Tangram: Integrated Control of Heterogeneous Computers

Raghavendra Pradyumna Pothukuchi
pothuku2@illinois.edu
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA

Joseph L. Greathouse
Karthik Rao
Christopher Erb
Leonardo Piga
First.Last@amd.com
Karthik.Rao2@amd.com
Advanced Micro Devices, Inc.
Austin, Texas, USA

Petros G. Voulgaris
Josep Torrellas
voulgari,torrella@illinois.edu
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA

ABSTRACT

Resource control in heterogeneous computers built with subsystems from different vendors is challenging. There is a tension between the need to quickly generate local decisions in each subsystem and the desire to coordinate the different subsystems for global optimization. In practice, global coordination among subsystems is considered hard, and current commercial systems use centralized controllers. The result is high response time and high design cost due to lack of modularity.

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

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Hardware →** *Platform power issues*; *Chip-level power issues*; *Temperature control*.

KEYWORDS

Distributed resource management, formal control, heterogeneous computers, modular control.

ACM Reference Format:
Raghavendra Pradyumna Pothukuchi, Joseph L. Greathouse, Karthik Rao, Christopher Erb, Leonardo Piga, Petros G. Voulgaris, and Josep Torrellas. 2019. Tangram: Integrated Control of Heterogeneous Computers. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358285

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358285

1 INTRODUCTION

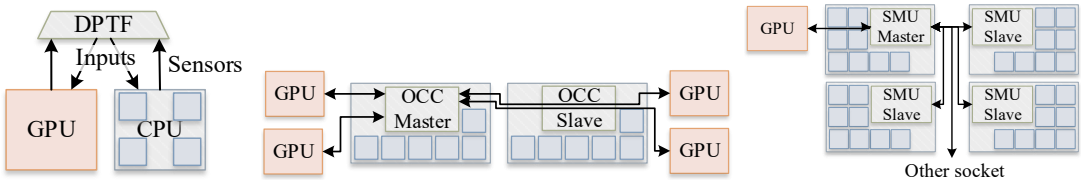
An emerging trend in today's computing systems is to integrate subsystems built by different vendors into heterogeneous computers [10, 45, 47, 55, 77, 79, 86]. Such subsystems can be CPUs, GPUs, or various accelerators. This approach is attractive because the individual components are often easier and cheaper to develop separately, and they can be reused across multiple products. For example, the same GPU design is used in AMD's Ryzen™ mobile processors [8] and in Intel's multi-chip Core i7-8809G [30].

Such heterogeneous systems, like other computers, need a resource control system — a vital unit that keeps the execution efficient and safe. Resource controllers attain efficiency by customizing the usage of limited resources like energy and storage to match application requirements. They also protect the hardware from hazardous conditions like high temperature or high current variation with time (dI/dt) [40]. Computers today are increasingly being equipped with microcontrollers that monitor execution conditions, run control algorithms, and actuate a set of configurable parameters in the computer [15, 17, 19, 41].

Building resource controllers is challenging. There is a tension between the need to generate *local* decisions in each subsystem quickly for timely response and the desire to coordinate the different subsystems for *global* optimization. Global coordination is especially challenging in heterogeneous computers with multi-vendor subsystems, as one needs to compose logic from different vendors that was designed oblivious of the full system configuration.

The current approach chosen by industry is to use centralized decision-making [3, 5, 15, 17, 19, 41, 67], despite the availability of per-subsystem sensors and actuators. The reason is the difficulty of composing independently designed controllers for system-wide efficiency [19, 57, 58]. There are many heuristic policies that are difficult for designers to develop even within a single subsystem like a CPU; it is even harder to redesign such logic to make it work across subsystems [19].

Researchers too have examined the joint optimization of multiple hardware subsystems. For example, they have optimized the combination of CPU and GPU [57, 58], GPU and memory [56], CPU and memory [23], multiple cores in a multicore [14, 27, 63, 82, 83], and servers in a datacenter [62]. In many of these works, however, the decision-making is centralized [14, 23, 27, 56–58, 82, 83]. In addition, many of these systems also rely on heuristics.



(a) Intel DPTF [22] **(b) IBM OCC [67]** **(c) AMD SMU [3, 17]**

Figure 1: State-of-the-art resource control architectures for heterogeneous computers from leading vendors.

To control emerging heterogeneous computers effectively, we need a new approach that is fast, globally coordinated, and modular. This paper presents such an approach, based on a new control framework called *Tangram*. Tangram is a decentralized framework for fast response time. However, decentralization does not come at the expense of global optimization. Further, Tangram is modular, so it can be used in different computer configurations.

Tangram introduces a new controller that combines an optimizing engine from Robust Control [73] with fast engines for safety and reconfiguration. This controller has a standard interface and is present in each subsystem of the computer. As a heterogeneous computer is built by assembling different subsystems, the controllers of the subsystems are also connected hierarchically, exchanging standard coordination signals. The resulting Tangram control framework is fast, globally coordinated, and modular. It is fast because each controller makes decisions on its own local subsystem immediately. It is globally coordinated because coordination signals propagate information and constraints between controllers. Finally, it is modular because each controller is built with knowledge of only its subsystem and has a standard interface.

We prototype Tangram in a multi-socket heterogeneous server that we built using components from three different vendors. The server has two quad-core CPU chips and a GPU chip. The controllers use a robust control theoretic design. They run as privileged software, accessing the System Management Units (SMUs) of the subsystems. Compared to state-of-the-art control, Tangram reduces, on average, the execution time of heterogeneous applications by 31% and their Energy-Delay Product (EDP) by 39%. The contributions of the paper are:

- Tangram, a fast, globally coordinated, and modular control framework to manage heterogeneous computers.
- A novel controller design that combines multiple engines and uses formal control principles.
- A prototype of Tangram in a server that we build using components from three different vendors, and its evaluation.

2 COMPUTER CONTROL TODAY

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

2.1 Organization

Most controllers from leading vendors are centralized (Figure 1). As shown in Figure 1a, Intel uses the Dynamic Power and Thermal Framework (DPTF) to manage the CPU and GPU in their multi-chip Core i7-8809G [19, 22]. The DPTF is a centralized kernel driver. Each chip exposes sensor data and allows DPTF to set its controllable inputs.

Figure 1b shows the On Chip Controller (OCC) in IBM POWER9. It is a centralized hardware controller that actuates each on-chip core and the GPUs attached to the chip [66, 67]. In a 2-socket system, one OCC becomes the master for global control, and the other a slave with restricted decision-making. The slave OCC is limited to sending sensor data and applying input values given by the master.

Figure 1c shows the hardware System Management Unit (SMU) design from AMD [17]. AMD's EPYC™ and Ryzen™ processors consist of one or more dies, each of which has one SMU. When multiple dies are used in a socket, one SMU becomes the master and the others are slaves, as with IBM. The slave SMUs handle only events like high temperatures or current, where quick response is necessary. The master SMU makes centralized decisions for the modules in all the dies in the system. In a two-socket system, there is a single master for both sockets.

When an on-die GPU is integrated with the CPU, AMD uses the Bidirectional Application Power Management (BAPM) algorithm, which is a centralized algorithm running on firmware to manage power between the CPUs and GPU [3]. With discrete GPUs, there is no communication channel between the CPU and GPU controllers. Therefore, system-level coordination needs to be handled through software drivers, as with the Intel design above.

Centralized hardware control is also the choice in research [14, 23, 27, 44, 56–60, 82, 83]. Unfortunately, centralized control is slow because data and decisions must cross chip boundaries, and experience contention at the single controller. It is also non-modular because integrating a new component or using a different configuration of the subsystems requires a full re-design of the controller.

As an alternative, Raghavendra *et al.* [62] describe a *Cascaded* design for power capping in datacenters. The proposal is shown in Figure 2. The datacenter is organized as a set of enclosures, each containing a set of server blades. A Group divider splits the total power budget among the enclosures. Then, the Enclosure divider in each enclosure splits its designated power level $P_{enclosure}$ among its blades. Then, each blade supervisor (Sup) receives its designated power level P_{blade}, and enforces it by providing a target for a PID controller. The PID controller changes the blade frequency to achieve the target. The enclosure divider and the blade supervisor always keep the power of the enclosure and blade at the respective $P_{enclosure}$ and P_{blade} values they receive.

While this design is scalable, it has the limitation that the dividers are not controllers that could optimize the system; they just divide the power budget. In addition, changing a blade's power budget

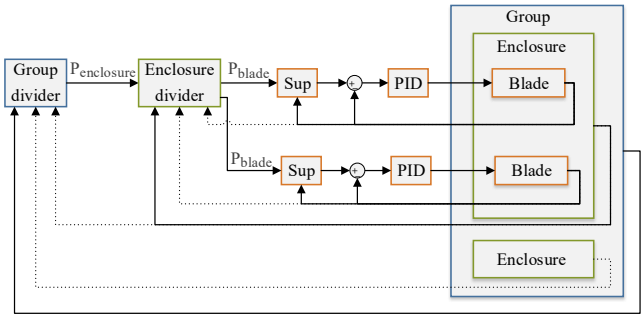


Figure 2: Cascaded control of a datacenter.

requires a long chain of decisions. When it is necessary to increase the blade's power, the group divider first increases $P_{enclosure}$, after which the enclosure divider can raise P_{blade}. Then, the supervisor changes the target and, finally, the PID controller can increase frequency. As the group divider's decision loop is much slower than the innermost PID controller [62], there is a long delay between the need for a power increase and the actual increase. This may cause suboptimal operation and instability [12].

2.2 Controller Objectives

There are *Safety* controllers that protect the system from dangerous conditions (e.g., high current or voltage droops), and *Enhancement* controllers that optimize the execution for goals like power, performance, or EDP [17]. Safety controllers usually provide continuous monitoring and an immediate response, while enhancement controllers periodically search through a multidimensional trade-off space for the best operating point. In current industrial designs, these two types of controllers typically operate in a decoupled manner [17].

Designs from research typically focus on enhancement, disregarding interaction with safety mechanisms. Some exceptions are works that consider temperature as a soft constraint (e.g., [34, 60, 62]), or those that probabilistically characterize safety mechanisms like circuit breaker tripping (e.g., [27]).

2.3 Formal Control vs Heuristics

Current industrial designs typically use heuristics for resource control [3, 37, 61, 67, 69, 74]. A few use heuristics plus PID controllers [16, 17, 19, 41]. In most cases, controllers monitor one single parameter (i.e., output), like power or skin temperature, and actuate a single parameter (i.e., input), like frequency. Hence, they are Single Input Single Output (SISO). Often, multiple controllers actuate the same input, such as the CPU frequency. In this case, the conflicting decisions are combined using heuristics. For example, IBM's OCC assigns each controller a vote and a majority algorithm sets the input [66].

Heuristics can result in unintended inefficiencies [28, 42, 44, 59, 78, 84]. Further, they make it difficult to decentralize resource control [19], which is necessary for fast response and modularity. Paul *et al.* [57, 58] show real examples of how multiple controllers using heuristics fail to coordinate in a system with a CPU and GPU. For instance, in a workload whose performance is limited by the GPU, CPU heuristics see low CPU memory traffic and boost CPU frequency. This does not improve performance and wastes power.

3 BACKGROUND: ROBUST CONTROL

Robust control focuses on controlling systems with only partial information [73]. Figure 3 shows a robust control loop. S is the system (e.g., a multicore) and K is the robust controller. The system has outputs y (e.g., power consumed) and configurable inputs u (e.g., frequency). The outputs must be kept close to the output targets r. The controller reads the deviations of the outputs from their targets ($\Delta y = r - y$), and sets the inputs. In this paper, we use controllers with Multiple Inputs Multiple Outputs (MIMO); they can set several inputs to meet several output targets simultaneously. This removes the need for several piecemeal optimization algorithms.

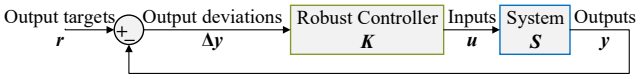


Figure 3: Robust control loop.

The controller is a state machine characterized by a state vector, $x(T)$, which is its accumulated experience that evolves over time T. The controller advances its state to $x(T+1)$ and generates the system inputs $u(T)$ by reading the output deviations $\Delta y(T)$ and using its state $x(T)$:

$$\begin{aligned} x(T+1) &= A \times x(T) + B \times \Delta y(T) \\ u(T) &= C \times x(T) + D \times \Delta y(T) \end{aligned} \qquad x(0) = 0 \tag{1}$$

where A, B, C, and D are matrices that encode the controller. Equation 1 is similar to how non-robust controllers like LQG [59] or MPC [44] operate. However, with robust control, the controller K has special properties that are relevant for this work.

First, designers can specify how big the output deviations can be for each output. Hence, important outputs can be set with tighter deviation bounds. Designers can also specify the relative overheads of changing each individual input. Then, the controller will minimize the changes to the inputs with higher overheads.

Second, the controller can be built with only a partial model of the true system. All unmodeled behavior is considered "uncertainty", and designers specify the worst case impact of such uncertainty, called *Uncertainty Guardband*. For example, a 50% uncertainty guardband means that the true system's outputs can be up to 50% different from what the model predicts. The controller guarantees to keep the output deviations within bounds even though it was built with this much inaccurate system information.

Robust controller design is automated [32] and tools aid designers in setting the controller parameters. Recently, robust control has been used in Yukta [60], to co-design controllers for different layers in the computing stack (e.g., hardware and OS).

4 TANGRAM: DECENTRALIZED CONTROL

Our goal is to design and prototype a control framework for heterogeneous computers that is decentralized, globally coordinated, and modular. Decentralization is needed for fast control. However, it should not come at the expense of global optimization. Further, the

framework should be modular to be usable in different computer configurations. These requirements rule out the conventional centralized and cascaded organizations. Moreover, for effectiveness, the controllers in this framework should combine safety and enhancement functionalities, be formal rather than heuristic-based, and be MIMO. We call our new framework *Tangram*.

In this section, we start by introducing the novel controller in Tangram, and then the Tangram modular control framework. Later, in Section 5, we build a Tangram prototype.

4.1 Controller Architecture

To the *Safety* and *Enhancement* types of controllers, we add a third one, which we call *Preconfigured*. Table 1 shows the controller differences and the control strategies they follow. Section 2.2 described safety and enhancement controllers. A preconfigured controller looks for a certain well-known operating condition. When the execution is under such a condition, the preconfigured controller uses a preset decision to bring the system to an optimal configuration. The priority of preconfigured controllers is lower than safety controllers but higher than enhancement controllers.

Table 1: Types of controllers.

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

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

4.1.1 Enhancement Engine. We use robust control [73] to build a MIMO enhancement controller. The controller monitors all local outputs to be optimized, like performance, power, and temperature, and sets the local inputs to keep all outputs close to the desired targets. It works with a *Planner*, which changes these targets to match changing conditions and to optimize metrics combining multiple outputs like EDP. The combination of controller and planner is the enhancement engine (Figure 4).

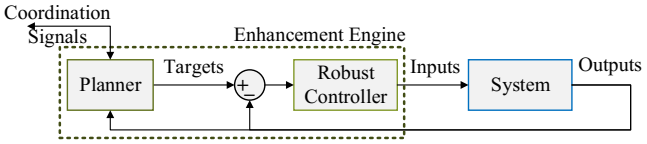


Figure 4: Enhancement engine.

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

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

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

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

Figure 5 shows the full architecture of our controller, with potentially multiple safety and preconfigured engines in parallel with the enhancement engine. All engines are connected to an arbiter. A mode detector chooses one engine by controlling the arbiter. Each engine can monitor potentially different outputs.

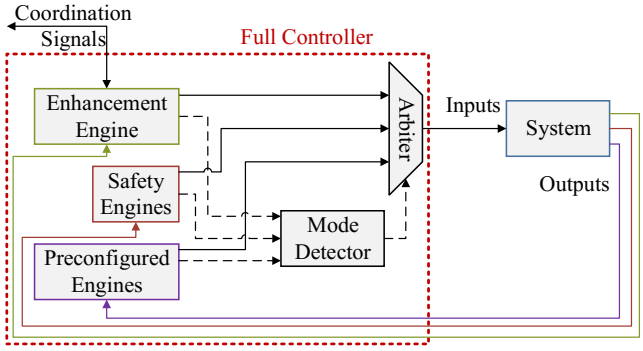


Figure 5: Our proposed controller.

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

At runtime, the enhancement engine optimizes the system. It may inadvertently trigger a safety engine, which then sets the inputs to the lowest values. The change induced by the safety engine is within the uncertainty guardband used in the controller's design. Once the hazardous condition is removed, the enhancement engine resumes operation but it remembers (using its state) to avoid further safety triggers. Thus, the enhancement engine can optimize inputs without repeated conflicts with the safety engines [25].

4.2 Subsystem Interface

Computers are organized as a hierarchy of subsystems, possibly built by different manufacturers. For example, a motherboard that

contains a GPU and a CPU chip is a subsystem, and a multicore chip that contains multiple cores is also a subsystem. To build decentralized, globally-coordinated modular control, we propose that each subsystem has a controller with a standard interface.

Figure 6 shows the interface. To understand it, we logically break a subsystem into its controller and the rest of the subsystem, which we call the *Component*. The figure shows one subsystem with its controller and its component. The controller generates or receives three sets of signals:

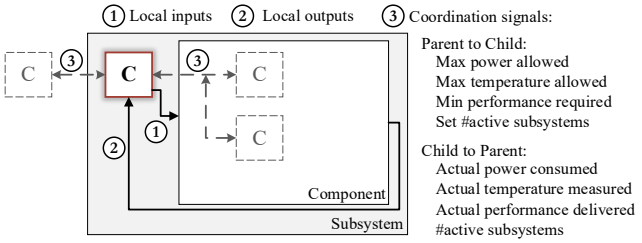


Figure 6: Proposed controller interface. In the figure, *C* means controller.

• **Coordination Signals** ③. These signals connect a controller with its parent controller and its potentially multiple child controllers. The figure shows a controller with two child controllers. The Coordination signals are shown on the right of the figure. From a controller to its child controllers, the signals set the operating constraints (i.e., the maximum power allowed, maximum temperature allowed, minimum performance required, and number of active subsystems). The child controllers use this information as constraints as they optimize their own components. From a controller to its parent controller, the signals report on the operating conditions (i.e., actual power consumed, actual temperature measured, actual performance delivered, and number of active subsystems). The parent controller uses this information to potentially assign new constraints to all of its children. The coordination signals use parameters readily available in current systems.
• **Local Inputs** ① and **Local Outputs** ②. These are the conventional signals that a controller uses to change and sense its component, respectively. They require no coordination and, therefore, can be manufacturer-specific. Examples of local inputs are frequency and cache size, and of local outputs are performance, power, temperature, and dI/dt. Figure 5 shows how the inputs are generated from the output measurements.

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

4.3 Tangram Control Framework

As a computer is built by assembling different subsystems hierarchically, the controllers of different subsystems are also connected hierarchically, exchanging the standard Coordination signals (Section 4.2). The result is the Tangram control framework. Figure 7 shows the framework – without the proprietary Local Input and Local Output signals – for a computer node that contains two modules, with one of the modules containing two chips.

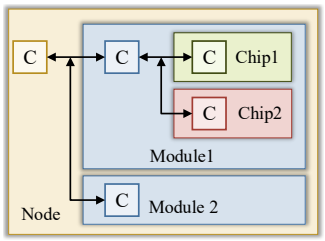


Figure 7: The decentralized, modular Tangram framework. *C* means controller.

The Tangram control framework is modular, fast, and globally coordinated. It is modular because each controller is built with knowledge of only its subsystem. For example, in Figure 7, the designers of Chip 1 and Chip 2 develop their controllers independently. Similarly, the Module 1 controller is developed without knowing about the Chip 1 and Chip 2 controllers. Further, changing a subsystem is easy – only the interfacing controllers need to be rewired and reprogrammed. For example, if we change Module 2, only the Node controller is affected.

The framework is fast because each controller makes decisions on its own subsystem immediately. This is unlike in cascaded designs where, to make a change that affects the local system requires a long chain of decisions (Section 2.1). It is also unlike centralized systems, where decisions are made in a faraway central controller.

Finally, the framework is globally coordinated because there are coordination signals that propagate information and constraints across the system. These signals are used differently than in cascaded systems. In Tangram, the local controller in a subsystem uses the constraints given by the coordination signals from the parent to identify the subsystem's best operating point; the local controller in a subsystem passes constraints to the child controllers. In the cascaded design discussed earlier (Section 2.1), instead, the divider simply provides, at each level, the exact parameters that fully determine the subsystem's operating point; the local controller at the leaf node tries to keep the outputs at this operating point.

4.4 Comparison to Contemporary Systems

The modular structure of Tangram may make the design appear obvious. Therefore, why do current systems not use a similar framework? A major reason is that their controllers do not use formal control. The use of a MIMO robust controller in each subsystem ensures that its optimizations work in the presence of other controllers in other subsystems. The controllers connected in a hierarchy can coordinate their actions. These benefits cannot be guaranteed by the current heuristic controllers used in individual subsystems, and simply using them together does not lead to cohesive decisions [19, 57].

4.5 Tangram Implementation

Tangram can be implemented in hardware or in software. For time-critical and hardware-specific measures such as DVFS, the controllers should be implemented in hardware or firmware, and signals should be carried by a special control network. Examples of such a network are AMD's SMU in EPYC™ systems [17] and

IBM's OCC in POWER9 [67] (Section 2.1). For less critical measures, controllers can be implemented in software, and communication between controllers can proceed using standard software channels.

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

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

4.6 Example of Tangram's Operation

Figure 8 is an example of how Tangram works. The figure considers a module composed of a CPU chip and a GPU chip. It shows the timeline of the actions of the three controllers, as they run in preconfigured, safety, or enhancement modes. The figure shows the coordination signals passed between the three controllers, which can be the local values measured (solid) or new constraints (dashed). For simplicity, we only consider power-related and activation/deactivation signals. As shown in the # Tasks timeline, the execution starts with zero tasks, then one CPU task appears, then one GPU task is added, and then many CPU tasks are added.

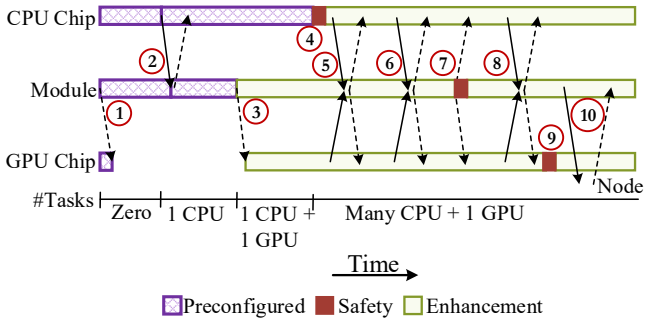


Figure 8: Example of coordination in Tangram. Solid arrows send measured local values, while dashed arrows provide new constraints.

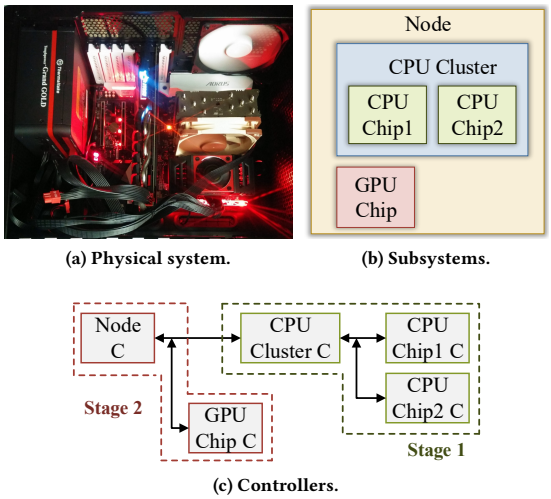
All controllers begin in preconfigured modes. At ①, the module controller turns the GPU chip off. When a CPU task arrives, the CPU controller changes to a new preconfigured mode. As the module realizes that there is one thread running (②), it changes the power assignment to the CPU chip, and changes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

4.7 Scalability of Tangram

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

5 TANGRAM PROTOTYPE

We prototype Tangram in a multi-socket heterogeneous server that we build using components from different vendors. We bought a computer motherboard from GIGABYTE [29], which we call the *Node*. The motherboard comes with an AMD Ryzen™ 7 1800X CPU cluster that has two quadcore processors with 2-way SMT [7]. To this, we add a GPU card from MSI [49] that contains an AMD Radeon™ RX 580 GPU [6]. Figure 9a shows a picture of the computer, and Figure 9b its organization. The system is a node with two subsystems: a CPU Cluster and a GPU Chip. The CPU Cluster has two quad-core CPU chips. The computer has subsystems from GIGABYTE, AMD, and MSI.



(a) Physical system. (b) Subsystems.

(c) Controllers.

Figure 9: Tangram organization.

Figure 9c shows the Tangram framework. To demonstrate Tangram's modularity, we build it in two stages, similar to how we assembled the computer. In stage one, we design and interconnect the controllers in the CPU Cluster subsystem. In stage two, we design the controllers for the GPU Chip and the Node subsystems, and interconnect them with the stage one controllers to build the full Tangram network.

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

The controllers read outputs and change inputs using Model Specific Registers (MSRs) [3] and internal SMU calls with proprietary libraries. Since AMD GPUs do not provide public access to dynamic performance counters, we read them using an internally developed library that intercepts OpenCL™ calls to identify the running kernel and its performance.

5.1 Stage One: CPU Cluster Subsystem

Figure 10 shows the Tangram network for the CPU cluster subsystem, with the different signals labeled. Table 2 shows the controllers' output and input signals that we use, based on the available sensors and actuators in our testbed. As we see, in a controller, the enhancement, safety, and preconfigured engines measure different outputs.

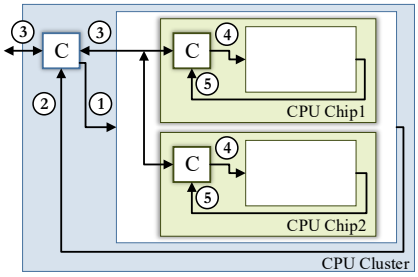


Figure 10: Tangram network in Stage One.

Table 2: Inputs and outputs of Stage One controllers.

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

Consider Table 2. A CPU chip controller monitors many outputs. The enhancement engine monitors the chip's performance (measured in billions of instructions committed per second or BIPS), and power. The safety engine monitors the Thermal Design Current (TDC) used to prevent voltage regulator overheating [21], and the hotspot temperature. The preconfigured engine monitors the number of running threads. The controller sets two inputs, namely, the chip's frequency (2.2 GHz – 3.6 GHz) and the number of active cores (0 – 4).

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

The coordination signals (③) measure the values and set the constraints discussed in Section 4.2.

5.2 Stage Two: Node Subsystem

Figure 11 shows the Tangram network for the whole node. It shows the details for the new controllers at this stage, namely, the controllers for the GPU Chip and Node. Table 3 lists the controllers' output and input signals, organized as in Table 2.

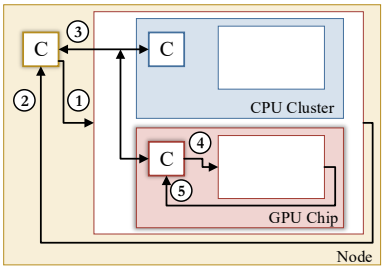


Figure 11: Tangram network in Stage Two.

Table 3: Inputs and outputs of Stage Two controllers.

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

As shown in Table 3, the GPU chip controller monitors the following outputs: the GPU performance and power (enhancement), the current and temperature (safety), and the number of kernels (preconfigured). The controller sets the frequency of the GPU compute units (300 – 1380 MHz) and of the graphics memory (300 – 2000 MHz). The node controller monitors similar outputs at its level (including whether the threads are CPU-type, GPU-type, or both), and sets the node frequency for the board's circuitry (300 – 2000 MHz).

The coordination signals (③) measure the values and set the constraints discussed in Section 4.2. The Node controller has no parent controller.

5.3 Structures in the Controllers

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

Enhancement Engine - Robust Controller: To design a robust controller, we need to (i) model its system (e.g., a CPU chip), and (ii) set the controller's input weights, uncertainty guardband, and output deviation bounds [73]. For the former, we use the System Identification modeling methodology [43]. In this approach, we run training applications on the system and, during execution, change the system inputs. We log the observed outputs and the inputs. From the data, we construct a dynamic polynomial model of the system:

$$y(T) = a_1 \times y(T-1) + \ldots + a_m \times y(T-m) + b_1 \times u(T) + \ldots + b_n \times u(T-n+1) \quad (2)$$

In this equation, y(T) and u(T) denote the outputs and inputs at time T. This model describes outputs at any time T as a function of the m past outputs, and the current and n-1 past inputs. The constants a_i and b_i are obtained by least squares minimization from the experimental data [43].

Appendix A describes how we set the controller parameters: input weights, uncertainty guardband, and output deviation bounds. With the model and these parameters, standard tools [32] generate the set of matrices that encode the robust controller (Section 3).

Enhancement Engine - Planner: The planner monitors local outputs and receives coordination signals from the parent and child controllers. With this information, it issues the best targets for all local outputs to the robust controller, and coordination signals to parent and child controllers. For example, the planner in a CPU chip's controller receives power, performance, temperature, and activation settings from the CPU cluster controller, and generates targets for its controller to optimize EDP. Our planners use the Nelder-Mead algorithm [48] described in Appendix A to search for the best targets under constraints received from the parent controller. We choose this search algorithm for its simplicity, effectiveness, and low resource requirements to run on firmware controllers [48].

Safety Engines: We consider two safety conditions: current and temperature. A hazard occurs if any exceeds the limits. Appendix A lists the hazardous values for current and temperature. If a hazard occurs in the CPU chip, the controller turns off all the cores except one, and sets the latter to the lowest frequency. If it occurs in the CPU cluster, the controller turns off one CPU chip and runs the other at the lowest frequency. If it occurs in the GPU chip or in the Node, the controllers set the frequencies to the lowest values.

Preconfigured Engines: We build the preconfigured engines of the different controllers to have the modes in Table 4.

5.4 Controller Overhead and Response Time

To show the nimbleness of our prototype, we list the overhead and response time of the controllers. Table 5 lists the overhead of the four structures that comprise the CPU Chip controller. For each structure, the table lists the dimension, storage required, number of instruction-like operations in the computation needed to produce an output, latency of computation, and power consumption. A robust controller's dimension is the number of elements in its state (i.e., the length of vector x in Equation 1 of Section 3). A planner's dimension is the number of possible modes in the Nelder-Mead search in Appendix A. From the table, we see that the storage, operation count, latency, and power values are very small – especially for the safety and preconfigured engines. The overheads of the controllers in the CPU cluster, GPU chip, and Node are similar.

Table 4: Modes of the different preconfigured engines.

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

Table 5: Overheads of Tangram's CPU chip controller.

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

We now consider the response time of the enhancement engines. Each enhancement engine has a robust controller and a planner (Figure 4). The response time of the robust controller includes reading outputs and targets, deciding on new inputs, and applying the new inputs. The response time of the planner includes reading outputs and coordination signals, deciding on targets, and communicating targets to the controller. Table 6 shows the measured response time of the robust controllers and planners in the enhancement engines of the different controllers in Tangram. For a parent controller/planner, the response time includes the time for its decisions to propagate through all the children and grandchildren until they affect the leaf robust controller's decision to change inputs. This may take multiple invocations of the leaf robust controller, which is activated every 50 ms. For comparison, we also show data for a centralized and a cascaded control framework that we implemented.

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

In *Centralized*, we place the single enhancement engine in the Node subsystem. Since the engine has to read many sensors, buffer

Table 6: Response time of the enhancement engines.

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

the data, and change many inputs across the system, it has a sizable response time (200 ms). Hence, it is not suited for fast response.

In *Cascaded*, we build the design in Figure 2. Only the leaf subsystems (CPU and GPU chips) have robust controllers, and their response time is similar to those in Tangram. Higher levels in the hierarchy only have dividers, which set the power levels. A leaf controller cannot steer the system to new outputs until all the dividers in the hierarchy, starting from the topmost one, have observed a change in regime and, sequentially, agreed to a change in the power assignment. Because each divider is activated at longer and longer intervals as we move up the hierarchy, a round trip from the leaf subsystem to the outermost divider in the Node and back to the leaf takes 665 ms. Therefore, *Cascaded* has a long response time.

Table 6 shows that our software implementation of Tangram is fast. In mainstream processors, control algorithms are typically implemented as firmware running on embedded micro-controllers [17, 19, 52, 67, 68], and operate at ms-level granularity. Therefore, we envision the controllers in Tangram to be deployed as vendor-supplied firmware running on micro-controllers in their respective subsystems. This requires little change to existing hardware. Further, the storage overhead and number of operations from Table 5 indeed show that Tangram can be easily run as firmware on a micro-controller. With a firmware implementation, we estimate that Tangram's response times in Table 6 reduce by about one order of magnitude, providing much better real-time control. A firmware implementation would also lower the response times of the other frameworks in Table 6, but is unlikely to change the relative difference between the frameworks.

6 EVALUATING THE PROTOTYPE

6.1 Applications

We use the Chai applications [31], which exercise both the CPUs and the GPU *simultaneously*, unlike most benchmarks. They cover many collaboration patterns and utilize new features in heterogeneous processors like system-wide atomics, inter-worker synchronization, and load balancing of data parallel tasks. We use two applications for training (*pad* and *sc*) and five for evaluation (*bfs, hsti, rscd, rsct*, and *sssp*). For Stage One controllers, we run NAS 3.3 [24] and PARSEC 2.1 [13]. From NAS, we use two applications to train (*bt* with dataset D and *mg* with dataset C) and nine for evaluation (*dc* with dataset B, *cg, ft, lu, sp* and *ua* with dataset C, and *ep, is* and *mg* with dataset D). From PARSEC, we use two applications to train (*raytrace* with dataset native and *swaptions* with dataset simlarge) and eight for evaluation (*blackscholes, bodytrack, facesim, ferret, swaptions, fluidanimate, vips* and *x264*, all with dataset native).

6.2 Designs for Comparison

Our evaluation is comprised of three sets of comparisons, each evaluated using the appropriate subsystem of the prototype that can give us the most insights. The systems compared are: different enhancement engine designs on a CPU chip, different control architectures on a CPU cluster, and different complete frameworks on our full prototype. In all cases, our goal is to minimize the EDP of the system under constraints of maximum power, temperature, and current in each subsystem.

Comparing Enhancement Engine Designs. We compare our enhancement engine (which we call *Robust*) to alternative designs, such as *LQG* and *Heuristic* (Table 7) on a CPU chip. *LQG* is the Linear Quadratic Gaussian approach proposed by Pothukuchi *et al.* [59]. *Heuristic* is a collection of heuristics that use a gradient-free search to find the inputs that optimize the EDP metric. Instead of using a controller or a planner, it approximates the gradient using the past 2 output measurements and navigates the search space. The search uses random-restart after convergence, to avoid being trapped in local optima. This design is based on industrial implementations [2, 3, 17].

Table 7: Comparing enhancement engine designs.

Strategy	Description
LQG	LQG controller from [59].
Heuristic	Industrial-grade gradient-free optimization heuristics [3].
Robust	Our enhancement engine of Figure 4

Comparing Control Architectures. We take our proposed controller from Figure 5 and use it in Tangram, Centralized, and Cascaded architectures on a CPU cluster. *Centralized* uses a single instance of our Figure 5 controller in the CPU cluster. *Cascaded* follows the design by Raghavendra *et al.* [62]. It uses a controller in each leaf subsystem, and a divider in the CPU cluster. For *Centralized* and *Cascaded*, each subsystem has its own safety engine for fast response time, as in existing systems [17].

Comparing Complete Frameworks. Finally, we compare complete framework designs on our full computer. The framework designs are built with combinations of the above control architectures and enhancement engine designs, as listed in Table 8. Specifically, *Tangram Robust* is our proposed framework with our robust controller. *Cascaded LQG* is a state-of-the-art design combining prior work [59, 62]. *Tangram LQG* uses our control framework with LQG-based controllers. *Tangram Heuristic* uses our control framework with controllers based on industry-class heuristics. For this complete framework evaluation, we use the Chai programs.

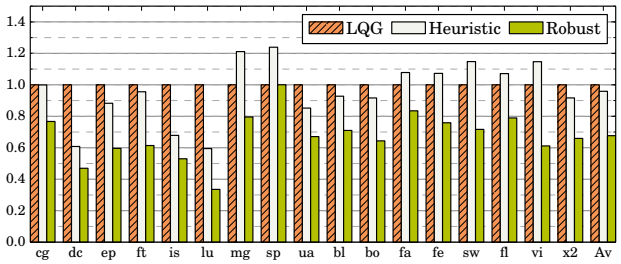
Table 8: Comparing complete control frameworks.

Control System	Description
Cascaded LQG	Architecture based on [62] with LQG controllers from [59].
Tangram LQG	Tangram architecture using LQG-based controllers.
Tangram Heuristic	Tangram architecture using industry-standard heuristics.
Tangram Robust	Tangram architecture with our proposed controllers.

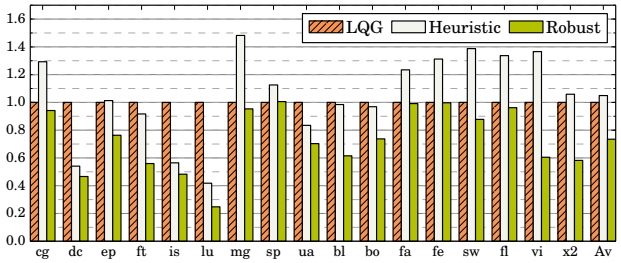
7 RESULTS

7.1 Comparing Enhancement Engine Designs

We compare the enhancement engine designs in Table 7 on a single CPU chip running NAS and PARSEC applications. Figures 12a and 12b show the execution time and EDP, respectively, with *LQG*, *Heuristic*, and *Robust* enhancement engines, normalized to *LQG*.



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



(b) Normalized EDP (lower is better).

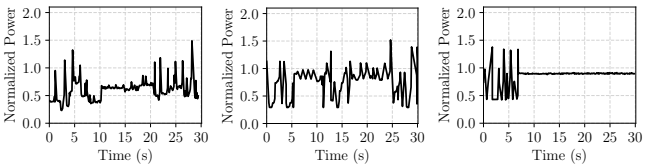
Figure 12: Comparing enhancement engine designs.

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

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

Heuristic also operates inefficiently, with oscillations between safety and enhancement engines as with *LQG*. It cannot effectively identify a configuration that is optimal and safe with heuristics alone. It has the highest average EDP. Finally, *Robust* has the fastest execution because the robust controller learns to optimally track output targets without safety hazards. Since it converges faster than *LQG* and keeps the output deviations within guaranteed bounds, the planner's search is effective and completes fast. Overall, Figure 12 shows the superiority of the *Robust* engine. On average, it reduces the execution time by 33%, and the EDP by 27% over *LQG*.

For more insight, Figure 13 shows a partial timeline of the power consumed by a CPU chip when running *ep*, an embarrassingly parallel NAS application that has a uniform behavior. The power is shown normalized to the maximum power that the CPU chip can consume in steady state. For this application, *LQG* and *Heuristic* have many oscillations due to switching between the safety and enhancement engines. However, the power in *Robust* converges rapidly and stays constant, thanks to the better control of its enhancement engine.

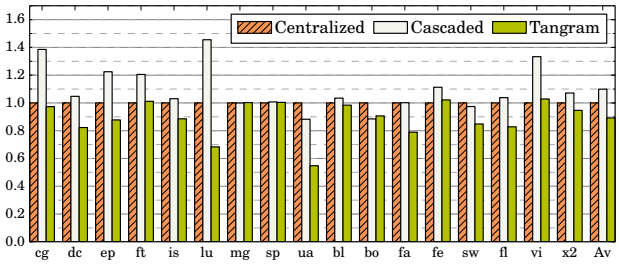


(a) LQG **(b) Heuristic** **(c) Robust**

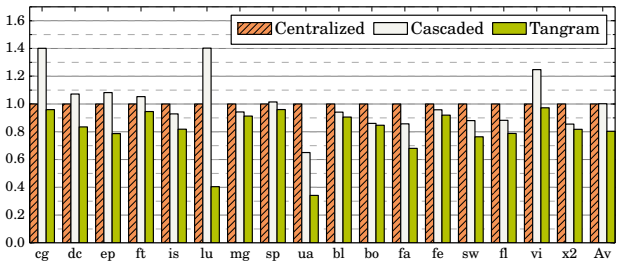
Figure 13: Partial timeline of the power consumed by *ep*.

7.2 Comparing Control Architectures

We compare Tangram to the *Centralized* and *Cascaded* architectures running applications in the CPU cluster. They all use our proposed controller from Figure 5 with a robust enhancement engine. Tangram uses a controller in each subsystem, *Centralized* uses a single controller in the CPU cluster, and *Cascaded* uses a controller in each leaf subsystem and a divider in the CPU cluster. Figures 14a and 14b show the execution time and EDP of the architectures, respectively, normalized to *Centralized*.



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

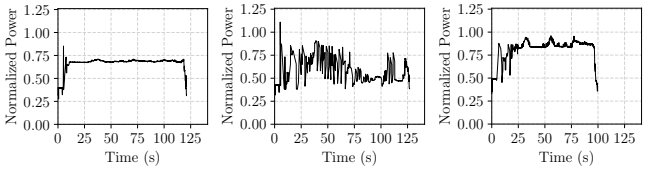


(b) Normalized EDP (lower is better).

Figure 14: Comparing control architectures.

These architectures differ mainly in the response times of their robust controllers and planners/dividers. In *Centralized*, there is a single planner and robust controller that have long response times. As a result, they sometimes miss opportunities to adjust power and performance, even though they have a global view. This results in inefficient execution. In *Cascaded*, the CPU chip controllers respond fast, but their interaction with the next-level divider is slow. As a result, the targets used by the controller lag behind the system state, limiting efficiency. Moreover, a divider is not as effective as a controller in setting the targets. Finally, *Tangram* has the lowest response times at all levels. Therefore, on average, it reduces the execution time by 11% and the EDP by 20% over *Centralized*.

To provide more insight, Figure 15 shows the power consumed by the entire CPU cluster as a function of time in *dc*, another NAS application. The power is shown normalized to the maximum power that the CPU cluster can consume in steady state. We see that *Tangram* uses higher power than *Centralized* and *Cascaded*. This is because it is responsive to application demands with its fast response time. While controllers communicate, they independently optimize their components for changing conditions by generating output targets and inputs fully locally. Therefore, the application finishes the earliest and even consumes the least energy.



(a) Centralized (b) Cascaded (c) Tangram

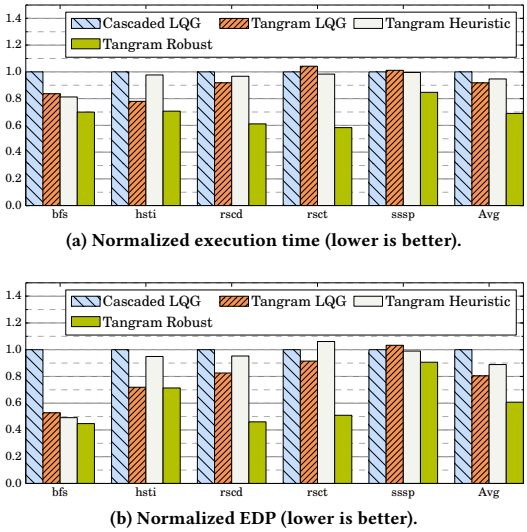
Figure 15: CPU cluster power in *dc* as a function of time.

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

7.3 Comparing Complete Frameworks

Finally, we compare our proposed *Tangram Robust* framework to state-of-the-art designs (Table 8) on our full heterogeneous prototype. Figures 16a and 16b show the execution time and EDP, respectively, running the heterogeneous Chai applications. The bars are normalized to those of *Cascaded LQG*, which we consider the state-of-the-art.

If we compare *Cascaded LQG* to *Tangram LQG*, we see that the latter has a lower execution time and EDP. This is because *Cascaded* is a slow response-time architecture in large systems (Table 6). Moreover, it lacks the hierarchy of MIMO controllers that optimize each level of the control hierarchy. In particular, the *bfs* application suffers from this limitation.



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

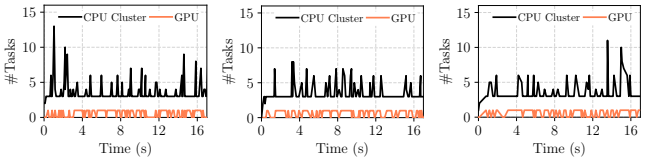
(b) Normalized EDP (lower is better).

Figure 16: Comparing complete control frameworks.

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

Finally, we see that our proposed framework (*Tangram Robust*) has the lowest execution time and EDP. This efficiency is due to two factors: its fast response time (Table 6), and the safety and optimality guarantees from using robust controllers at each level of the hierarchy. We see large gains even for programs like *rsct* that finely divide compute between the CPUs and the GPU. Overall, *Tangram Robust* reduces, on average, the execution time by 31% and the EDP by 39% over the state of the art *Cascaded LQG*. This makes *Tangram Robust* a significant advance.

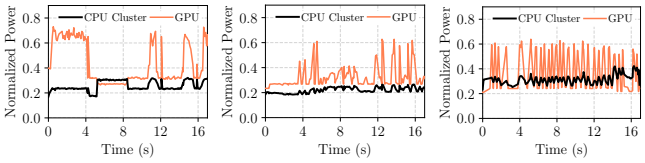
For more insight, consider *rsct*, which has rapidly-changing GPU kernels and CPU threads. Figure 17 shows a partial timeline of the number of active CPU threads and GPU kernels. Figure 18 shows a partial timeline of the power consumed by the CPU Cluster and the GPU. The power is shown normalized to the maximum power that the node can consume in steady state.



(a) Cascaded LQG (b) Tangram LQG (c) Tangram Robust

Figure 17: Partial timeline of the number of active threads in the CPU cluster and number of kernels in the GPU in *rsct*.

The frequent peaks and valleys in the three charts of Figure 17 show that this application is very dynamic. The number of active threads in the CPU cluster and the number of kernels in the GPU



(a) Cascaded LQG (b) Tangram LQG (c) Tangram Robust

Figure 18: Partial timeline of the power consumed by the CPU cluster and the GPU in *rsct*.

changes continuously. Hence, all the frameworks try to continuously change the power assigned to the CPU cluster and the GPU, based on their activity. In many cases, they trigger the preconfigured engines, such as when only a few CPU threads are active or no GPU kernel is running.

However, as shown in Figure 18, the different frameworks manage power differently. In Figure 18a, we see that *Cascaded LQG* is slow to shift power between the CPU cluster and the GPU, and vice-versa. This framework reacts slowly for two reasons. First, the cascaded architecture intrinsically has a long response time (Table 6). Second, the LQG controller takes long to converge. As a result, many tasks in the CPU or GPU start and complete before the power to that subsystem is changed.

In Figure 18b, we see that *Tangram LQG* is more responsive. However, the slow LQG engine in *Tangram LQG* can hardly match the fast-changing execution. In contrast, *Tangram Robust* in Figure 18c quickly reassigns power to the subsystem which best improves the overall EDP. This capability explains the programs' lower execution time and EDP in this framework.

8 RELATED WORK

General Control

There is a large body of work on controlling homogeneous or single-ISA heterogeneous processors [1, 2, 23, 27, 38, 44, 46, 51, 56, 59, 60, 62, 65, 70, 75, 78, 85]. Only a few consider heterogeneous processors with CPUs and GPUs [11, 57, 58, 69, 80]. Still, they use non-modular controllers that do not match the modular heterogeneous environments we target.

Resource control in production computers is predominantly heuristic [2–4, 16, 17, 37, 53, 57, 58, 61, 68, 69, 72, 74, 76, 78, 81]. As we indicated, heuristic control has limitations.

Research works propose many optimizing controllers [27, 34, 36, 38, 44, 59, 60, 62, 82–84]. Most do not consider interference from dedicated safety controllers. Therefore, they do not simultaneously guarantee optimality and safety. Some works do consider temperature as a soft constraint [34, 60, 62] while some probabilistically characterize mechanisms like circuit breaker tripping for their search [27]. In real designs, there are many safety engines that interrupt and override optimizing engines unpredictably. We guarantee optimality and safety simultaneously.

Enhancement Approaches

Heuristic Control. Many designs rely heavily on heuristics for resource control [23, 36, 37, 56–58, 68, 69, 72, 78]. While easy to implement for simple systems, designing, tuning and verifying heuristics becomes dramatically expensive as systems and resource management goals become complex. This can result in unintended inefficiencies [28, 42, 44, 59, 78, 84].

Formal Control. PID controllers are popularly used in many works due to their simplicity [9, 17, 18, 28, 39, 50, 51, 62, 64, 68, 70, 71]. However, PID controllers are Single Input Single Output (SISO) designs, inadequate to meet the multiple objectives in computers [44, 59, 60]. LQG [59, 63] and MPC [44] controllers can handle Multiple Input Multiple Output (MIMO) systems, but are relatively less effective in uncertain and multi-controller environments [60, 73]. Yukta [60] proposes the use of robust controllers for computer systems. These controllers operate well in environments that are not fully modeled. Yukta introduces the use of a robust controller for each system layer (e.g., the hardware and OS layers). In this paper, we focus only on a single layer, and propose a framework with a controller in each subsystem, forming a hierarchy of controllers connected with coordination signals.

Other Systematic Methods. Some works formulate Energy×Delay^n minimization as a convex optimization problem solved with linear programming solvers [34, 35, 65]. Solver-based approaches require more time to generate a decision than robust controllers. Some use market-theory [82, 83] or game-theory [27] to manage resources in specific contexts. Finally, some researchers use machine learning (ML) techniques for resource management [14, 20, 26, 33]. Mishra *et al.* [51] use ML to tune a PID controller and a solver that manage a big.LITTLE processor.

Control System Architectures

Centralized. Most works use centralized frameworks (e.g., [34, 35, 44, 57–59, 84]). Some use two-step proxy designs where a proxy module in each component requests resources and a centralized manager performs the allocation [14, 27, 36, 38, 82, 83]. Centralized designs are not modular, do not scale to multi-chip computers, and do not fit IP-based system designs. As systems grow large, the controller's response time degrades quickly because it runs a bulky algorithm and becomes a point of contention.

Cascaded. Raghavendra *et al.* [62] propose a multilevel cascaded system to manage power in a datacenter. Rahmani *et al.* [63] propose a similar 2-level design for a big.LITTLE processor. Here, each component has two LQG controllers. A supervisor chooses one of them to control the system and provides targets for all local outputs. We showed that cascaded is non-modular and has a poor response time.

Other Architectures. Muthukaruppan *et al.* [54] use a combination of cascaded and decoupled PID controllers for a big.LITTLE processor. Some designs order decoupled controllers by priority for limited coordination [28, 71].

9 CONCLUSION

To control heterogeneous computers effectively, this paper introduced Tangram, a new control framework that is fast, globally coordinated, and modular. Tangram introduces a new formal controller that combines multiple engines for optimization and safety, and has a standard interface. Building the controller for a subsystem requires knowing only about that subsystem. As a heterogeneous computer

is assembled, the controllers in the different subsystems are connected, exchanging standard coordination signals. To demonstrate Tangram, we prototyped it in a multi-socket heterogeneous server that we assembled using subsystems from multiple vendors. Compared to state-of-the-art control, Tangram reduced, on average, the execution time of heterogeneous applications by 31% and their energy-delay product by 39%.

ACKNOWLEDGMENTS

This work has been supported in part by NSF under grants CNS 17-63658, CCF 17-25734, and CCF 16-29431.

A PROTOTYPE DESIGN DETAILS

Modeling the Subsystems

To model computer systems, we use black-box system identification from experimental data [43]. For the CPU chip and CPU cluster, we run two training applications from NAS and two from PARSEC. The obtained models have an order of 2 for the CPU chip and CPU cluster (i.e., $m = n = 2$ in Equation 2). For the GPU chip and node, we use two training applications from Chai [31]. The model orders for the GPU chip and node are 2 and 4, respectively.

Designing Robust Controllers

To design the robust controllers with automated tools, we need to specify the input weights, uncertainty guardbands and output deviation bounds [32]. We specify the weight for each input in a subsystem based on the relative overhead of changing that input. For the CPU chip, turning a core on/off takes at least twice as long as changing the frequency. Hence, we use weights of 1 and 2 for the CPU frequency and number of active cores, respectively. For the CPU cluster, we also use weights of 1 and 2 for the CPU cluster frequency and number of active CPU chips. The GPU chip inputs are compute frequency and memory frequency. They have similar overheads and hence, we use weights of 1 for both. The node has a single input, namely the node frequency, and we set its weight to 1.

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

With the system model, weights, and uncertainty guardband, MATLAB [32] gives the smallest output deviation bounds the controller can provide. We use the priority of outputs along with these suggestions to set the final output deviation bounds. In each controller, we rank performance bounds as less critical than power bounds. With these specifications, MATLAB generates the set of matrices that encode the robust controller (Equation 1). The output deviation bounds guaranteed by the robust controller in the CPU chip, CPU cluster, and GPU chip are [±15%, ±10%] for performance and power, respectively. For the node, the bounds are [±25%, ±20%] for performance and power, respectively.

Designing Planners

Our planners use Nelder-Mead search [48] to generate the local output targets and the coordination signals to the child controllers. As shown in the outline below, the algorithm moves through the following five modes.

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

2) Rank: Based on EDP, identify three points out of these $N + 1$ and rank them as Best, Worst, and Lousy (i.e., the point better only than Worst). Compute the centroid of all the $N + 1$ points except Worst. The Best, Worst, and Lousy points, plus the centroid are shown in Figure 19.

3) Reflect: Find a new point by reflecting the Worst point about the Centroid. This is shown as Point 1 in Figure 19. If the EDP at this point is better than Worst, Point 1 replaces Worst and the search returns to Step 2. Otherwise, the search moves to Step 4.

4) Contract: The search finds a new point which is the midpoint between Centroid and Worst. This is Point 2 in Figure 19. If this point is better than Worst, it replaces Worst and the search returns to Step 2. Otherwise, the search moves to Step 5.

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

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

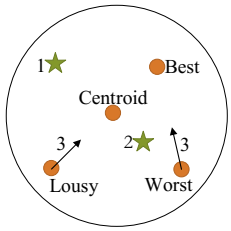


Figure 19: Nelder-Mead search used by the planner.

Safety Engine Limits

The maximum values that we use for current (TDC) and thermal safety of each subsystem are shown in Table 9.

Table 9: Limits used in safety engines in all controllers.

	CPU Chip	CPU Cluster	GPU Chip	Node
Current (A)	15	25	30	50
Temperature (° C)	55	65	60	70

AMD, the AMD Arrow logo, EPYC, Radeon, Ryzen, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc. Other product names used herein are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Almutaz Adileh, Stijn Eyerman, Aamer Jaleel, and Lieven Eeckhout. 2016. Maximizing Heterogeneous Processor Performance Under Power Constraints. *ACM Trans. Archit. Code Optim.* 13, 3 (Sept. 2016), 29:1–29:23.

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

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

[4] Advanced Micro Devices, Inc. 2018. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 70h-7Fh Processors. http://developer.amd.com/resources/developer-guides-manuals/. Advanced Micro Devices, Inc.

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

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

[7] Advanced Micro Devices, Inc. 2019. AMD Ryzen. http://www.amd.com/en/ryzen. Accessed: 2019.

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

[9] Nawaf Almoosa, William Song, Yorai Wardi, and Sudhakar Yalamanchili. 2012. A Power Capping Controller for Multicore Processors. In *American Control Conference*.

[10] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *International Symposium on Computer Architecture*.

[11] Trinayan Baruah, Yifan Sun, Shi Dong, David Kaeli, and Norm Rubin. 2018. Airavat: Improving Energy Efficiency of Heterogeneous Applications. In *Conference on Design, Automation and Test in Europe*.

[12] Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. 2012. The Need for Speed and Stability in Data Center Power Capping. In *International Green Computing Conference*.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*.

[14] 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*.

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

[16] Martha Broyles, Christopher J. Cain, Todd Rosedahl, and Guillermo J. Silva. 2015. *IBM EnergyScale for POWER8 Processor-Based Systems*. Technical Report. IBM.

[17] Thomas Burd, Noah Beck, Sean White, Milam Paraschou, Nathan Kalyanasundharam, Gregg Donley, Alan Smith, Larry Hewitt, and Samuel Naffziger. 2019. "Zeppelin": An SoC for Multichip Architectures. *IEEE J. Solid-State Circuits* 54, 1 (Jan. 2019), 133–143. https://doi.org/10.1109/JSSC.2018.2873584

[18] Xinwei Chen, Yorai Wardi, and Sudhakar Yalamanchili. 2017. Power Regulation in High Performance Multicore Processors. In *IEEE Conference on Decision and Control*.

[19] Srinivas Chennupaty. 2018. Thin & Light & High Performance Graphics. https://www.hotchips.org/hc30/1conf/1.04_Intel_Thin_Light_Gaming_HotChips_SC_Final.pdf. In *Hot Chips: A Symposium on High Performance Chips*. Intel Corporation.

[20] 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*.

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

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

[23] 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*.

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

[25] John C. Doyle, Joseph E. Wall, and Gunter Stein. 1982. Performance and Robustness Analysis for Structured Uncertainty. In *IEEE Conference on Decision and Control*.

[26] Christophe Dubach, Timothy M. Jones, and Edwin V. Bonilla. 2013. Dynamic Microarchitectural Adaptation Using Machine Learning. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013), 31:1–31:28.

[27] Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. 2016. The Computational Sprinting Game. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[28] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated Multi-objective Control for Self-adaptive Software Design. In *Joint Meeting on Foundations of Software Engineering*.

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

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

[31] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*.

[32] Da-Wei Gu, Petko H. Petkov, and Mihail M. Konstantinov. 2013. *Robust Control Design with MATLAB* (2nd ed.). Springer.

[33] Can Hankendi, Ayse Kivilcim Coskun, and Henry Hoffmann. 2016. Adapt&Cap: Coordinating System- and Application-Level Adaptation for Power-Constrained Systems. *IEEE Des. Test* 33, 1 (2016), 68–76.

[34] Vinay Hanumaiah, Digant Desai, Benjamin Gaudette, Carole-Jean Wu, and Sarma Vrudhula. 2014. STEAM: A Smart Temperature and Energy Aware Multicore Controller. *ACM Trans. Embed. Comput. Syst.* 13, 5s (Oct. 2014), 151:1–151:25.

[35] Jin Heo, Dan Henriksson, Xue Liu, and Tarek Abdelzaher. 2007. Integrating Adaptive Components: An Emerging Challenge in Performance-Adaptive Systems and a Server Farm Case-Study. In *International Real-Time Systems Symposium*.

[36] Canturk Isci, Alper Buyuktosunoglu, Chen-yong Chen, 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*.

[37] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. 2012. Power Management of the Third Generation Intel Core Micro Architecture formerly Codenamed Ivy Bridge. In *Hot Chips: A Symposium on High Performance Chips*.

[38] Sudhanshu Shekhar Jha, Wim Heirman, Ayose Falcón, Trevor E. Carlson, Kenzo Van Craeynest, Jordi Tubella, Antonio González, and Lieven Eeckhout. 2015. Chrysso: An Integrated Power Manager for Constrained Many-core Processors. In *ACM International Conference on Computing Frontiers*.

[39] Philo Juang, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi, and Douglas W. Clark. 2005. Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors. In *International Symposium on Low Power Electronics and Design*.

[40] Youngtaek Kim, Lizy Kurian John, Sanjay Pant, Srilatha Manne, Michael Schulte, W. Lloyd Bircher, and Madhu S. Sibi Govindan. 2012. AUDIT: Stress Testing the Automatic Way. In *International Symposium on Microarchitecture*. IEEE Computer Society, 212–223.

[41] Charles Lefurgy. 2013. Avoiding Core Meltdown! - Adaptive Techniques for Power and Thermal Management of Multi-Core Processors. https://researcher.watson.ibm.com/researcher/files/us-lefurgy/DAC2013_Lefurgy_v6.pdf. Tutorial, Design Automation Conference.

[42] Xiaodong Li, Zhenmin Li, Francis David, Pin Zhou, Yuanyuan Zhou, Sarita Adve, and Sanjeev Kumar. 2004. Performance Directed Energy Management for Main Memory and Disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[43] Lennart Ljung. 1999. *System Identification : Theory for the User* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[44] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. 2017. Automated Control of Multiple Software Goals Using Multiple Actuators. In *Joint Meeting on Foundations of Software Engineering*.

[45] Ravi Mahajan, Robert Sankman, Neha Patel, Dae-Woo Kim, Kemal Aygun, Zhiguo Qian, Yidnekachew Mekonnen, Islam Salama, Sujit Sharan, Deepti Iyengar, and Debendra Mallik. 2016. Embedded Multi-die Interconnect Bridge (EMIB) – A High Density, High Bandwidth Packaging Interconnect. In *IEEE Electronic Components and Technology Conference*.

[46] Abhinandan Majumdar, Leonardo Piga, Indrani Paul, Joseph L. Greathouse, Wei Huang, and David H. Albonesi. 2017. Dynamic GPGPU Power Management Using Adaptive Model Predictive Control. In *International Symposium on High Performance Computer Architecture*.

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

[48] John H. Mathews and Kurtis D. Fink. 2006. *Numerical Methods Using MATLAB*. Pearson Education, Limited. https://books.google.com/books?id=-DpDPgAACAAJ

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

[50] Asit K. Mishra, Shekhard Srikantaiah, Mahmut Kandemir, and Chita R. Das. 2010. CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[51] 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*.

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

[53] Benjamin Munger, David Akeson, Srikanth Arekapudi, Tom Burd, Harry R. Fair, Jim Farrell, Dave Johnson, Guhan Krishnan, Hugh McIntyre, Edward McLellan, Samuel Naffziger, Russell Schreiber, Sriram Sundaram, Jonathan White, and Kathryn Wilcox. 2016. Carrizo: A High Performance, Energy Efficient 28 nm APU. *IEEE J. Solid-State Circuits* 51, 1 (2016), 105–116.

[54] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. 2013. Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In *Design Automation Conference*.

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

[56] Indrani Paul, Wei Huang, Manish Arora, and Sudhakar Yalamanchili. 2015. Harmonia: Balancing Compute and Memory Power in High-performance GPUs. In *International Symposium on Computer Architecture*.

[57] Indrani Paul, Srilatha Manne, Manish Arora, W. Lloyd Bircher, and Sudhakar Yalamanchili. 2013. Cooperative Boosting: Needy Versus Greedy Power Management. In *International Symposium on Computer Architecture*.

[58] Indrani Paul, Vignesh Ravi, Srilatha Manne, Manish Arora, and Sudhakar Yalamanchili. 2013. Coordinated Energy Management in Heterogeneous Processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis*.

[59] 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*.

[60] 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*.

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

[62] 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 Conference on Architectural Support for Programming Languages and Operating Systems*.

[63] 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*.

[64] Karthik Rao, William Song, Sudhakar Yalamanchili, and Yorai Wardi. 2015. Temperature Regulation in Multicore Processors using Adjustable-gain Integral Controllers. In *IEEE Conference on Control Applications*.

[65] Karthik Rao, Jun Wang, Sudhakar Yalamanchili, Yorai Wardi, and Handong Ye. 2017. Application-Specific Performance-Aware Energy Optimization on Android Mobile Devices. In *International Symposium on High Performance Computer Architecture*.

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

[67] Todd Rosedahl, Martha Broyles, Charles Lefurgy, Bjorn Christensen, and Wu Feng. 2017. Power/Performance Controlling Techniques in OpenPOWER. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.). Springer International Publishing, 275–289.

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

[69] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32 (March 2012).

[70] Muhammad Husni Santriaji and Henry Hoffmann. 2016. GRAPE: Minimizing Energy for GPU Applications with Performance Requirements. In *International Symposium on Microarchitecture*.

[71] Stepan Shevtsov and Danny Weyns. 2016. Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 229–241.

[72] Balaram Sinharoy, Randy Swanberg, Naresh Nayar, Bruce G. Mealey, Jeff Stuecheli, Berni Schiefer, Jens Leenstra, Joefon Jann, Philipp Oehler, David Levitan, Susan Eisen, Dean Sanner, Thomas Pflueger, Cedric Lichtenau, William E. Hall, and Tim Block. 2015. Advanced Features in IBM POWER8 systems. *IBM Jour. Res. Dev.* 59, 1 (Jan. 2015), 1:1–1:18.

[73] Sigurd Skogestad and Ian Postlethwaite. 2005. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons.

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

[75] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. 2014. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. In *International Symposium on Microarchitecture*.

[76] Sriram Sundaram, Sriram Samabmurthy, Michael Austin, Aaron Grenat, Michael Golden, Stephen Kosonocky, and Samuel Naffziger. 2016. Adaptive Voltage Frequency Scaling Using Critical Path Accumulator Implemented in 28nm CPU. In *International Conference on VLSI Design*.

[77] Sehat Sutardja. 2015. The Future of IC Design Innovation. In *International Solid-State Circuits Conference*.

[78] 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 Microarchitecture*.

[79] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. 2017. Design and Analysis of an APU for Exascale Computing. In *International Symposium on High Performance Computer Architecture*.

[80] Hao Wang, Vijay Sathish, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. 2012. Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors. In *International Conference on Parallel Architectures and Compilation Techniques*.

[81] Xin Wang. 2017. Intelligent Power Allocation: Maximize performance in the thermal envelope. ARM White Paper.

[82] Xiaodong Wang and José F. Martínez. 2015. XChange: A Market-based Approach to Scalable Dynamic Multi-resource Allocation in Multicore Architectures. In *International Symposium on High Performance Computer Architecture*.

[83] Xiaodong Wang and José F. Martínez. 2016. ReBudget: Trading Off Efficiency vs. Fairness in Market-Based Multicore Resource Allocation via Runtime Budget Reassignment. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[84] Yefu Wang, Kai Ma, and Xiaorui Wang. 2009. Temperature-constrained Power Control for Chip Multiprocessors with Online Model Estimation. In *International Symposium on Computer Architecture*.

[85] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. 2004. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[86] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natahlie Enright Jerger, and Gbriel H. Loh. 2018. Modular Routing Design for Chiplet-Based Systems. In *International Symposium on Computer Architecture*.