $\approx 30\times$) more important than IPS. In other words, we are willing to trade 1% deviation from the power reference for 30% deviation from the IPS reference.

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

We consider that it is more important for the outputs to remain close to their reference values than to minimize the overheads of changing inputs. Hence, we give higher weights to the outputs than to the inputs. As discussed in Section IV-B2, if the ratio of output to input weights is too high, the system becomes ripply and takes longer to converge; if it is too low, the inputs are sluggish, and any perturbation also takes long to disappear. We need to experiment with MATLAB to ensure that the chosen ratio falls in between the two scenarios, and hence the system converges reasonably fast. To select the output to input weight ratio, we need to consider the less important output (IPS) and the most important input (frequency). As shown in Table III, we use weights of 1000:1 for IPS:frequency, which makes IPS $\approx 30\times$ more important than frequency.

Section VIII-A performs a sensitivity analysis of the weights.

2) Model Identification & Uncertainty Analysis: It is challenging to build analytical processor models that can accurately relate processor performance and power with cache size and frequency. Hence, we perform experimental System Identification [27], [32] of a cycle-level simulator that we wrote to model the processor system. We run four profiling applications from SPEC CPU 2006 on the simulator — two integer (sjeng and gobmk) and two floating-point (leslie3d and namd). For each program, we apply test waveforms of cache size and frequency changes at runtime. We record the time variation for the inputs and outputs.

Table III: Control and architecture parameters.

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

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

As per Section IV-B4, we use uncertainty analysis to revise the design (Figure 3). We run two additional applications (h264ref and tonto) on both the simulator and the model obtained with system identification, changing size and frequency signals. The outputs are compared. We find that the *maximum* error in the model is 14% for IPS and 10% for power. Then, we conservatively set the uncertainty guardbands to 3$\times$ these values, namely to 50% for IPS and 30% for power (Table III).

Recall from Section IV-B4 that these uncertainty guardbands refer to the *average* prediction errors across the whole application execution that are tolerable. After choosing the uncertainty guardbands, we run Robust Stability Analysis to see if the system converges. If it does not, we use MATLAB to reconstruct the controller with larger input weights, until the system is shown to converge for the desired guardbands.

Section VIII-C performs a sensitivity analysis of the uncertainty guardband.

B. Optimizer Design

As discussed in Section V, the Optimizer performs a high-level search for the optimal operating point, according to Figure 5(b). Depending on our goal, the search can be in the $E{\times}D$, $E{\times}D^2$, … $E{\times}D^{k-1}$ space. To minimize $E{\times}D^{k-1}$, the algorithm tries to maximize IPS^k/P.

Every time that the algorithm is invoked, it starts by setting the inputs to their midrange values: 1GHz frequency and (4,2) associativity for (L2,L1) caches. Then, it makes a move in one of the two directions in Figure 5, namely "Up" (higher IPS but only slightly higher power) or "Down" (slightly lower IPS and much lower power). If the resulting value of the measure IPS^k/P is higher than the previous one, the algorithm continues to explore more points in the same direction. Otherwise, it reverses the search direction. This process repeats for a fixed number of trials (*MaxTries* as shown in Table III). We do not use backtracking in this algorithm.

C. Overheads of the Design

Both controller and optimizer operation cause very minor overheads. The controller is invoked every 50μs, and operates entirely in hardware. It reads performance and power counters, and computes the difference between the values and their references. It then performs four floating-point vector-matrix multiplies, and generates the actuations on cache and frequency. The controller only stores less than 100 floating-point numbers. The optimizer is invoked every 10ms or when there is a phase change as detected in [8]. It also runs in hardware.

D. Adding an Additional Input

To show the flexibility of MIMO control, in a second set of experiments, we augment the controller by adding a third configurable input: the size of the reorder buffer (ROB). The ROB size is changed by power gating 16 entries at a time, as described in [37]. Since the full ROB has 128 entries, we have 8 different ROB sizes (Table III).

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

VII. Evaluation Methodology

A. Infrastructure

We base our evaluation on simulations of a processor like the ARM Cortex-A15 modeled with the ESESC simulator [38]. The architecture parameters optimized for *best* E$\times$D *product in the baseline architecture* are listed in Table III. We modified ESESC to model the configurable inputs and implement the hardware controller and optimizer. Power estimates are obtained from the McPAT modules that are integrated within ESESC. We use CACTI 6.0 for cache power estimates. DVFS pairs are obtained from interpolating published A15 DVFS values [39]. We use MATLAB's System Identification and Robust Control Toolboxes [40] for system identification, LQG controller design, tuning, and robustness evaluation. We run all the SPEC CPU 2006 applications except zeusmp, which our infrastructure does not support. We group the applications into a training set (sjeng, gobmk, leslie3d, and namd) and a production set (the remaining ones). Each application is monitored for an average of 50 billion instructions, after fast forwarding 10 billion instructions.

B. Experiments

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

2) Time-Varying Tracking: The goal of this experiment is for the outputs to track time-changing reference values. As indicated before, an example is when a high-level agent throttles performance and limits power consumption based on operating conditions such as battery levels [34], [35]. We model such a scenario by changing the IPS and power targets based on the recently-introduced Quality of Experience

(QoE) parameter in handheld devices [36]. We use the analytical models for QoE and battery charge consumption in [36] to compute how the targets should be changed. We set the time betweeen changes to 2,000 epochs of 50μs each, and the total energy supply to 1 J.

3) Fast Optimization Leveraging Tracking: The goal of this experiment is to generate outputs that minimize energy (E), $E\times D$, or $E\times D^2$. For the optimization (Figure 5(b)), the optimizer can try at most *MaxTries* trials.

C. Architectures Evaluated

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

Table IV: Architectures compared.

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

The other designs are our configurable architecture with different hardware controller algorithms to drive input adaptations. *Heuristic* uses a sophisticated heuristics-based algorithm similar to [41], which is tuned with the training set applications. The algorithm has two steps. First, it ranks the adaptive features (cache size, frequency, and ROB size) according to their expected impact in this application, like [8]. The second step depends on the experiment performed.

In tracking experiments, the second step involves taking different actions, using the ranked features in order, depending on the difference (magnitude and sign) between each output value and its reference value. These actions are qualified by threshold values experimentally determined. In the optimization experiments, the second step involves searching the space (e.g., $E\times D^2$) using an iterative process, testing a few configurations of each of the adaptive features in rank order. This is similar to earlier schemes [10], [23], [41], [42].

Details of the algorithms can be found in [43]. Note that, for *Heuristic*, the algorithms developed and tuned for the two-input system (cache size and frequency) have to be completely redesigned from scratch for the three-input system (cache size, frequency, and ROB size).

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

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

VIII. Results

This section evaluates our four architectures. Due to space limitations, the evaluation is kept concise. More details can be found in [43].

A. Impact of Input and Output Weights

To see the impact of input and output weights, we run the MIMO controller with the different sets of weights in Table V, tracking 2.5 BIPS for IPS and 2 W for power (P). For the application *namd*, Figure 6 shows the epochs taken to achieve steady state (a), and the output tracking errors (b) — i.e., the difference between output values at steady state and their reference values. In all cases, we initialize the system with the same input values, which are 20% and 30% different than the reference IPS and power values, respectively.

Table V: Different sets of weight choices.

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

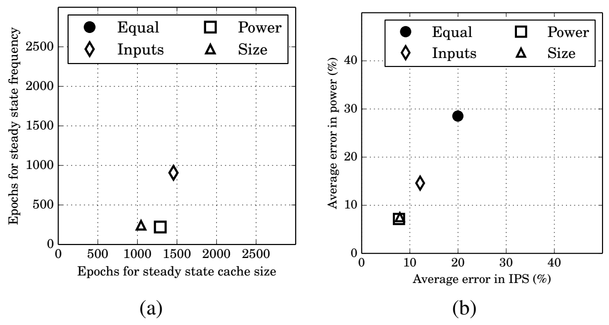


Figure 6: Epochs to achieve steady state (a) and output tracking errors (b) for different weight choices.

In *Equal*, all inputs and outputs have the same weight. In this case, the relatively high input weights make the controller reluctant to change inputs significantly. The controller makes only small input changes, many of which are rounded to zero. As a result, for the duration of our experiment, the outputs do not converge yet. Hence, the *Equal* datapoint is missing in Figure 6(a), and has not moved from initial conditions in Figure 6(b). In *Inputs*, minimizing input changes is less important than meeting the output targets. Hence, the output tracking errors decrease (Figure 6(b)) and the system converges within the measured time (Figure 6(a)).

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

B. *Impact of Model Dimension*

The number of model dimensions is a tradeoff between accuracy and computation overhead. With more model dimensions, we model the true system more accurately, but the controller requires more computations. In practice, for our small system, computation overhead is not a concern. Still, we would like to use as few dimensions as possible while retaining accuracy. Hence, we compare the IPS and P attained by the true system (i.e., the simulator) and our model with different dimensions. We refer to the difference as the error. Figure 7 shows the maximum errors for different dimensions. Based on this result, we use a model dimension of 4.

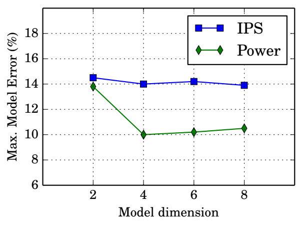


Figure 7: Maximum prediction errors for different model dimensions.

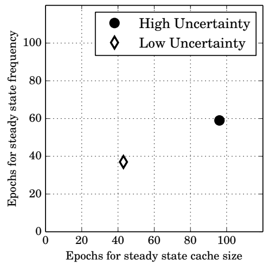


Figure 8: Time to achieve steady state for different uncertainty guardbands.

C. *Impact of Uncertainty Guardband*

The size of the uncertainty guardband is a tradeoff between the time to attain steady state and the risk of system instability. If we bet that production applications will behave more like the training applications and, hence, a smaller uncertainty guardband is acceptable, we can reduce the input weights. Then, the system will reach the steady state faster. However, if a production application deviates more than we expected, the system will become too ripply and not reach steady state. As per Section VI-A2, to design our controller, we use uncertainty guardbands equal to 50% for IPS and 30% for power. Figure 8 shows the resulting number of epochs needed to reach steady state with our controller (*High Uncertainty*). It also shows the number needed if we had used a more aggressive design with lower uncertainty guardbands equal to 30% for IPS and 20% for power (*Low Uncertainty*). From the figure, we see that the more aggressive design is still stable — and hence needs fewer epochs to achieve steady state. Hence, our controller design is conservative.

D. *Using MIMO for Tracking Multiple References*

We compare how effectively *MIMO*, *Heuristic*, and *Decoupled* can track multiple output reference values — specifically, 2.5 BIPS for IPS and 2 W for P. As indicated in Section VII-B1, this IPS reference value is high, and several memory-bound applications cannot reach it. For these applications, which we call *Non-responsive*, no amount of control can get IPS and P very close to their targets. For the rest, which we call *Responsive* applications, different control architectures have different effectiveness. The non-responsive applications are *bzip2*, *gcc*, *hmmer*, *h264ref*, *libquantum*, *mcf*, *omnetpp*, *perlbench*, *Xalan*, *bwaves*, *dealII*, *GemsFDTD*, *lbm*, and *soplex*.

Figure 11(a) and (b) show the average error in IPS and P for the responsive and non-responsive applications, respectively, for the *MIMO*, *Heuristic*, and *Decoupled* architectures. For each architecture, the figures show a small data point for each application and a large datapoint for the average of all the applications.

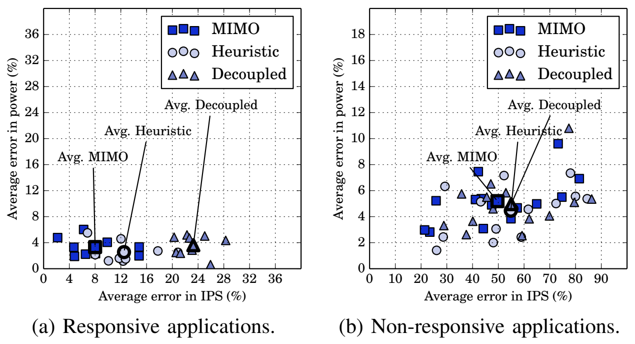


Figure 11: Results for tracking multiple references.

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

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

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

E. *Using MIMO for Time-Varying Tracking*

We change the IPS and P reference values periodically, to minimize the decrease in quality of experience in handheld devices [36], and observe the outputs using the *MIMO*, *Heuristic*, and *Decoupled* architectures. Figure 12 shows the resulting IPS values as a function of time for each architecture and the reference. Figures 12(a) and (b) correspond to *astar* and *milc*, respectively. We do not show P values as all the architectures perform similarly well.

The figure shows that *MIMO* is able to track the time-varying reference IPS values well, much closer than the other architectures. *Heuristic* and *Decoupled* attain an IPS that is

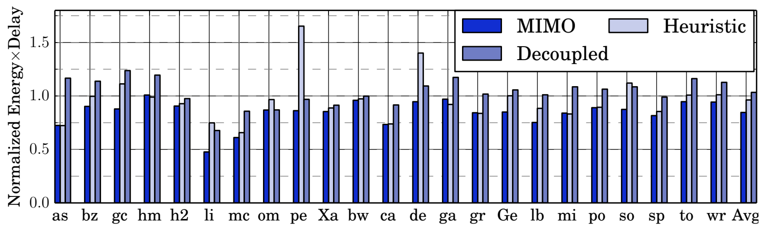


Figure 9: Energy×Delay minimization.

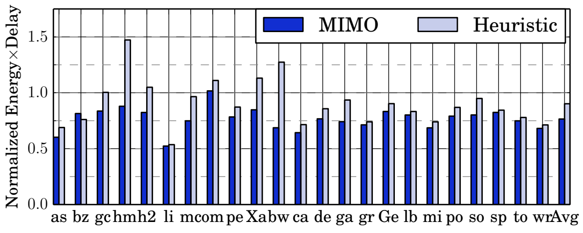
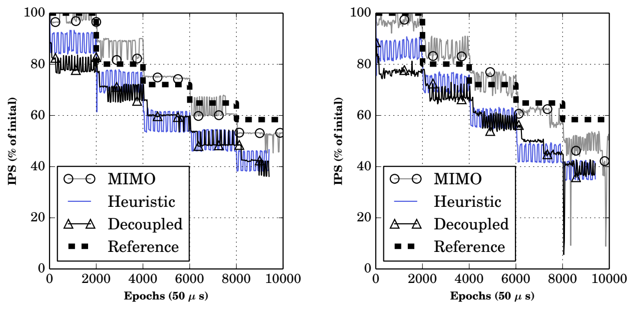


Figure 10: Energy×Delay minimization with 3 inputs.



(a) astar (b) milc

Figure 12: Examples of time-varying tracking.

lower than the reference. For the lowest IPS at the end of the battery life, *MIMO* performs a bit worse than expected (but still better than the other architectures). This is because we set an IPS reference that is too aggressive when combined with the companion P reference.

F. Using MIMO for Fast Optimization Leveraging Tracking

We compare the ability of the different controllers to optimize a combination of outputs. We first consider minimizing $E{\times}D$. Figure 9 shows the $E{\times}D$ of the different applications under *MIMO*, *Heuristic*, and *Decoupled*. For each application (and the average in the far right), the bars are normalized to the $E{\times}D$ of *Baseline*.

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

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

G. Adding a New Input: Configurable ROB Size

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

Figure 10 shows the $E{\times}D$ of the different applications under *MIMO* and *Heuristic*. As usual, the bars in each application (and the average) are normalized to the $E{\times}D$ for *Baseline*. On average, *MIMO* and *Heuristic* reduce the $E{\times}D$ of the applications by 25% and 12%, respectively.

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

IX. Related Work

The argument for systematic coordinated control of multiple power/performance management policies has been advocated in prior research [10], [20], [44], [45], [46], [47]. In addition, hardware support for adaptive power/performance management is increasingly being used in modern processors [1], [2], [3], [5]. The Intel Skylake processor [2] uses a SISO PID controller within its energy management architecture. The IBM POWER 8 processor [1] has reconfiguration registers within the pipeline, and supports fast and fine-grained per-core DVFS. The Intel IvyBridge processor [3] can resize its last-level cache by power gating its ways.

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

Rule-Based Heuristics. There are some works that use rule-based heuristics to adapt configurable architectures. Some of these works adapt one resource (e.g., [7], [8]). Other works adapt multiple resources in a coordinated manner (e.g., [6],

[10], [24], [25]). In particular, Vega et al. [10] demonstrate the conflicting nature of decoupled management of multiple policies. Zhang and Hoffmann [41] propose a framework for maximizing performance under a power cap using a heuristic algorithm. Our *Heuristic* architecture uses a similar algorithm.

Model-Based Heuristics. There are some works that use models to drive the adaptation heuristics. For example, they use models for joint memory and processor DVFS [22], cache size [48], multicore thermals [23], or on-chip storage [49]. Our MIMO methodology also uses an offline model of the processor dynamics.

Control Theoretic Techniques. In addition to the SISO schemes discussed in Section II [2], [9], [15], [17], there are works that use multiple SISO controllers managed together with heuristics [50], [51], [52]. The use of such decoupled controllers has to be planned carefully before deploying them. If there are cross dependences between the input of one and the output of another, then the controllers will conflict with each other.

The approach that combines multiple SISO models to generate a larger MISO controller [12], [14], [16] still requires heuristics to encode some decisions. This may make the controllers suboptimal and error prone [19], [53]. The MIMO methodology can natively model the required interactions, eliminating unanticipated behavior and finding better solutions.

There have been some designs that use hierarchical control loops to coordinate multiple conflicting policies or objectives. Specifically, Raghavendra et al. [44] propose such a scheme to control power in datacenters, and Fu et al. [13] propose another such design to control utilization in processors. In these proposals, a formal controller is used only in the innermost loop, and each higher level works on a different abstraction and specifies the targets for the lower levels. This approach works well for systems that are naturally suited for hierarchical management. For other cases, it might reduce the freedom of the slower timescale controller, hence resulting in a sub-optimal operation.

Machine Learning techniques. Machine learning techniques have been used to tune architectural parameters (e.g., [20], [21], [54]). They have two main differences with control theory techniques. The first difference is runtime feedback. Machine learning techniques learn by recording what input values are best for different observed output conditions. However, if they find different output conditions at runtime than those they were trained on, they provide a lower-quality solution unless they go through an expensive re-training phase. Control theory techniques, instead, when they find runtime output conditions to be different than those modeled, they use their intrinsic feedback loop to adapt to the new conditions with low overhead. The second difference is design guarantees. Unlike machine learning techniques, control theory techniques can provide convergence, stability, and optimality guarantees.

Optimization techniques. There are some works that adapt based on optimization formulations. For example, they optimize inputs to minimize power consumption [55], [56], performance subject to power constraints [18], or $E{\times}D^2$ [29]. STEAM [19] models the power and performance of cores as a function of P- and T-states, IPC, temperature, and memory accesses. It then uses a convex optimization solver to maximize the ratio of performance over power.

X. Conclusion

Control theoretic MIMO controllers, which actuate on multiple inputs and control multiple outputs in a coordinated manner are likely to be key as future processors become more resource-constrained and adaptive. In this paper, we used MIMO control theory techniques to develop controllers to dynamically tune architectural parameters in processors. To our knowledge, this is the first work in this area. We discussed three ways in which a MIMO controller can be used. We developed an example MIMO controller and showed that it is substantially more effective than controllers based on heuristics or built by combining single-output formal controllers.

Acknowledgments

This work was supported in part by NSF under grants CCF-1012759 and CCF-1536795.

References

[1] B. Sinharoy *et al.*, “Advanced features in IBM POWER8 systems,” *IBM Jour. Res. Dev.*, vol. 59, no. 1, pp. 1:1–1:18, Jan. 2015.

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

[3] S. Jahagirdar *et al.*, “Power Management of the Third Generation Intel Core Micro Architecture formerly Codenamed Ivy Bridge,” in *Hot Chips*, 2012.

[4] A. Naveh *et al.*, “Power and Thermal Management in the Intel Core Duo Processor,” *Intel Technology Journal*, vol. 10, no. 2, pp. 109–122, May 2006.

[5] E. Rotem *et al.*, “Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge,” *IEEE Micro*, vol. 32, no. 2, pp. 20–27, Mar. 2012.

[6] D. H. Albonesi *et al.*, “Dynamically Tuning Processor Resources with Adaptive Processing,” *Computer*, vol. 36, no. 12, pp. 49–58, Dec. 2003.

[7] R. Balasubramonian *et al.*, “Memory Hierarchy Reconfiguration for Energy and Performance in General-purpose Processor Architectures,” in *MICRO*, 2000.

[8] C. Isci *et al.*, “Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management,” in *MICRO*, 2006.

[9] Z. Lu *et al.*, “Control-theoretic Dynamic Frequency and Voltage Scaling for Multimedia Workloads,” in *CASES*, 2002.

[10] A. Vega *et al.*, “Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control,” in *MICRO*, 2013.

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

[12] A. Bartolini *et al.*, “A Distributed and Self-calibrating Model-Predictive Controller for Energy and Thermal Management of High-Performance Multicores,” in *DATE*, 2011.

[13] X. Fu *et al.*, "Cache-Aware Utilization Control for Energy Efficiency in Multi-Core Real-Time Systems," in *ECRTS*, 2011.

[14] Y. Wang *et al.*, "Temperature-constrained Power Control for Chip Multiprocessors with Online Model Estimation," in *ISCA*, 2009.

[15] Q. Wu *et al.*, "Formal online methods for voltage/frequency control in multiple clock domain microprocessors," in *ASPLOS*, 2004.

[16] F. Zanini *et al.*, "Multicore Thermal Management using Approximate Explicit Model Predictive Control," in *ISCAS*, 2010.

[17] K. Ma *et al.*, "Scalable Power Control for Many-core Architectures Running Multi-threaded Applications," in *ISCA*, 2011.

[18] P. Petrica *et al.*, "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems," in *ISCA*, 2013.

[19] V. Hanumaiah *et al.*, "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.

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

[21] C. Dubach *et al.*, "A Predictive Model for Dynamic Microarchitectural Adaptivity Control," in *MICRO*, 2010.

[22] Q. Deng *et al.*, "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *MICRO*, 2012.

[23] H. Jung *et al.*, "Stochastic Modeling of a Thermally-Managed Multi-Core System," in *DAC*, 2008.

[24] C. Isci *et al.*, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *MICRO*, 2006.

[25] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *ISCA*, 2002.

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

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

[28] T. Karkhanis and J. Smith, "A First-Order Superscalar Processor Model," in *ISCA*, 2004.

[29] B. C. Lee and D. Brooks, "Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity," in *ASPLOS*, 2008.

[30] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *ASPLOS*, 2006.

[31] B. C. Lee *et al.*, "CPR: Composable Performance Regression for Scalable Multiprocessor Models," in *MICRO*, 2008.

[32] L. Ljung, "Black-box Models from Input-output Measurements," in *IMTC*, 2001.

[33] H. Zeng *et al.*, "ECOSystem: Managing Energy As a First Class Operating System Resource," in *ASPLOS*, 2002.

[34] J. Flinn and M. Satyanarayanan, "Energy-aware Adaptation for Mobile Applications," in *SOSP*, 1999.

[35] D. C. Snowdon *et al.*, "Koala: A Platform for OS-level Power Management," in *EuroSys*, 2009.

[36] K. Yan *et al.*, "Characterizing, Modeling, and Improving the QoE of Mobile Devices with Low Battery Level," in *MICRO*, 2015.

[37] D. Ponomarev *et al.*, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," in *MICRO*, 2001.

[38] E. K. Ardestani and J. Renau, "ESESC: A Fast Multicore Simulator Using Time-Based Sampling," in *HPCA*, 2013.

[39] V. Spiliopoulos *et al.*, "Introducing DVFS-Management in a Full-System Simulator," in *MASCOTS*, 2013.

[40] *MATLAB and Simulink Release 2013b*. Natick, Massachusetts: The MathWorks Inc., 2013.

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

[42] W. Wang *et al.*, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-time Multi-core Systems," in *DAC*, 2011.

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

[44] R. Raghavendra *et al.*, "No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center," in *ASPLOS*, 2008.

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

[46] M. C. Huang *et al.*, "Positional Adaptation of Processors: Application to Energy Reduction," in *ISCA*, 2003.

[47] X. Li *et al.*, "Performance Directed Energy Management for Main Memory and Disks," in *ASPLOS*, 2004.

[48] R. Sen and D. A. Wood, "Reuse-based Online Models for Caches," in *SIGMETRICS*, 2013.

[49] S. Dropsho *et al.*, "Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power," in *PACT*, 2002.

[50] A. Mishra *et al.*, "CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors," in *SC*, 2010.

[51] K. Ma *et al.*, "DPPC: Dynamic Power Partitioning and Control for Improved Chip Multiprocessor Performance," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1736–1750, Jul. 2014.

[52] P. Juang *et al.*, "Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors," in *ISLPED*, 2005.

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

[54] C. Dubach *et al.*, "Dynamic Microarchitectural Adaptation Using Machine Learning," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 31:1–31:28, Dec. 2013.

[55] K. Meng *et al.*, "Multi-optimization Power Management for Chip Multiprocessors," in *PACT*, 2008.

[56] W. Wang and P. Mishra, "Leakage-Aware Energy Minimization Using Dynamic Voltage Scaling and Cache Reconfiguration in Real-Time Systems," in *VLSID*, 2010.