SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management

Bryan Donyanavard
Tiago Mück
Amir M. Rahmani
Nikil Dutt
bdonyana@uci.edu
tmuck@uci.edu
a.rahmani@uci.edu
dutt@ics.uci.edu
UC Irvine

Armin Sadighi
Florian Maurer
Andreas Herkersdorf
armin.sadighi@tum.de
flo.maurer@tum.de
herkersdorf@tum.de
TU Munich

ABSTRACT

Resource management strategies for many-core systems dictate the sharing of resources among applications such as power, processing cores, and memory bandwidth in order to achieve system goals. System goals require consideration of both system constraints (e.g., power envelope) and user demands (e.g., response time, energy-efficiency). Existing approaches use heuristics, control theory, and machine learning for resource management. They all depend on static system models, requiring a priori knowledge of system dynamics, and are therefore too rigid to adapt to emerging workloads or changing system dynamics.

We present SOSA, a cross-layer hardware/software hierarchical resource manager. Low-level controllers optimize knob configurations to meet potentially conflicting objectives (e.g., maximize throughput and minimize energy). SOSA accomplishes this for many-core systems and unpredictable dynamic workloads by using rule-based reinforcement learning to build subsystem models from scratch at runtime. SOSA employs a high-level supervisor to respond to changing system goals due to operating condition, e.g., switch from maximizing performance to minimizing power due to a thermal event. SOSA's supervisor translates the system goal into low-level objectives (e.g., core instructions-per-second (IPS)) in order to control subsystems by coordinating numerous knobs (e.g., core operating frequency, task distribution) towards achieving the goal. The software supervisor allows for flexibility, while the hardware learners allow quick and efficient optimization.

We evaluate a simulation-based implementation of SOSA and demonstrate SOSA's ability to manage multiple interacting resources in the presence of conflicting objectives, its efficiency in configuring knobs, and adaptability in the face of unpredictable workloads. Executing a combination of machine-learning kernels and microbenchmarks on a multicore system-on-a-chip, SOSA achieves target performance with less than 1% error starting with an untrained model, maintains the performance in the face of workload disturbance, and automatically adapts to changing constraints at runtime. We also demonstrate the resource manager with a hardware implementation on an FPGA.

ACM Reference Format:
Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In *MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358312

1 INTRODUCTION

As system size and capability scale, designers face a large space of configuration parameters controlled by actuation knobs, which in turn generate a large number of cross-layer actuation combinations [53]. Making runtime decisions to configure knobs in order to achieve a simple goal (e.g., maximize performance) can be challenging. That challenge is exacerbated when considering a goal that may change throughout runtime, and consist of conflicting objectives (e.g., maximize throughput within a power budget) [46]. Additionally, modern embedded devices, like in cars, are expected to support combinations of a wide range of applications, e.g., assistant and entertainment systems, without any prior knowledge of their workload.

Consider a typical entertainment system scenario in which the resource management goal is to maximize performance within a power budget. One could write an optimization heuristic to find the optimal operating point for a small number of knobs, but such a solution does not scale well to coordinate a large number of knobs. A heuristic that does a thorough optimal estimation is not be efficient, and one with a rougher estimation is not be flexible enough to changes in workload.

Alternatively, we could use feedback control to adaptively configure our knobs to achieve high performance within a power budget. Control theoretic solutions have been proposed to achieve exactly

MICRO '52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358312

this goal in a variety of ways, most recently using LQG [35] and SSV [36]. Control theoretic resource managers can coordinate knobs with the added benefit of formalism, and can scale in cases that the system and its knobs can be decomposed into subsystems.

Consider now that the system we are controlling experiences a thermal event. The goal is no longer to maximize performance: the controller should change its priority to minimize power. Classical control theoretic solutions lack the ability to adapt to changing goals. Rahmani et al. [39] use Supervisory Control Theory to address the issue of dynamic goals by changing the priorities of low-level controllers adaptively. However, designing a controller requires a stable (sub)system model to be identified. This model is fixed and must be known at design time. If our user installs a new application that exercises the system in ways we do not anticipate (e.g., using Bluetooth connectivity), it may break our resource manager. Due to model dependency, the controller may not converge when introduced to an unknown combination of applications.

Machine learning is well-suited to navigate large configuration spaces. Machine learning would be useful for finding the optimal operating point for a set of knobs in order to achieve the provided goal. We could also build and update a model at runtime using reinforcement learning. Reinforcement learning is not commonly applied to scenarios such as ours due to the computational cost. Mishra et al. [29] use reinforcement learning to tune feedback control parameters at runtime, but the learning is done periodically on a remote server, and requires an initial model. We need the ability to capture the dynamics of a system during execution without any previous observation, and continuously update that model. If we can learn system dynamics at runtime, beginning with a blank slate (i.e., empty model), we can define (1) objectives without static relationships to subsystem states, and (2) optimization goals without access to the targeted hardware platform. For example, we can define objective targets in term of application-specific quality of service (QoS) metrics (e.g. heartbeats [21], frames-per-second (FPS)), and learn the dynamics for different applications and combinations of applications.

In this paper, we leverage supervisory control to adapt to changing goals at runtime, in combination with reinforcement learning hardware that allows us to efficiently optimize for any unpredictable workload or system dynamics. To our knowledge, SOSA is the first learning-control-hybrid resource manager to provide self-adaptivity via software supervisor and self-optimization via hardware-based reinforcement learning on-device. **Key contributions** of this paper are:

- We provide **self-adaptivity** to resource managers through hierarchical supervision, allowing the resource manager to respond to changing system goals. We achieve self-adaptivity through dynamic goal management by updating the policy (i.e., parameters) of low-level controllers according to high-level goal(s).
- We enable **self-optimization** of low-level controllers through reinforcement learning in order to adapt to unpredictable workloads. Reinforcement learning is accomplished by implementing learning classifier tables (LCTs) as low-level controllers in hardware. The LCTs capture system dynamics by building a model from scratch at runtime, and continuously updating the model. LCTs enable us to perform **model-independent** rule-based learning **on-device**.
- Experimental Case Study: We deploy SOSA on a FPGA board including a **hardware implementation** of LCTs and supervisor to validate the function and design feasability. We **compare the effectiveness** of SOSA to state-of-the-art alternatives for resource management of multicores in a simulated environment, showing SOSA's accuracy in meeting performance demands and responsiveness to dynamic power constraints for workloads with unpredictable background task interference.

2 BACKGROUND AND RELATED WORK

Resource management approaches for processors can be broadly categorized in three primary ways: (1) heuristic-based approaches [8, 10–12, 17, 19, 23, 24, 34, 38, 47, 49], (2) control-theory-based approaches [22, 26, 27, 30, 33, 35–37, 39–41, 44, 45], and (3) stochastic / machine-learning-based approaches [1, 6, 7, 9, 25, 29]. There exist some proposed solutions that incorporate aspects of multiple categories, e.g., there are a number of works that use learning to build predictive models, and use heuristics to make runtime decisions based on the predictive models [4, 13, 15, 16].

We can use a number of properties to describe the capabilities of existing adaptive resource managers. To start, as a requirement all approaches must be **efficient** enough to deploy at runtime and be responsive, and **coordinate** multiple knobs to achieve one or more objectives. Machine learning and feedback-control based approaches have the added benefit of providing **formalism**. Classical control-theoretic approaches can provide **robustness** by guaranteeing bounded behavior. Reinforcement learning approaches can **self-optimize** by continuously updating models based on observation.

Table 1 shows the coverage of existing on-chip resource management approaches in handling key issues. Some machine-learning-based and heuristic approaches (e.g., [4, 16–18]) focus on **efficiency (3)** and **coordination (4)**, but fail to address other attributes such as providing **robustness (1)** against unexpected corner cases. Classical control-theoretic approaches (e.g., [35, 37]) provide means to address **robustness (1)**, **formalism (2)**, and **efficiency (3)**, with the ability to concurrently **coordinate (4)** and control multiple objectives in a non-conflicting manner. However, classical control lacks **scalability (5)** for heterogeneous multi-processing (HMP) architectures due to 1) the exponential growth in computational complexity with increasing numbers of inputs and outputs, and 2) the difficulty of performing Dynamic System Model identification for large systems. Although multiple simple controllers have been used in nested loops to achieve scalability in simple control problems [22, 26], they suffer from scalability issues in complex resource management problems for many-core systems where coordination of multiple actuators is necessary.

Recently, in Yukta [36], Pothukuchi et al. solve the **scalability (5)** issue of classical controllers by using Robust Control and hierarchically linking controllers to perform resource management at various layers in the system stack. However, Yukta, like classical controllers, lacks **self-adaptivity (6)**, which enables rapid responses to abrupt runtime changes. In SPECTR [39], Rahmani et al. also solve the scalability issue via hierarchy. SPECTR uses Supervisory Control Theory at the top of its hierarchy which additionally provides **self-adaptivity (6)** in conjunction with classical controllers by coordinating their reference values and updating priorities dynamically.

Methods	Estimation-/Model-based Heuristics [10, 11, 13, 17, 24]	Classical Control Theory [22, 26, 35, 40, 41]	Machine Learning [4, 16, 18]	Hierarchical Control [36, 39]	Hybrid Control + Machine Learning [SOSA]
1. Robustness		✓		✓	
2. Formalism		✓	✓	✓	✓
3. Efficiency	✓	✓	✓	✓	✓
4. Coordination	✓	✓	✓	✓	✓
5. Scalability				✓	∗
6. Self-Adaptivity				✓	∗
7. Self-Optimization			✓		✓
8. Model-Independence	✓				∗

Table 1: Major on-chip resource management approaches and the key challenges they address (∗ = uniquely addressed by SOSA).

In this paper, we deploy a hierarchical supervisory controller in order to provide scalability and self-adaptivity. We use rule-based reinforcement learning by deploying Learning Classifier Tables, or *LCT*s, as low-level controllers. The LCTs provide **self-optimization (7)** and **model-independence (8)** by continuously updating the optimal configuration(s) based on runtime observation. Theoretical investigation into managing DVFS using reinforcement learning [6] for a single objective has been promising. Prior to our work, Mishra et al. proposed a learning and control hybrid resource manager in CALOREE [29]. CALOREE uses a predictive model to optimize the control parameters for the controller making decisions. CALOREE requires an initial model trained ahead of execution. The model is updated using reinforcement learning at runtime, however, the learning is done off-device, and requires communication with a remote server. Continuously updating a statistical model on device was applied by Kasture et al. in [25] to control DVFS in datacenters for latency-critical workloads. Compared to SOSA their approach is neither self-adaptive to runtime changes, nor provides coordination of several conflicting objectives.

3 MOTIVATION

3.1 Challenges of Model-dependence

Consider the DVFS feedback controller shown in Figure 1. The controller sets the operating frequency (and voltage) of a single-core system to achieve a desired heartbeat rate. The heartbeat rate is a quality-of-service (QoS) metric the application designer specifies through source code annotation [21].

In the case of control theory, the controller is designed based on a static model that identifies the achievable heartbeat based on the operating frequency. This assumes that a physical system is available for observation of system dynamics, which is required to generate the model used to design the controller ahead of deployment. Furthermore, the frequency→HB relationship is application-specific. In other words, (a) workloads must be known ahead of *design* time, (b) systems must be available for observation of known workloads, and (c) each resulting controller only applies to the specific workload it was designed for. These are impractical assumptions when designing controllers for general-purpose systems with dynamic and unpredictable workloads. More challenges arise due to changes in system dynamics over time or between devices. Consider the effects of process variability on the behavior of different devices with respect to

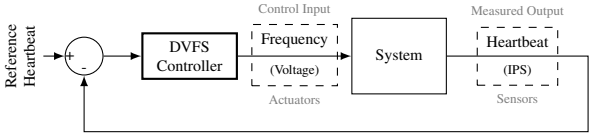


Figure 1: Feedback controller with frequency as control input and heartbeat [21] rate as measured output.

operating frequency and voltage. It is impractical to expect an optimized model to be derived at design time for each device that utilizes the controller. As a result, a system may display non-ideal behavior according to the model used to design the controller. The controller would make potentially poor decisions due to an inaccurate model [14].

The challenges outlined so far assume that the system dynamics being modeled can be estimated with a simple linear equation, which is the case for the frequency→HB controller. However, some knobs have more complex system dynamics and are not practical to model with discrete difference equations, e.g., task migration. Complex models with large configuration spaces such as task migration are proper candidates to apply learning. It is important to manage the scale of a complex model if it is to be learned at runtime. A model-based learner that uses a *static* model would face the same challenges as described thus far. To solve these issues we can employ online reinforcement learning. Online reinforcement learning addresses the static-model challenges by continuously updating the system model based on runtime observations. If we can implement such a learner on-device, it can capture complex dynamics such as task migration.

3.2 Benefits of Reinforcement Learning

Consider again the DVFS feedback controller shown in Figure 1. We implement the feedback controller in two different ways: (1) using single-input-single-output (SISO) control theory, and (2) using rule-based reinforcement learning (LCT). Figure 2 shows the accuracy achieved by SISO (blue) and LCT (orange) controllers tracking a specified HB for the `k-means` clustering algorithm executing on a simulated ARM core (detailed in Section 6.1). The SISO begins with an error of 20 %, and is able to eventually reduce the error to less than 5 % after two seconds of execution. The LCT begins with near 100 % error, and is able to to reduce the error to 15% after two

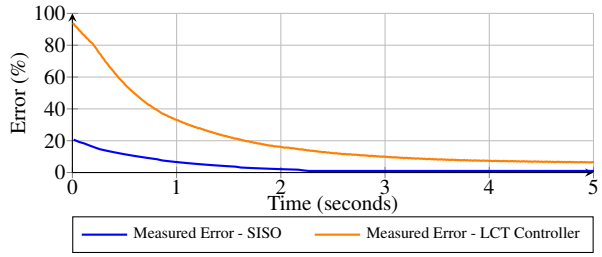


Figure 2: Accuracy of classical (SISO) and learning (LCT) controllers tracking application heartbeat rate using core operating frequency.

seconds of execution, and eventually down to 7 % after five seconds. Although the SISO is robust, its design requires a model at design time. The LCT is a blank-slate that learns the model during execution, which is why it begins with nearly 100% error. In this instance we did not tune LCT parameters or optimize rules – with some design iterations, we could reduce the error further. The LCT's ability to learn to manage HB on the fly indicates that there is opportunity to exploit this approach to coordinate knobs for subsystems in the context of a resource management hierarchy.

In the example, we define both the LCT and SISO objectives in terms of instructions-per-second (IPS), and use a HB→IPS converter to set HB references. Although the converter is a requirement for the classical controller due to its inability to adapt to each application's unique frequency→HB model, it is not a restriction on the LCT. Our design decision is made for fairness in the comparison, but the online learning done by the LCT allows it to define its objective in terms of HB directly, with the ability to adapt to different applications. A controller using a fixed model (e.g., classical controller) simply *cannot* model the relationship between application-level metrics to hardware-level knobs at runtime for a dynamic workload. The ability to specify an objective for a low-level controller in terms of an application-specific user-defined metric is a significant advantage when providing self-optimization.

The final advantage of self-optimization through reinforcement learning over classical control theory is the ability to minimize or maximize objectives, as opposed to achieve a fixed setpoint. Our example is a simple one with a single objective (HB), but still requires an achievable setpoint for the classical controller to behave desirably, meaning an optimizer is required to calculate the desired setpoint. The learner can minimize or maximize a given objective, integrating the functionality of an optimizer.

3.3 Hardware Efficiency

The LCT hardware implementation allows for efficient runtime reinforcement learning and has distinct advantages over software controllers. For one, the LCT has direct access to hardware sensors and actuators, enabling much shorter periods between invokations (epochs). Shorter epochs means that finer-grained actuations can be supported by the LCT. The LCT can still support more coarse-grained actuations over longer epochs that involve software sensors or actuators. For some actuations, short epochs only enabled by hardware are required to provide a sufficient sampling rate to support

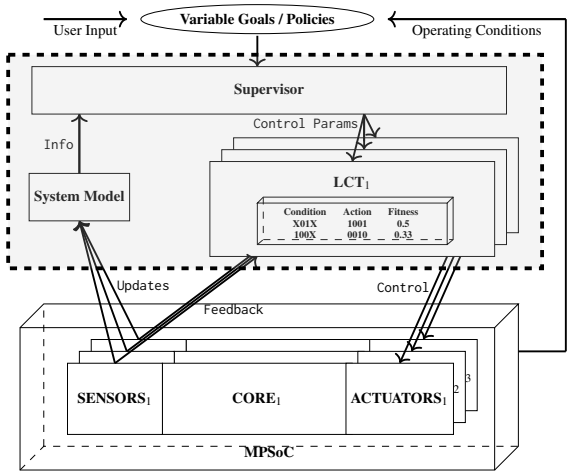


Figure 3: SOSA hierarchical overview on a multicore processor. SOSA components in the shaded region.

the learning. The optimal rules must be learned through numerous observations, which may take too long to be effective in a software implemented controller.

4 SOSA

SOSA is a hierarchical resource manager (Section 4.1) consisting of a high-level supervisor (Section 4.2) that guides distributed low-level controllers (Section 4.3) to achieve a global goal. We design the entire hierarchy in the context of our case study: managing QoS of a focus application within a power budget on an MPSoC (Section 4.4).

4.1 Hierarchical System Architecture

Figure 3 depicts a high-level view of SOSA for many-core system resource management. Either the user or the system software may specify *Variable Goals and Policies*. The *Supervisor* aims to achieve goals by managing the low-level controllers. High-level decisions are made based on the feedback given by the high-level model, or *System Model*, which provides an abstraction of the entire system. Low-level controllers are implemented as *LCTs*, which control the system via *actuators*. The supervisor provides objective-function parameters such as output references (i.e., target values) or constraints to each LCT during runtime according to the system policy. Actions taken by the LCTs indirectly update the system model through *senors* to maintain the global system state, and potentially trigger the supervisor to take action. The high-level model can be designed in various fashions (e.g., rule-based or estimator-based [20, 31, 43]) to track the system state and provide the supervisor with guidelines.

4.2 Supervisor

We use Supervisory Control Theory (SCT) [42] to design our supervisor similarly to [39]. SCT solves complex synthesis problems by breaking them into small-scale sub-problems, known as modular synthesis. The results of modular synthesis characterize the conditions under which decomposition is effective. In particular, results identify whether a valid decomposition exists. A decomposition is valid if the solutions to sub-problems combine to solve the original

problem, and the resulting composite supervisors are non-blocking and minimally restrictive. This horizontal decomposition in sub-problems allows us to divide the overall system into several smaller subsystems which are controlled by individual controllers.

Figure 4 illustrates how a supervisory control structure can hierarchically manage feedback control loops. As shown in the figure, supervision is vertically decomposed into tasks performed at different levels of abstraction [48]. The supervisory controller is designed to control the high-level *system model*, which represents an abstraction of the system. The *subsystems* compose the pre-existing *system* that does *not* meet the given specifications without the aid of a controller or a supervisor. The *information* channel provides information about the updates in the high-level model to the supervisory controller. Due to the fact that the system model is an abstract model, the controlling channel is an indirect *virtual control* channel. In other words, the control decisions of the supervisory controller will be implemented by controlling the *low-level controller(s)* through *control parameters*. Consequently, the low-level controller(s) can control one or multiple subsystems using the *control* channel and gather information via *feedback*. The changes in the subsystems can trigger *model updates* in the state of the high-level system model. These updates reflect the results of low-level controllers' controlling actions.

The scheme of Figure 4 describes the division of supervision into high-level management and low-level operational supervision. Virtual control exercised via the high-level control channel can be implemented by modifying control parameters to adaptively coordinate the low-level controllers, e.g., by adjusting their objective functions according to the system goal. The combination of horizontal and vertical decomposition enables us to not only physical divide the system into subsystems, but also to logically divide the sub-problems in any appropriate way, e.g., due to varying epochs (control invocation period) or scope. The important requirement of this hierarchical control scheme is control consistency and hierarchical consistency between the high-level model and the low-level system, as defined in the standard Ramadge-Wonham control mechanism [48]. For a detailed description of SCT, we refer the reader to [2, 42, 43, 48].

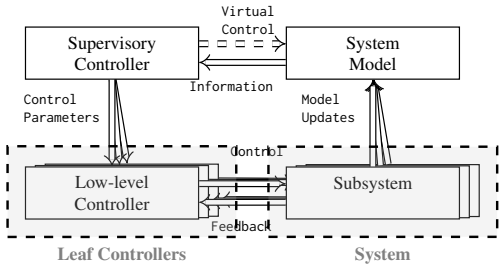


Figure 4: Supervisory Control structure. Low level control loops are guided by the Supervisory Controller that achieves system-wide goals based on the high-level system model.

4.3 Learning Classifier Table Low-Level Controllers

In general, a controller's task is to find the optimal actuation knob configuration for a given system state. The optimum is defined by some (measurable) metric, the objective function. The objective function (δ) can take one or more objectives ($\delta_1, \delta_2, \ldots, \delta_n$) into account

$$\delta\left(\delta_1, \delta_2, \ldots, \delta_n, \ldots\right) \tag{1}$$

to compare the desirability of different system states. In our hierarchical setup, the low level controllers' task is to execute actions that result in a more desirable *sub*system state over time according to the given objective function.

Zeppenfeld and Herkersdorf use learning classifier tables (LCTs) as low-level controllers in ASoC [52], which is an approach exploiting autonomic principles for runtime task distribution and frequency scaling. In ASoC, low-level controllers are unsupervised. The overall system goal emerges from the controllers' objective function definition and coordination.

LCTs are a subset of Wilson's XCS [51], which is a type of learning classifier system (LCS). LCSs describe the system they control by 'if `condition` then `action`' rules, i.e., classifiers. The `condition` corresponds to a specific set(s) of possible sensor values (i.e., system state). The `action` modifies the actual knob configuration to change system settings. Each rule contains a fitness value which describes its capability to improve the system state according to the objective function. A rule's fitness is updated each time the rule's condition matches the system state and its action is applied. The fitness is updated based on the rule's effectiveness toward achieving the objective. This is how the LCT builds and updates the system model, without any initial training required.

A set of rules, population [P], is needed to model system dynamics. Different rules may have overlapping conditions or actions. Figure 5 represents the general operating mode of LCS implemented as an

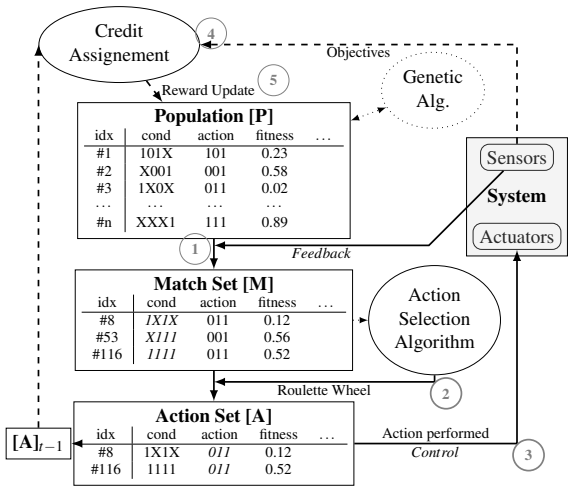


Figure 5: Overview of the LCT logic (dashed lines correspond to the fitness update path; dotted entities are part of LCSs, but not LCTs).

LCT. This operating mode consists of five periodically occurring steps:

(1) By comparing the sensor values to the conditions of all rules in the population [P], a match set [M] is generated. Conditions can contain wildcards to match several sensor values.
(2) Next, the roulette-wheel selection algorithm [50] decides on an action based on the fitnesses of the rules in [M]. All rules in [M] with the same action selected by the roulette wheel are saved as the action set [A].
(3) The selected action is forwarded to the actuators to apply it to the (sub)system. Further, the action set is saved $[A]_{t-1}$.
(4) After applying the action, the effect of the action is measured by sensors, and based on the effectiveness toward achieving the objective, some reward is given by the credit assignment component. The fitnesses of the rules in the saved action set $[A]_{t-1}$ get updated based on the reward according to a modified Q-learning which was proposed by Wilson [50] (see Equation 5).
(5) Finally, the updated rules are forwarded to the population [P] for the next evaluation.

To calculate the reward a rule receives, the credit assignment component observes changes of the objective function (δ). The objective function is calculated as normalized error. The error is the difference between the measured sensor value and its target value. For example, we define our objective function in terms of a performance metric ($PERF$):

$$\delta = \frac{|PERF - ref_{PERF}|}{max_{PERF}} \qquad \delta \in \{x \mid 0 \leq x \leq 1\} \tag{2}$$

This equation describes a performance optimization objective with a performance target (ref_{PERF}). We can further constrain this function, e.g., to maximize PERF within a power budget ($constr_{Power}$), by setting ref_{PERF} to a large value and subjecting the equation to

$$Power \leq constr_{Power} \tag{3}$$

The observed change in the objective function results in different rewards ($reward$) according to the following reward function:

$$reward = 1 - \delta \qquad reward \in \{x \mid 0 \leq x \leq 1\} \tag{4}$$

In the case that the constraint is violated, the reward is set to 0. This reward function with a discrete range supports the ability to distinguish between two different actions for the same condition which both improve or degrade the system state to varying degrees.

Based on the reward, the fitness (fit) of $[A]_{t-1}$ is updated by reinforcement learning using a modified Q-learning algorithm

$$fit \leftarrow fit + \beta \left[\left(reward + \gamma \cdot max \left(fit_{[A]_{t-1}} \right) \right) - fit \right] \tag{5}$$

with the learning rate parameter β and the discount factor γ. The discounting $\left(\gamma \cdot max\left(fit_{[A]_{t-1}}\right)\right)$ is omitted in single-step problems like DVFS [51], and therefore also in SOSA.

Our reward and fitness function prevents fitness values of constantly improving/degrading rules from ending up at the minimum or maximum value on the long term. The selected fitness update procedure does not necessary result in stable fitnesses over time for all kind of rules (e.g., general rules which include a lot of don't-cares for a single or multiple sensor inputs). Accuracy-based genetic algorithms are able to recognize unstable rules [51]. Genetic algorithms have also been used in LCS's to discover new rules. Identifying and removing unstable rules, and generating and testing new rules are not currently parts of the LCT, and therefore out of the scope of this work. We are addressing this in ongoing work in order to enable LCTs to add potentially beneficial rules to the ruleset and remove unstable rules during execution.

4.4 Case Study

Figure 6 shows an overview of our evaluation platform. We target a 4-core homogeneous CMP consisting of high-performance ARM cores. We consider a typical embedded scenario in which a performance-sensitive application (focus application) is running concurrently with various other (background) applications starting and stopping unpredictably. This mimics a typical use-case in which the main purpose of the device is performed in the foreground in conjunction with background debugging, reporting and logging.

The system goals are twofold: i) meet the performance requirement of the foreground application while minimizing its energy consumption; and ii) ensure the total system power always remains below the Thermal Design Power (TDP). In other words, the performance is a *target*, while the power is an *upper bound*. There is no advantage to exceeding the performance requirement.

We consider two actuation decisions: one to set the operating frequency and associated voltage of each core; and one to migrate tasks between cores. We measure the power consumption of each core, and simultaneously monitor the performance (in IPS) of the designated application to compare it to the required performance (IPS_{ref}).

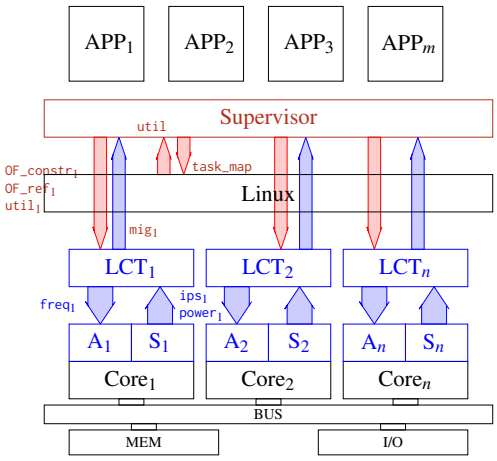


Figure 6: Example SOSA implementation on an MPSoC. Each core has an associated LCT, with local sensors (IPS, power, ...) and actuators (core frequency). The software supervisor communicates directly with LCTs to: (a) send global sensor data (per-core utilization); (b) update rules, or objective functions (targets, constraint); (c) receive hardware sensor data (migration request). In this example, the supervisor also communicates with Linux to receive software sensor data (utilization) and send software actuation commands (task migration).

Supervisory control attempts to meet the performance requirement while honoring the power budget. The supervisor prioritizes IPS and power in each **LCT** appropriately based on total system-wide power measurements. Supervisory control commands guide the LCTs to determine the operating frequency of each core and migrate tasks to the core. The LCTs set local core operating frequency directly via hardware actuator. The task migration actuator is implemented in software as part of the supervisor. The supervisor coordinates the migration flags sent by each LCT to perform global task migration.

An **objective function** (δ) is used to set the optimization objective of the LCTs. The objective functions are defined in terms of IPS or power, each with a constraint (see Equations 2 and 3). We define two objective functions for this case-study: 1) *IPS-oriented* function ensures that the focus application can meet the performance target value, and 2) *Power-oriented* function limits the power consumption while possibly sacrificing some performance if the system is exceeding the power budget threshold.

We use the **reward function** to enforce the constraints ($constr_{IPS}$, $constr_{Power}$). For the IPS-oriented objective function (δ_{IPS}), we set the reward to 0 when $IPS < constr_{IPS}$; for the Power-oriented objective function (δ_{Power}), we set the reward to 0 when $Power > constr_{Power}$. This has the effect of "forbidding" the violation of the desired system state in each respective objective function. Subsequently, we embed our optimization within the learning mechanism: for δ_{IPS}, we set $ref_{Power} = 0$, so that while the objective function achieves ref_{IPS}, it also minimizes *Power*. The result is the following objective functions:

$$\delta_{IPS} = \frac{Power}{max_{Power}}, \text{ subject to } IPS \geq constr_{IPS} \tag{6}$$

$$\delta_{Power} = \frac{|IPS - ref_{IPS}|}{max_{IPS}}, \text{ subject to } Power \leq constr_{Power} \tag{7}$$

5 HARDWARE IMPLEMENTATION

We implement SOSA in hardware on a Xilinx Virtex®-7 FPGA using Gaisler's SPARCv8 library (GRLIB) to validate the design as well as verify the feasibility.

5.1 Hardware Setup

For the hardware evaluation, we use a three-core Leon3 system with per-core LCTs and a single supervisor. This setup features a similar multilayer architecture as shown in Figure 6 with two key differences: (1) we execute applications as bare-metal code without an operating system, and (2) we implement the supervisor in hardware instead of software.

The supervisor communicates directly with LCTs to (a) update rules or objective functions and (b) receive hardware sensor data. The hardware implementation uses application specific registers (ASRs) to communicate between software and SOSA hardware. The ASRs primarily support initialization of LCTs at runtime, debugging, and measurement for evaluation. The ASRs are also used to implement the task migration actuator. Hardware LCTs are invoked periodically every 5 ms, and the supervisor is interrupt-driven by the LCT, meaning it can also be invoked up to every 5 ms.

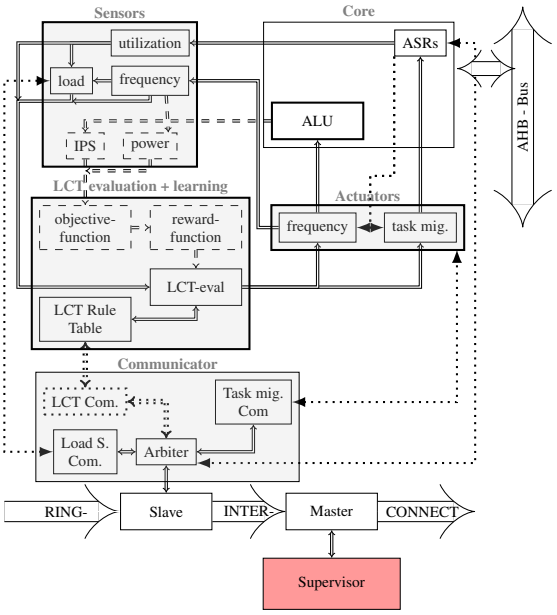


Figure 7: Hardware structure of a single LCT including communication connections to its local core and the dedicated ring interconnect. LCT components are shaded.

Figure 7 shows the block diagram of a single core and its LCT. The bus, ARM's Advanced Microcontroller Bus Architecture Advanced High-performance Bus (AHB), links each core to the other cores of the system, the memory, etc. Two kinds of sensors are associated with each subsystem: one to determine the condition of the subsystem (i.e., frequency, utilization, and relative workload to the other cores in the MPSoC), and one to evaluate the current performance of the core with respect to the objective function, which is defined in terms of power consumption or IPS. For differentiation, condition sensors are solid and objective sensors are dashed in Figure 7 (similarly to Figure 5). The high-level system model is updated at regular time intervals. LCT actuators consist of frequency scaling and task migration.

The low-level controller includes a communication interface in addition to the entities necessary for learning. The communication interface is used to negotiate task migrations in hardware, to provide shared sensor values like workload, and to get updates from the supervisor. All communication between LCTs and the supervisor is transmitted over a dedicated autoring [5] ring-interconnect.

5.1.1 Actuators. Due to the lack of an operating system, task migration must be emulated. Instead of performing actual migration, we maintain an ASR on each core for task scheduling, with one bit of the ASR dedicated to each task in the entire system. If a task's bit is set in a core's scheduler ASR, this task gets executed in the core's next scheduling epoch. Additionally, because our implementation is not globally asynchronous locally synchronous (GALS), frequency scaling of individual cores is not possible. To emulate frequency scaling, the core pipeline is stalled periodically by a pulse density modulated (PDM) signal.

5.1.2 Sensors. The frequency sensor is implemented simply by forwarding the current actuation setting (see Figure 7). The IPS sensor directly accesses the trace port of its corresponding Leon3 core and counts the changes of the program counter within a specific time period. For simplification, our power model assumes the power is directly proportional to the product of frequency and utilization ($P \propto f \cdot u$). Therefore, we encode the power value as the product of the current frequency and utilization values as an approximation, since we are only concerned with relative power values (rather than absolute).

5.2 Functional Validation and Design Feasibility

The execution scenario consists of a synthetic benchmark executed on the system. The benchmark mimics an IP-forwarding application, in which packets requiring varying computational effort are continuously received. The varying effort required by input invokes unpredictable changes in workload. The application consists of nine tasks in total: five perform the packet processing, one generates the changing computational effort, and three are for debugging (e.g., sending/receiving UART messages, providing sensor values to the UART task, and interpreting received UART messages). The data provided over UART to the host PC includes information about the current system state, and is used for evaluation.

We use the same objective functions as described in Section 4.4 (δ_{IPS} and δ_{Power}): in *IPS-oriented* mode, there is an IPS constraint higher than the IPS target in *Power-oriented* mode and the objective function is equal to the measured power in order to always meet IPS but minimize power; in *Power-oriented* mode, there is a power constraint equal to the power budget and the objective function is equal to the IPS target in order to always honor the power budget but achieve some target IPS.

5.2.1 Validation. The graphs in Figure 8 show power and IPS achieved by the HW implementation of SOSA. We subject SOSA to three different scenarios.

In the first scenario (seconds 0-66 of Figure 8), we set a target IPS that is achievable within the power budget for our workload. Observe in Figure 8a that SOSA in the first 33 seconds of execution is able to track the target IPS with minimal error (we define IPS error as the *% below* the IPS target), well within the power budget. Partway through execution, we inject disturbance in the form of additional tasks (seconds 33-66). Notice the increase in power in Figure 8b. The IPS is visibly degraded, and there is a small amount of consistent error, but overall SOSA's learners adjust. This demonstrates that SOSA can adjust correctly to dynamic workload.

In the second scenario (seconds 67-100 of Figure 8), we emulate a system emergency by lowering the power budget such that the target IPS is no longer achievable within the budget. Observe in Figure 8b, that the power constraint of the LCTs successfully prevents the power budget from being violated. Instead, SOSA finds an operating point close to the budget in order to maximize the IPS without violations. This demonstrates that SOSA can (relatively strictly) abide by the power budget, even in extreme scenarios.

In the third and final scenario (seconds 100-115), we change the objective function in the LCTs to minimize power while requiring the IPS target to be met. Observe in Figure 8a that the IPS is easily surpassed within the original power budget. In fact, even the reduced power budget (Figure 8b) is only slightly violated while meeting or even exceeding the target IPS. By setting the power target to 0 in the LCT with an IPS constraint, the LCT optimizes the operating point. This demonstrates that SOSA can integrate optimization simply using its low-level controllers (LCTs).

Overall, we confirm SOSA's ability to both adapt and optimize to internal (workload) and external (environmental) dynamism, while always honoring the specified goals.

5.2.2 Overhead. Our hardware evaluation platform adds 9.62 % of slices compared to a standalone MPSoC design. The LCTs are in total only 18 % of this overhead. The overhead, and therefore the additional cost, can be significantly reduced by using senors and actuators already present in state-of-the-art MPSoCs. In terms of performance impact on the executed application, SOSA (1) does not invoke any delay on the workload and (2) supports shorter duty cycles than a software controller. Due to the hardware implementation of the low-level controllers (LCTs), the invocation period is only limited by the observation time required to provide accurate sensor values, and the actuation time to affect the system.

6 SIMULATION EVALUATION

In this secion, we demonstrate SOSA's ability to self-adapt and self-optimize by building a model at runtime. We compare to control-theoretic-based resource managers.

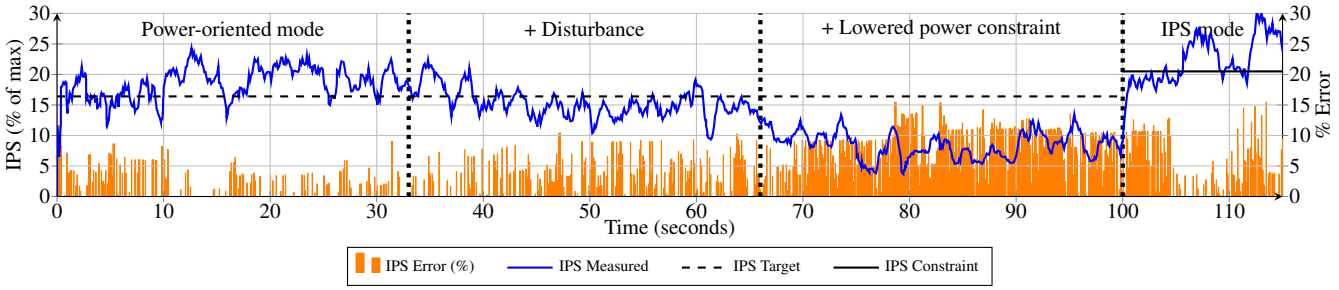
6.1 Simulation Setup

We perform our evaluations for the platform described in our case study (Figure 6) using the gem5 architectural simulator in full-system mode. We implement all resource management software using the MARS middleware framework[1] [32] for Linux, and LCTs are implemented as gem5 simulated hardware modules, mimicking the hardware design as closely as possible. The MARS Linux userspace daemon process invokes the supervisor every 100 ms. The supervisor communicates directly with LCTs using memory-mapped I/O to: (a) send software sensor data; (b) update rules or objective functions; (c) receive hardware sensor data; (d) receive software actuation commands (task migration). LCTs are invoked every 10 ms. We use a combination of ARM's Performance Monitor Unit (PMU) and simulator hardware sensors for the performance and power measurements required by the resource managers. The IPS performance target is provided by the focus application. The learning is done in the gem5 simulated hardware modules, whereas their actions are carried out by the OS (Linux) for the task migration and by the gem5 simulated hardware for the frequency changes. This accounts for the timing and performance overhead of actuations. The frequency of task migration is inherently limited: we allow a maximum of one task migration each time the migration policy is invoked.

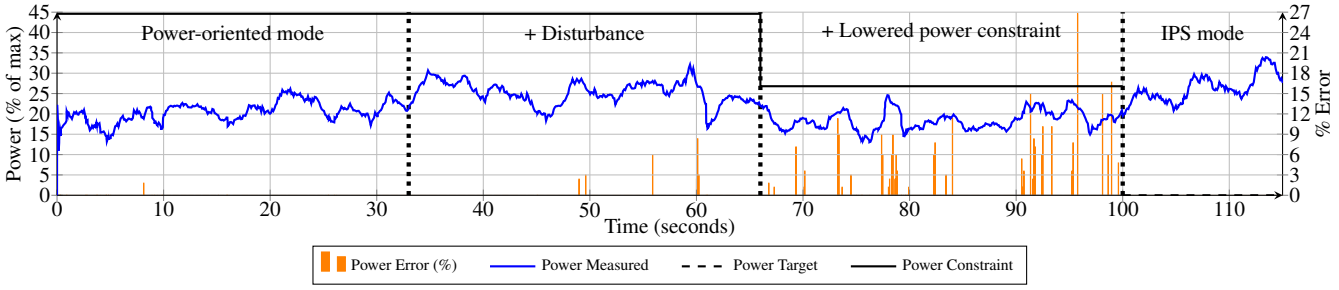
The resource managers for comparison use control-theoretic low-level controllers designed with the Matlab System Identification Toolbox [28].[2] We generate training data by executing a microbenchmark (from MARS) and varying control inputs in the format of a staircase test (i.e., a sine wave). The microbenchmark consists of a

[1] https://github.com/duttresearchgroup/MARS

[2] We generate the models with a stability focus. All systems are stable according to Robust Stability Analysis. We use Uncertainty Guardbands of 50 % for IPS and 30 % for power, as in [35].



(a) HW-SOSA instructions-per-second (IPS) achieved for a packet-forwarding benchmark. We measure error as the difference between measured IPS and target IPS as % of the max IPS value possible, only when measured IPS is *below* the target.



(b) HW-SOSA power. We measure error as the difference between measured power and power budget / constraint as % of the max power possible, only when measured power is *above* the bound.

Figure 8: IPS and power tracking on HW for a synthetic benchmark in bare-metal. After second 33, background tasks are introduced. After second 66, the power budget is reduced. After second 100, IPS is prioritized. The plots of measured values have been smoothed for clarity, but error is calculated based on the raw (finer-grained) data. This is why in some cases the error appears to differ from the displayed difference between target and measured values (e.g., power error shown when it appears the budget is never violated).

sequence of independent multiply-accumulate operations performed over both sequentially and randomly accessed memory locations, thus yielding various levels of instruction-level and memory-level parallelism. The range of exercised behavior resembles or exceeds the variation we expect to see in typical embedded workloads, which is the focus application domain of our case studies.

We use the following focus applications to evaluate the resource managers: `k-means` clustering, k-nearest neighbors classification (`knn`), and linear regression (`linreg`) machine-learning kernels; and `blackscholes` from the PARSEC [3] benchmark suite. To compose a workload, we launch four instances of the focus application to emulate data-parallel multithreading.

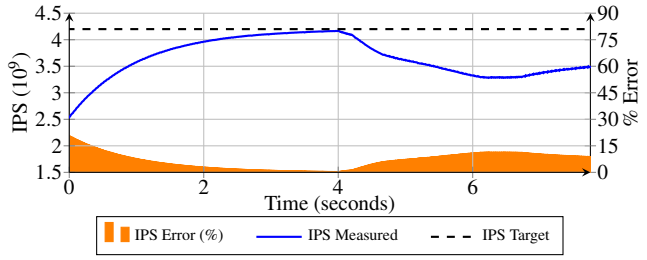
6.2 Model-independence Evaluation

To show SOSA's ability to identify system dynamics from scratch, we study its ability to track a fixed goal for a fixed workload. Our execution scenario consists of a single focus application with an achievable IPS target within the power budget. The goal is to meet target IPS. We compare to a baseline resource manager (BASE) which uses SISO controllers, one for each core. The SISOs have a frequency control-input and IPS measured-output (Frequency→IPS), and are invoked every 10 ms. BASE includes a software supervisor to provide target coordination to the SISOs as well as a simple task migration heuristic. Task migration is performed at the same frequency as low-level controllers are invoked. In all of our evaluation scenarios, we seek to distribute the total system utilization equally among cores. Therefore, our focus applications consist of four threads, and our target for each LCT are one quarter of the total system target. All threads are initially mapped to the same core, so the resource manager is completely responsible for migration. Distribution of system-wide budgets and targets is the subject of orthogonal research, e.g., [6].

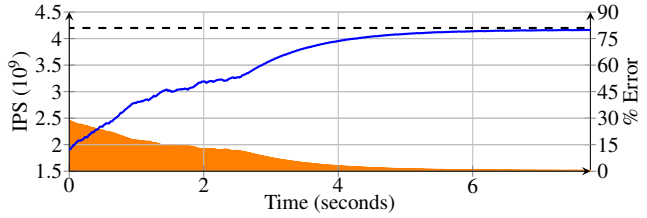
Figure 9 shows SOSA's and BASE's ability to track a fixed IPS target for a fixed workload. The first 4 s of execution consist of only the focus application `k-means`. Figure 9a demonstrates the classical controller's ability to achieve the target performance in an ideal execution scenario, based on the system model identified at design time. Figure 9b demonstrates the LCT controller's ability to achieve the target performance by learning the system dynamics during execution, without prior workload observation.

6.3 Self-optimization Evaluation

To show SOSA's ability to identify system dynamics at runtime for unpredictable workloads, we study its ability to track a fixed goal for a dynamic workload. Our execution scenario consists of a single focus application with background tasks entering in the middle of execution (to induce interference from other tasks). The goal is to meet target IPS, which is achievable within the power budget. We compare again to BASE.



(a) BASE instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.

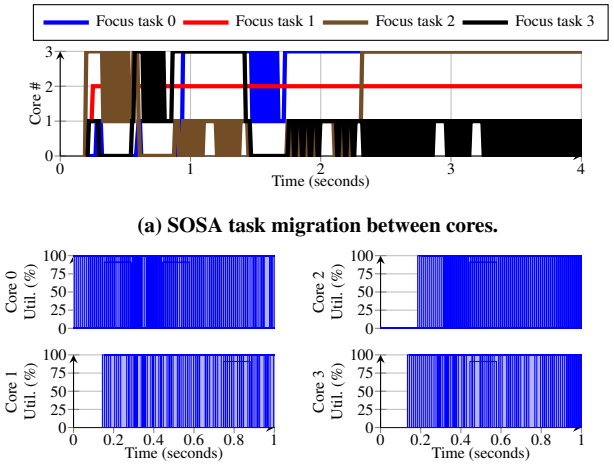


(b) SOSA instructions-per-second (IPS). Error measured as % of the max power possible *below* target.

Figure 9: IPS tracking for `k-means`. First four seconds consist only of `k-means`. After four seconds, microbenchmarks are added for disturbance. Measured values are smoothed using averaging.

Figure 9 shows SOSA and BASE ability to track a fixed IPS target for a dynamic workload with tasks coming and going unpredictably. After 4 s of execution, backgrounds tasks are started, consisting of single-threaded microbenchmarks. Figure 9a shows that once disturbance is introduced, the controller experiences up to 14 % performance degradation. Figure 9b demonstrates the LCT controller's ability to self-optimize to find a configuration that achieves IPS with <5 % error once disturbance is introduced in the form of workload variability, eventually settling with error <1 %. We confirm SOSA's ability to learn the model from scratch at runtime and optimize in the face of disturbance are in line with our hardware evaluation.

6.3.1 Global Coordination. Figure 10a shows the migration of focus tasks between cores by SOSA over the first 4 s of execution of the `k-means` benchmark. Each task is represented by a different color line, and the y-axis represents the four different cores. Though it is difficult to see, all tasks are to be initially mapped to Core 0. Observe that over time, the migration policy spreads the tasks among all four cores. There are some exploration periods (e.g., in the first second), and some tasks experience oscillating migration more than others (e.g., task 3 from 2-4 seconds), but the policy accomplishes its goal of spreading the utilization out among the cores. Figure 10b shows the focus task utilization of each of the four cores through the first 2 s of execution. Observe that each core is utilized completely for almost the entire execution. After all focus tasks are initialized on the same core, they are migrated within the first 250 ms. This confirms that the migration policy achieves its goal.



(a) SOSA task migration between cores.

(b) SOSA per-core focus-task utilization.

Figure 10: Migration and utilization of SOSA for `k-means`.

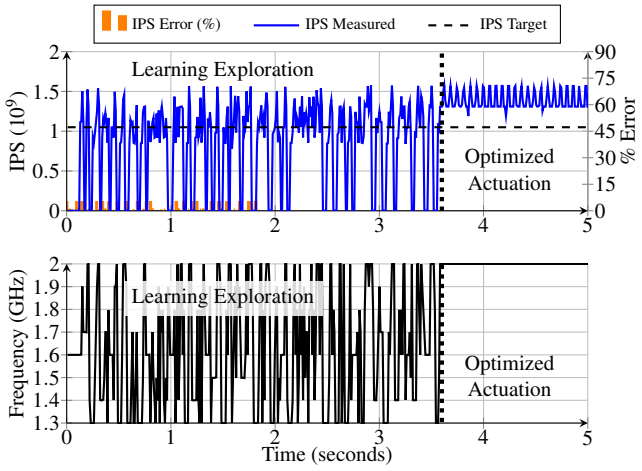
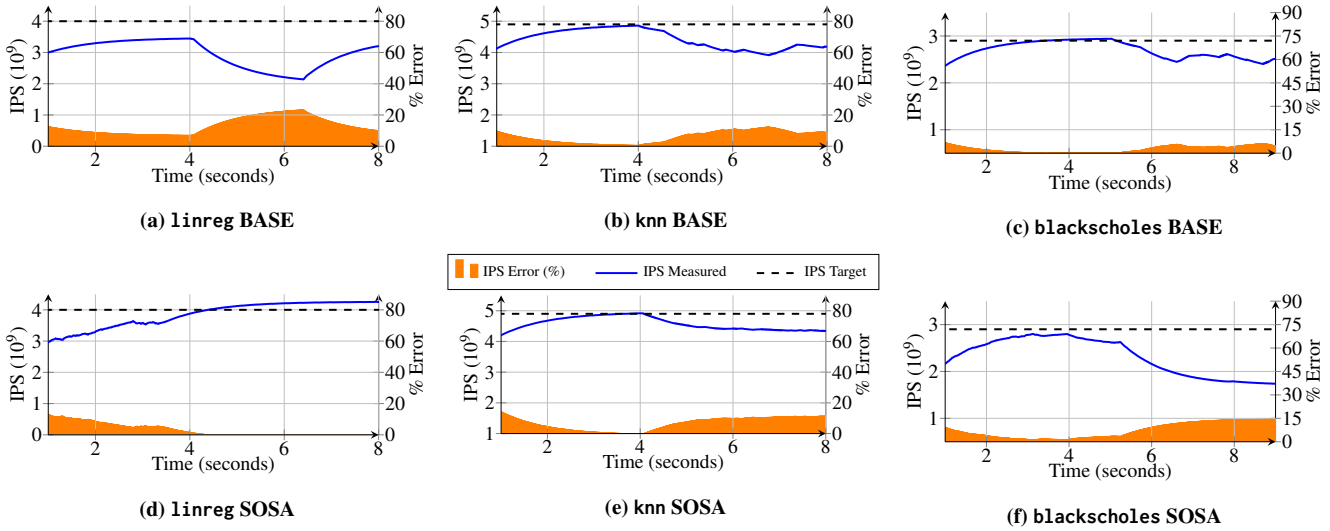


Figure 11: Single-core optimization of SOSA for `k-means` over the first 5 s of execution for Core 3. The top plot shows the core IPS achieved, and the bottom plot shows the core frequency.

6.3.2 Local Optimization. In Figure 11 we take a closer look at the behavior of a single core through the first 5 s of `k-means` controlled by SOSA. Observe the variation in core frequency displayed in the lower plot. The LCT explores the configuration space through the first 3.5 s of execution. After 3.5 s, the LCT settles on the maximum frequency in order to achieve the target IPS for the remainder of execution. This is in line with our system-wide observations made in Figure 9b.

6.3.3 Other Benchmarks. Figure 12 shows the self-optimization evaluation execution scenario for three additional focus applications.



(a) `linreg` BASE (b) `knn` BASE (c) `blackscholes` BASE

(d) `linreg` SOSA (e) `knn` SOSA (f) `blackscholes` SOSA

Figure 12: Additional benchmarks.

SOSA's result for `linreg` (Figure 12d) mirrors that of `kmeans`: learning occurs for the first ~3 s, after which the target IPS is achievable even through disturbance. The BASE manager struggles significantly with managing through the disturbance (Figure 12a). The performance degradation from 4-6 seconds is due to poor migration decisions – the BASE manager eventually recovers at 6 s.

SOSA and BASE perform similarly for `knn` (Figure 12e,12b): both managers achieve target IPS at a comparable rate in the first 4 s. After disturbance is introduced, both managers are unable to achieve target IPS. Eventually SOSA settle's at 14 % error, and BASE at 9 %.

The results for `blackscholes` (Figure 12f,12c) are opposite of `linreg`: in this case, SOSA struggles to achieve target IPS once background tasks are introduced. Although BASE also experiences degradation of performance after 4 s, SOSA's error is 2× higher. SOSA's performance degradation from 4-6 seconds is due to poor migration decisions.

6.4 Self-adaptivity Evaluation

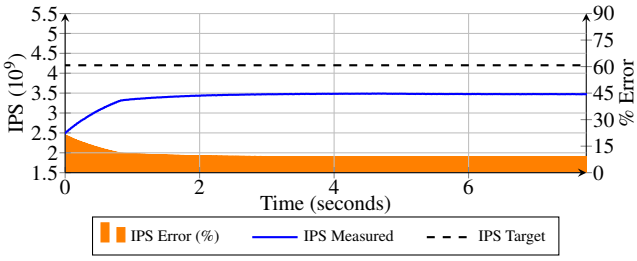
To show SOSA's ability to adapt to changing operating conditions, we study its ability to track a changing goal for a fixed workload. Our execution scenario consists of two different phases of execution:

(1) *Normal Phase*: In this phase, only the focus application executes. The goal is to meet target IPS and minimize power consumption.
(2) *Low-power Phase*: In this phase, the IPS target remains the same as that in the Normal Phase while the power budget is reduced. The goal is to prioritize honoring the power budget while maintaining target IPS (if possible).

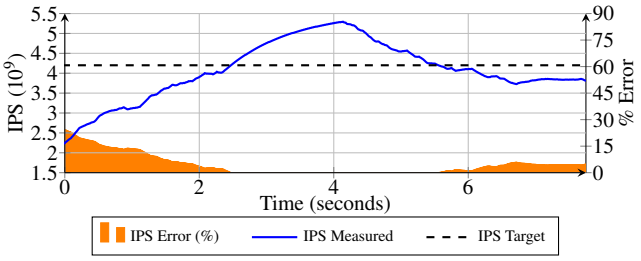
We compare to a resource manager (SOTA) that uses per-core SISOs (Frequency→IPS), with a self-adaptive software supervisor to provide target coordination and a simple task migration heuristic. SOTA consists of a self-adaptive supervisor and classical low-level controllers similarly to [39], therefore representative of a state-of-the-art resource manager. Our goal is to demonstrate that SOSA's self-adaptivity is on-par with state-of-the-art alternatives.

Figure 13 shows SOSA's and SOTA's ability to track the IPS target while honoring the power budget in the Normal Phase and Low-power Phase. The first 4 s of execution are in the Normal Phase. Observe the IPS of both managers (Figure 13a and 13b). SOTA reaches its peak value after 1 s of execution, maxing out at 9 % below the IPS target. SOSA's learning takes longer – it spends the first 2.5 s exploring the configuration space, and reaches peak value at 4 s. However, SOSA is able to match SOTA's peak IPS value after only 1.5 s, and reaches a peak value above the IPS target and more than **1.5**× that of SOTA, all within the power budget. We make two specific observations in SOSA's Normal Phase regarding the power (Figure 13d). First, the power spends the first 2.5 s with substantial noise, indicating exploration for learning. Second, the settling of the power around the 2.5 s mark indicates that SOSA has learned a meaningful model. This is reinforced by our observations about the IPS. SOTA's lower IPS peak and continuous power noise is due to excessive task migration. The task migrations negatively affect the utilization of each core by the focus tasks, limiting the maximum achieved IPS and causing dips in the power (i.e., the low ends of the spikes).

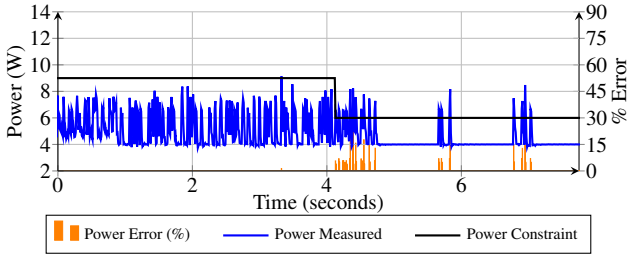
After 4 s, the execution enters the Low-power Phase, and the power budget is reduced by half. SOTA continues attempting configurations to increase IPS, as it is still not reaching the target. After 0.5 s, the controller responds to the change in priority, and it finds a configuration that honors the new budget without IPS degradation. In Figure 13c we see that the SOTA controller periodically experiences large power spikes that violate the budget. In the Low-power phase, SOSA's configuration violates the power budget, causing to LCTs switch to low-power models, which have yet to be populated. This explains the degradation in IPS (Figure 13b), as well as the noise in the power (Figure 13d). Observe that at 7 s, both the IPS and power begin to stabilize around their targets, with ~10 % error. Looking closely, just before 8 s, the power for SOSA dips significantly, indicating that the LCTs are continuously updating and acting in an attempt to honor the power constraint. Based on the



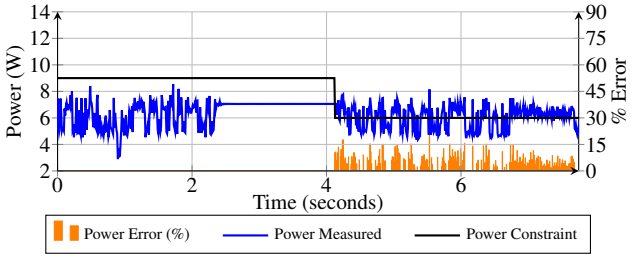
(a) SOTA instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.



(b) SOSA instructions-per-second (IPS). Error measured as % of the max IPS value possible *below* target.



(c) SOTA power. Error measured as % of the max power possible *above* budget.



(d) SOSA power. Error measured as % of the max power possible *above* budget.

Figure 13: IPS and power for `k-means` with a power budget change after 4 s. IPS values are smoothed using averaging.

learning observed in the self-optimization evaluation, as well as the Normal Phase, we believe that SOSA will have a populated model that will lead to efficient decisions. We also think that this presents an opportunity for applying transfer learning, as there are clearly some rules and configurations that are desirable for both Normal and Low-power Phases.

In conclusion, both resource managers are able to consider dynamic goals, balancing IPS targets while accounting for a power budget. The SOSA LCTs take time to learn new objectives compared to pre-populated models, but they are able to learn from undesirable configurations, and as a result SOSA considerably outperforms SOTA, which struggles to coordinate migration and DVFS.

7 CHALLENGES AND FUTURE WORK

For this work, we assume a straightforward use-case to demonstrate the fundamental capabilities of our resource manager. However, one could imagine more challenging scenarios in the real world, some of which we have identified throughout the manuscript. Here we identify two challenges that are the subject of ongoing work to improve our SOSA manager. The first aspect to address is the uniformity of the workload and resource budgets in our use-case. Realistic workload scenarios may have multiple performance demands concurrently from different applications, with asymmetric task loads and task-dependence. There has been a large amount of research in the area of budget allocation that would be advantageous to incorporate. The second aspect we are addressing is the static ruleset. With a static ruleset, generating a ruleset that produces favorable results requires designer knowledge as well as empirical analysis. There exist methods in LCS for expanding and pruning rulesets over time automatically, and we plan to incorporate them. We believe that with a good runtime algorithm, we could remove even more designer-dependence from this methodology.

8 CONCLUSION

In this paper, we set out to design a resource manager for embedded many-cores that can self-adapt when faced with changing goals due to external stimuli, and self-optimize by learning dynamic workloads at runtime. To this end, we propose SOSA, a hierarchical resource manager with a high-level supervisory controller that provides self-adaptivity and goal management by guiding distributed low-level LCT controllers that self-optimize subsystems by continuously updating runtime models. Our hardware implementation of SOSA in a Leon3-based MPSoC for an FPGA validates both the functionality and feasibility of the hierarchical design. We evaluate an implementation of SOSA designed to manage a performance target within a power budget. We implement SOSA in Linux and gem5 in order to compare to state-of-the-art alternatives for handling (1) unpredictable workload disturbance and (2) unpredictable changes in constraints.

ACKNOWLEDGEMENTS

The authors would like to thank Thomas Wild and Anmol Surhonne for their valuable feedback on the manuscript. We acknowledge financial support from DFG grant HE4584/7-1, NSF grant CCF-1704859, and the Marie Curie Actions of the European Union's H2020 Programme.

REFERENCES

[1] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing Cache Performance Under Uncertainty. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[2] Murray Wonham Bertil A. Brandin and Beno Benhabib. 1991. Discrete Event System Supervisory Control Applied to the Management of Manufacturing Workcells. In *Computer-Aided Production Engineering, C. Venkatesh and J.A. McGeough, eds. (Amsterdam: Elsevier)*.

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[4] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *International Symposium on Microarchitecture (MICRO)*.

[5] Abdelmajid Bouajila, Abdallah Lakhtel, Johannes Zeppenfeld, Walter Stechele, and Andreas Herkersdorf. 2012. A low-overhead monitoring ring interconnect for MPSoC parameter optimization. In *International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*.

[6] Zhuo Chen and Diana Marculescu. 2015. Distributed Reinforcement Learning for Power Limited Many-core System Performance Optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[7] Seungryul Choi and Donald Yeung. 2006. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *International Symposium on Computer Architecture (ISCA)*.

[8] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *International Symposium on Microarchitecture (MICRO)*.

[9] Luis Costero, Arman Iranfar, Marina Zapater, Francisco D Igual, Katzalin Olcoz, and David Atienza. 2019. MAMUT: Multi-Agent Reinforcement Learning for Efficient Real-Time Multi-User Video Transcoding. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[10] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *International Conference on Autonomic Computing (ICAC)*.

[11] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *International Symposium on Microarchitecture (MICRO)*.

[12] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *International Symposium on Computer Architecture (ISCA)*.

[13] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. 2016. SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

[14] B. Donyanavard, A. M. Rahmani, T. Muck, K. Moazemmi, and N. Dutt. 2018. Gain Scheduled Control for Nonlinear Power Management in CMPs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[15] Christophe Dubach, Timothy M. Jones, and Edwin V. Bonilla. 2013. Dynamic Microarchitectural Adaptation Using Machine Learning. *Transactions on Architecture and Code Optimization (TACO)* (2013).

[16] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. 2010. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *International Symposium on Microarchitecture (MICRO)*.

[17] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[18] Ujjwal Gupta, Joseph Campbell, Umit Y. Ogras, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, and Suat Gumussoy. 2016. Adaptive performance prediction for integrated GPUs. In *International Conference On Computer Aided Design (ICCAD)*.

[19] Vinay Hanumaiah, Digant Desai, Benjamin Gaudette, Carole-Jean Wu, and Sarma Vrudhula. 2014. STEAM: A Smart Temperature and Energy Aware Multicore Controller. *Transactions on Embedded Computing Systems (TECS)* (2014).

[20] João Pedro Hespanha. 2011. Tutorial on Supervisory Control. In *Lecture Notes for the workshop Control using Logic and Switching for the Conference on Decision and Control*.

[21] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. 2013. A generalized software framework for accurate and efficient management of performance goals. In *International Conference on Embedded Software (EMSOFT)*.

[22] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[23] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. 2006. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *International Symposium on Microarchitecture (MICRO)*.

[24] Hwisung Jung, Peng Rong, and Massoud Pedram. 2008. Stochastic modeling of a thermally-managed multi-core system. In *Design Automation Conference (DAC)*.

[25] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture (MICRO)*.

[26] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *International Symposium on Computer Architecture (ISCA)*.

[27] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. 2010. Controlling software applications via resource allocation within the heartbeats framework. In *Conference on Decision and Control (CDC)*.

[28] MathWorks. 2017. *System Identification Toolbox*. Technical Report. https://www.mathworks.com/products/sysid.html

[29] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Kasra Moazzemi, Biswadip Maity, Saehanseul Yi, Amir M. Rahmani, and Nikil Dutt. 2019. HESSLE-FREE: Heterogeneous Systems Leveraging Fuzzy Control for Runtime Resource Management. *Transactions on Embedded Computing Systems (TECS)* (2019).

[31] Stephen Morse. 1977. *Control using logic-based switching*. Springer.

[32] T Muck et al. 2019. Adaptive-Reflective Middleware for Power and Energy Management in Many-Core Heterogeneous Systems. *Many Core Computing: Hardware and Software, IET* (2019).

[33] T. R. Mück, B. Donyanavard, K. Moazzemi, A. M. Rahmani, A. Jantsch, and N. D. Dutt. 2018. Design Methodology for Responsive and Robust MIMO Control of Heterogeneous Multicores. *Transactions on Multi-Scale Computing Systems (TMSCS)* (2018).

[34] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. 2013. Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems. In *International Symposium on Computer Architecture (ISCA)*.

[35] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. 2016. Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures. In *International Symposium on Computer Architecture (ISCA)*.

[36] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. 2018. Yukta: Multilayer Resource Controllers to Maximize Efficiency. In *International Symposium on Computer Architecture (ISCA)*.

[37] Raghavendra Pradyumna Pothukuchi and Josep Torrellas. 2016. *A Guide to Design MIMO Controllers for Architectures*. Technical Report. http://iacoma.cs.uiuc.edu/iacoma-papers/mimoTR.pdf

[38] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *International Symposium on Computer Architecture (ISCA)*.

[39] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil Dutt. 2018. SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[40] Amir M. Rahmani, M. Hashem Haghbayan, Anil Kanduri, Awet Y. Weldezion, Pasi Liljeberg, Juha Plosila, Axel Jantsch, and Hannu Tenhunen. 2015. Dynamic power management for many-core platforms in the dark silicon era: A multi-objective control approach. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

[41] Amir M. Rahmani, M. Hashem Haghbayan, Antonio Miele, Pasi Liljeberg, Axel Jantsch, and Hannu Tenhunen. 2017. Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era. In *Transactions on Very Large Scale Integration Systems (TVLSI)*.

[42] Peter J. Ramadge and W. Murray Wonham. 1989. The Control of Discrete Event Systems. In *Proceedings of the IEEE*.

[43] Michael H. Safanov. 1997. *Focusing on the knowable: Controller invalidation and learning*. Springer.

[44] Sina Shahhosseini, Kasra Moazzemi, Amir M. Rahmani, and Nikil Dutt. 2017. Dependability evaluation of SISO control-theoretic power managers for processor architectures. In *Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*.

[45] Sina Shahhosseini, Kasra Moazzemi, Amir M. Rahmani, and Nikil Dutt. 2018. On the feasibility of SISO control-theoretic DVFS for power capping in CMPs. *Microprocessors and Microsystems* (2018).

[46] Elham Shamsa, Anil Kanduri, Amir M. Rahmani, Pasi Liljeberg, Axel Jantsch, and Nikil Dutt. 2019. Goal-driven autonomy for efficient on-chip resource management: Transforming objectives to goals. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[47] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. 2012. A Case for Coordinated Resource Management in Heterogeneous Multicore Platforms. In *International Symposium on Computer Architecture (ISCA)*.

[48] John Thistle. 1996. Supervisory control of discrete event systems. In *Mathematical and Computer Modelling*.

[49] Augusto Vega, Alper Buyuktosunoglu, Heather Hanson, Pradip Bose, and Srinivasan Ramani. 2013. Crank It Up or Dial It Down: Coordinated Multiprocessor Frequency and Folding Control. In *International Symposium on Computer Architecture (ISCA)*.

[50] Stewart W Wilson. 1994. ZCS: A zeroth level classifier system. *Evolutionary computation* 2, 1 (1994), 1–18.

[51] Stewart W Wilson. 1995. Classifier fitness based on accuracy. *Evolutionary computation* 3, 2 (1995), 149–175.

[52] Johannes Zeppenfeld, Abdelmajid Bouajila, Walter Stechele, and Andreas Herkersdorf. 2008. Learning Classifier Tables for Autonomic Systems on Chip. *GI Jahrestagung (2)* 134 (2008), 771–778.

[53] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.