2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

CASH: Supporting IaaS Customers with a Sub-core Configurable Architecture

Yanqi Zhou
Electrical Engineering Department
Princeton University
Princeton, USA
yanqiz@princeton.edu

Henry Hoffmann
Computer Science Department
University of Chicago
Chicago, USA
hankhoffmann@cs.uchicago.edu

David Wentzlaff
Electrical Engineering Department
Princeton University
Princeton, USA
wentzlaf@princeton.edu

***Abstract*—Infrastructure as a Service (IaaS) Clouds have grown increasingly important. Recent architecture designs support IaaS providers through fine-grain configurability, allowing providers to orchestrate low-level resource usage. Little work, however, has been devoted to supporting IaaS customers who must determine how to use such fine-grain configurable resources to meet quality-of-service (QoS) requirements while minimizing cost. This is a difficult problem because the multiplicity of configurations creates a non-convex optimization space. In addition, this optimization space may change as customer applications enter and exit distinct processing phases. In this paper, we overcome these issues by proposing CASH: a fine-grain configurable architecture co-designed with a cost-optimizing runtime system. The hardware architecture enables configurability at the granularity of individual ALUs and L2 cache banks and provides unique interfaces to support low-overhead, dynamic configuration and monitoring. The runtime uses a combination of control theory and machine learning to configure the architecture such that QoS requirements are met and cost is minimized. Our results demonstrate that the combination of fine-grain configurability and non-convex optimization provides tremendous cost savings (70% savings) compared to coarse-grain heterogeneity and heuristic optimization. In addition, the system is able to customize configurations to particular applications, respond to application phases, and provide near optimal cost for QoS targets.**

I. Introduction

Currently, cloud customers have little to no control. CASH gives these customers control with a combination of a configurable architecture – which can be reconfigured on an extremely fine timescale – and an optimizing runtime system – which can meet quality-of-service (QoS) guarantees while minimizing cost. Such configurability, in combination with automated runtime management, gives customers fine-grain control. We believe that deployment of such a system would then also benefit cloud providers by attracting more customers.

The move to IaaS systems has spawned new architectures optimized for the data center and the Cloud. Microservers [4, 37, 59], data center optimized accelerators [19, 31, 43], and fine-grain reconfigurable processors [64] are all examples.

Highly configurable processors have unique benefits for the IaaS Cloud as resources can be moved between customers. For instance, configurable cache hierarchies [20, 23, 28, 44, 52] and configurable pipeline architectures (modifiable issue width, instruction window size, number of physical registers, etc.) [15, 41, 57, 64] allow fine grain control over resource scheduling. As fine-grain configurability is adopted in data centers, IaaS customers are left with the challenge of configuring and purchasing fine-grain resources to meet their QoS needs while minimizing their cost.

A. Challenges

On the one hand, fine-grain configurability has the potential to reduce costs by tailoring resource usage to the customers' demands. One the other hand, it greatly increases complexity, which could alienate customers. Indeed, finding the optimal configuration for an application and customer's needs is often a combinatorial optimization problem. Furthermore, as configurability becomes increasingly fine-grained, the optimization space becomes non-convex, further increasing the difficulty of optimization. To realize the potential of fine-grain configuration in IaaS, we must find simple interfaces that hide users from the complexity of fine-grain configuration while ensuring that configuration is guided to meet their needs.

B. The CASH Architecture and Runtime

This paper addresses the above challenges with CASH: **C**ost-aware **A**daptive **S**oftware and **H**ardware. CASH co-designs both a hardware architecture and a runtime management system to provide QoS guarantees while minimizing cost in support of IaaS customers. CASH allows customers to express their performance needs through a simple interface. The CASH runtime then configures the CASH architecture to meet those goals with minimal cost.

The CASH architecture consists of a homogeneous fabric of ALUs, FPUs, fetch units, and caches which can be dynamically composed to create cores optimized for a particular application. In an IaaS setting, this configurability enables the runtime to match the in-core resources to user performance goals, providing many of the benefits of heterogeneous multicores while maintaining a homogeneous fabric. The CASH architecture is inspired by the Sharing Architecture [64], but improves on it with fast reconfiguration, a well defined software-hardware interface which is

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.65

IEEE computer society

needed for the CASH runtime, and the use of an on-chip network to monitor remote cores' performance. Monitoring a remote core's performance is critical for the CASH runtime to be able to assess the effectiveness of its control, but is challenging as the CASH architecture does not contain fixed cores.

The CASH runtime overcomes the software challenges listed above. It uses a combination of control theory and reinforcement learning to provide QoS guarantees while adapting to application phases and non-convex optimization spaces. The control system ensures QoS guarantees are met while reinforcement learning adapts control to application phases and prevents the system from getting trapped in local optima.

C. Summary of Results

We evaluate CASH using a cycle-accurate simulator and 13 applications including the apache webserver, a mail server, and the x264 video encoder. For each application, we set QoS goals and use CASH to meet those goals while minimizing cost. Our results demonstrate the following:

- Fine-grain configurability enables tremendous potential cost savings, but that savings must be achieved by navigating non-convex optimization spaces that vary with application phase (Sec. II).
- The architecture and runtime support fast reconfiguration with low overhead (Sec. VI-A).
- The CASH runtime produces near-optimal results with rare QoS violations. In contrast, convex optimization techniques result in more QoS violations and increased cost (Sec. VI-C).
- CASH quickly adapts to application phases to minimize cost (Sec. VI-D).
- CASH produces over 3× cost savings compared to coarse-grain heterogeneous architectures (Sec. VI-E).

D. Contributions

In summary, this paper makes the following contributions:

- It extends previous configurable core architectures to provide rapid, low-overhead reconfigurability (Sec. III-B).
- It develops a novel hardware-software interface for monitoring a sub-core configurable architecture (Sec. III-B2).
- It demonstrates the difficulty of managing the non-convex optimization spaces that arise with fine-grain configurable architectures (see Sec. II).
- It describes how to provide guarantees while overcoming the difficulties of non-convex optimization through a combination of control theory and machine learning (see Sec. IV).
- It demonstrates the CASH approach through simulation showing that the combination of fine-grain configurability and adaptive management provides near optimal cost for just a small number of QoS violations (see Sec. VI).

While other architectures have employed fine-grain configurability to support IaaS providers, we believe this is the first approach designed to support IaaS customers through its combination of cost-optimizing architecture and runtime.

II. Motivational Example

There is ample evidence in the literature that fine-grain configurable architectures have the potential to produce greater energy efficiency and cost savings than static architectures or even coarse-grain configurable architectures [41, 57, 64]. The one drawback of these approaches is that fine-grain configurability creates complicated, *non-convex* optimization spaces characterized by the presence of local extrema that can vary as the application moves from phase to phase.

This section motivates the need to co-design both reconfigurable architecture and runtime management system. We first demonstrate the complexity of fine-grain resource management and then show its potential benefits for IaaS customers.

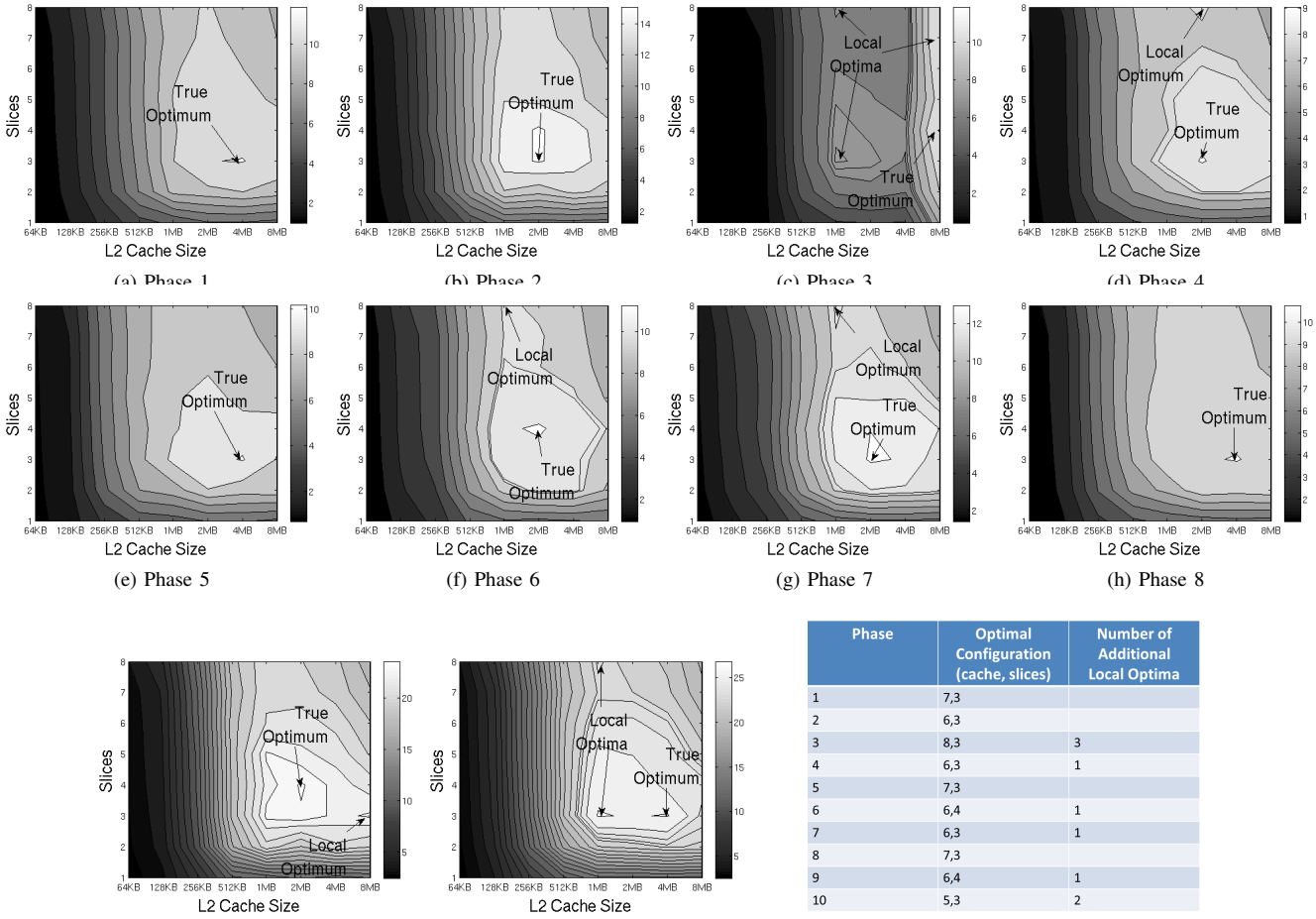
A. Allocating Resources within the CASH Architecture

We demonstrate the potential of fine-grain configuration by running the x264 video encoder [5] on the CASH Architecture (which extends the Sharing Architecture [64]). The video encoder is an excellent example of our target application as it has a clear QoS requirement: to maintain a particular frame rate. We would like to meet this frame rate while minimizing cost and adapting to phases in the source video.

The CASH Architecture supports this goal by allowing allocation of *Slices* and *L2 Cache banks*. Slices are simple out-of-order cores with 1 ALU, 1 Load-Store Unit, and a small L1 cache. Multiple Slices and L2 Cache banks can be combined to create a powerful *virtual core*. Users *rent* these resources and pay a cost based on their area. We would like to minimize resource usage (and thus cost) while ensuring QoS.

To demonstrate the optimization space, we run x264 on the CASH architecture with every possible virtual core constructed from 1 to 8 Slices and 64KB to 8MB L2 Caches in power of two steps. For our input video, we have identified 10 distinct phases of computation. For each phase and each virtual core configuration we measure the performance in instructions per clock. The results are shown in Fig. 1.

Figs. 1a–1j show contour plots that map virtual core configuration into performance. The x-axes show the L2 cache size (on a logarithmic scale), while y-axes show the number of Slices. Intensity represents performance, brighter shading indicates higher performance, with white being the



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

(e) Phase 5 (f) Phase 6 (g) Phase 7 (h) Phase 8

(i) Phase 9 (j) Phase 10

Phase	Optimal Configuration (cache, slices)	Number of Additional Local Optima
1	7,3	
2	6,3	
3	8,3	3
4	6,3	1
5	7,3	
6	6,4	1
7	6,3	1
8	7,3	
9	6,4	1
10	5,3	2

(k) Phase Breakdown

Figure 1: Phases of x264 executing on the CASH Architecture.

highest for a given phase. Fig. 1k shows a summary of the information in the contour plots.

This data clearly demonstrates the difficulty of allocating fine-grain resources to an application with distinct phases. Six of ten phases have local optima distinct from the true optimal performance. In addition, the location of the true optimal changes from phase to phase. In fact, **no two consecutive phases have the same optimal configuration.** In summary, any scheme that dynamically allocates Slices and L2 Cache banks to construct virtual cores must be capable of both avoiding local optima (*i.e.,* handling non-convex spaces) and adapting to phase changes. Avoiding local extrema is important as the true optimum is often far from local optima.

B. Benefits of Intelligent Resource Allocation

We have demonstrated the complexity of fine-grain resource management. We now show its potential benefits for IaaS customers. We again consider x264, this time with a QoS requirement that every phase meets the desired throughput. Optimal resource management must find the smallest virtual core configuration that ensures each phase runs at this speed. We compare optimal resource allocation to two other schemes: *convex optimization* and *race-to-idle*.

The convex optimization scheme uses a feedback control system to meet the QoS requirement [21]. It has a model of cost and performance tradeoffs and allocates resources to keep the performance at the desired QoS, but its convex model cannot account for local optima. We assume the race-to-idle scheme has some prior knowledge of the application and knows the lowest cost configuration that meets the QoS requirement in the worst-case. The race-to-idle approach allocates this worst-case virtual core for every phase. If a phase finishes early, this approach simply idles until the next phase begins. We (optimistically) assume that idling happens instantaneously and incurs no cost.

The results of this comparison are shown in Fig. 2. The plot on the top shows the cost. The plot on the bottom shows performance normalized to the QoS goal. The x-axes of both plots show cycle count (in millions of cycles), while the y-

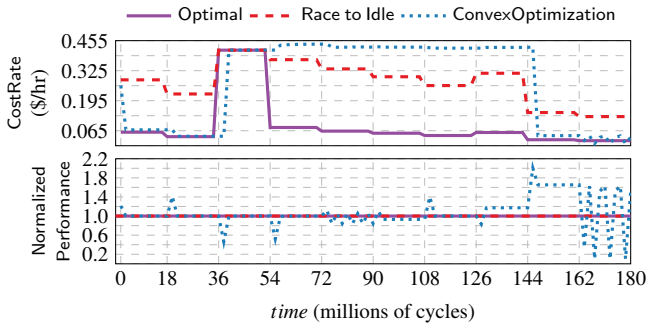


Figure 2: Comparison of fine-grain resource allocators.

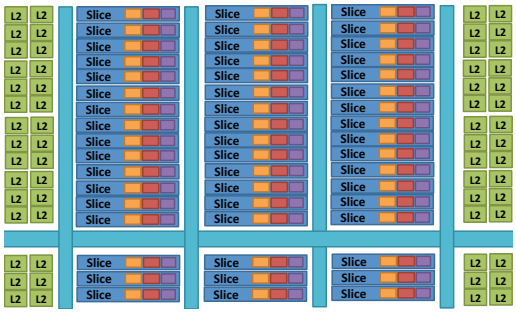


Figure 3: Slices, Cache Banks and Interconnection Network

axes show the cost and performance, respectively.

Two observations jump out immediately. First, convex optimization techniques cannot handle resource allocation for this application on this architecture as the incurred cost is much higher than optimal and the QoS requirement is repeatedly violated. Second, the race-to-idle approach does not violate QoS (given our optimistic assumptions), but incurs a much higher cost than necessary. In fact, for x264 **both race-to-idle and convex optimization produce over** 4.5× **optimal cost.** This is a significant additional cost for the customer to bear. Clearly, there is a need to help IaaS customers by combining fine-grain configurability with automated resource management. *CASH's goal is to achieve near-optimal fine-grain resource management with minimal QoS violations.*

III. The CASH Hardware Architecture

A. Architecture Overview

The CASH architecture extends the Sharing Architecture [64] – a prior configurable core architecture – by (1) innovating in fast reconfiguration, (2) providing a well designed hardware-software interface, and (3) allowing remote cores' performance to be monitored over an on-chip network. In this section, as background, we begin by describing the overall ideas of the CASH Architecture and then in Section III-B we describe how we co-designed the CASH Architecture to have support for fast reconfiguration.

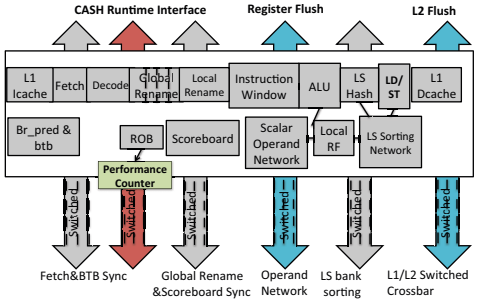


Figure 4: Slice Overview and Inter-Slice Network

CASH is a fine-grain configurable architecture which provides the flexibility to dynamically "synthesize" a virtual core with the correct amount of ALUs and cache, based on an application's demand and the need to optimize cost. Unlike multicore and manycore architectures where the processor core is statically fixed at design and fabrication time, the CASH Architecture enables the core to be reconfigured and use resources across a single chip. This fine-grain reconfigurability is especially useful for IaaS applications which have fixed performance goals (in terms of acceptable latency or throughput) and need to meet those goals while minimizing cost.

At the top level, like existing multicore chips used for IaaS applications, CASH can group multicore cores into *Virtual Machines* (VMs). Unlike fixed architecture multicore processors, the VMs in the CASH Architecture are composed of cores which themselves are composed of a variable number of ALUs, and cache. We call this flexible core a *virtual core*. A virtual core is composed out of one or more *Slices* and one or more *L2 Cache Banks*. Figure 3 shows an example array of Slices and Cache Banks. A full chip contains 100's of Slices and Cache Banks. The key insight is to construct highly scalable architectures on a homogeneous fabric.

The basic unit of computation in the CASH Architecture is a *Slice* as shown in Figure 4. A Slice is effectively a simple out-of-order processor with one ALU, one Load-Store Unit, the ability to fetch two instructions per cycle, and a small Level 1 Cache. Multiple Slices can be grouped together to increase the performance of sequential programs thereby empowering users to make decisions about trading off ILP vs. TLP vs. Process level parallelism vs. VM level parallelism while all utilizing the same resources.

The CASH Architecture is designed to enable very flexible sharing of resources and does not impose hierarchical grouping of Slices or Cache Banks. This feature enables IaaS Clouds to spatially allocate computation resources. In order to achieve this, we use general purpose switched interconnects wherever possible. Neither Slices or Cache Banks need to be contiguous in a virtual core for functionality. But for the purpose of performance, we group adjacent Slices into a single virtual core to reduce operand communication cost

and Cache access latency. Cache configuration is decoupled from the Slices, and the Cache access latency is modeled in proportion to the distance from the Cache to the Slices and Cache size. We do not see the need for contiguous Slices to be a limitation as all Slices are interchangeable and equally connected. Therefore, fixing fragmentation problems is as simple as rescheduling Slices to virtual cores. As larger Cache size leads to higher communication cost, virtual core configuration search space can be non-convex.

The CASH Architecture minimizes changes to the software stack by utilizing multiple switched interconnect whenever possible. Distributed local register files enabled by the two register renaming stages improve virtual core scalability. Applications do not need to be recompiled to use different Slice and Cache configurations. This property is key, as it allows the CASH runtime to dynamically change architectural configurations without requiring any application-level changes.

B. Reconfiguration

1) Hardware Reconfiguration: To reconfigure virtual cores, we rely on the CASH runtime to control the connections of the Slice and Cache interconnect. The runtime is time multiplexed with the client virtual machines. But, unlike client VMs which run on reconfigurable cores, we propose having the runtime execute only on single-Slice virtual cores. By having the runtime execute on single Slices only, it can then locally reconfigure protection registers and interconnection state to setup and tear-down client virtual cores. The runtime bypasses use of the reconfigurable cache system to prevent having to flush the L2 Caches on every time Slice.

When the number of Slices in a virtual core is reduced, there is potentially register state which needs to be transmitted to the surviving Slices within the virtual core. Because we use a distributed local register file scheme, one architectural register can have multiple copies in different Slices. In this case, only one copy (the primary one that originally wrote the value to the register) for each architectural register is conserved. In order to carry this out, we use a Register Flush instruction which pushes all of the dirty architectural register state to other Slices in the same virtual core using operand forwarding messages. This can be done quickly because there are only a limited number of global logical registers, only one copy of data needs to be saved, and the Scalar Operand Network is fast for transferring register data. Architectural registers are mapped onto global logical registers with global logical registers serving as a register name space which is mapped across all of the Slices, while local registers are the actual storage registers in individual Slices. When virtual core's L2 Cache configuration changes, all dirty state in L2 Cache Banks that are being removed must be flushed to main memory before reconfiguration.

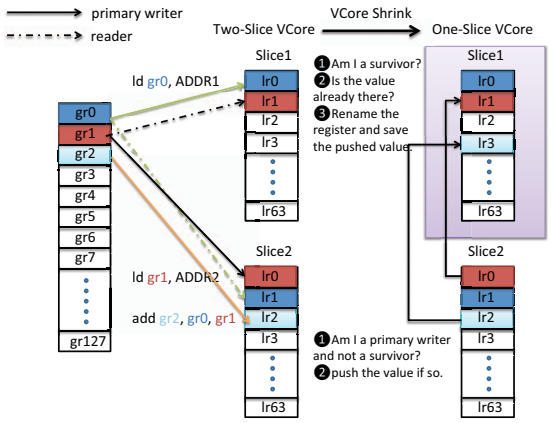


Figure 5: Register Flush When Shrinking a Virtual Core (VCore)

Figure 5 shows an example of flushing registers when shrinking a two-Slice virtual core to a one-Slice virtual core. Originally global registers gr0, gr1, and gr2 are primarily written and renamed to lr0 by Slice1, lr0 by Slice2 and lr2 by Slice2. Slice1 has an additional copy of gr1 due to an instruction read, and gr1 is renamed to lr1 by Slice1 locally. Similarly, Slice2 has a copy of gr0 renamed to lr1 locally. When the virtual core shrinks from two-Slices to one-Slice, the survivor, Slice1, needs to get all the updated register values from Slice2. As Slice2 is the primary writer of gr1 and gr2, it flushes both values to Slice1. Upon receiving the values, Slice1 determines to only save gr2 to lr3 as it already has a copy of gr1 due to a previous register read. Because only the primary writers need to flush register values, the total number of flushes is bounded by the total number of global registers.

2) Interface to Software: The software adaptation engine requires information such as performance and cost from the configurable hardware. The hardware communicates the performance and estimated cost to the software adaptation engine. The software engine uses this information to make more informed reconfiguration decisions after a pre-determined reconfiguration interval. We use instruction commit rate as a measure of performance. One challenge is how best to interface between the software adaptation engine and the hardware. This is made even more difficult because performance counter registers are typically read at the core level and the CASH Architecture has no fixed core.

To solve the challenge of performance counting without fixed cores, the CASH Architecture connects performance counters to a dedicated on-chip network, as shown in Figure 4. The CASH Runtime Interface Network allows a virtual core with sufficiently high privilege (*e.g.,* executing the CASH runtime) to measure performance counters on other virtual cores. The CASH runtime can read instruction commit rate, cache miss rate, branch miss-predict rate or

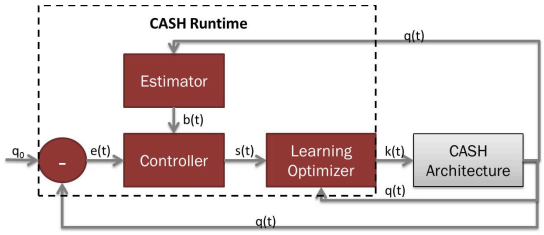


Figure 6: The CASH runtime.

other common performance counter information. Performance counters are queried with a simple request and reply protocol targeting a particular Slice. Each performance counter sample is timestamped which enables the CASH runtime to synthesize overall performance for a virtual core from performance counter readings from individual Slices.

The CASH Runtime Interface Network is also used to send reconfiguration commands such as EXPAND and SHRINK which target a particular Slice or L2 cache bank. Removing a Slice from a virtual core requires that the primary written registers be flushed to the surviving Slices through the Operand Network. L2 Cache banks which are being removed flush their dirty data to main memory across the L2 memory network. The data networks already exist in the Sharing Architecture, but the CASH Runtime Interface Network is newly added to support fast reconfiguration and performance gathering.

IV. Runtime Resource Allocation

The CASH runtime configures Slices and L2 Cache to meet an application's QoS demands while minimizing cost. The runtime addresses the software challenges listed in the introduction: (1) guaranteeing QoS guarantees, (2) adapting to phases in application workload, and (3) avoiding local optima. Additionally, the runtime is designed to be very low overhead.

The CASH runtime is illustrated in Fig. 6. It takes a QoS goal q and measures the current QoS $q(t)$ at time t. It computes the error $e(t)$ between the goal and the delivered QoS. This error is passed to a Controller which computes a speedup $s(t)$ to apply to the application. The controller is modified online by an Estimator that adapts to application phases. The speedup signal is passed to a LearningOptimizer which determines the minimal cost configuration $k(t)$ of Slices and L2 to achieve the desired speedup. The Controller enforces QoS requirements, the Estimator recognizes application phases, and the LearningOptimizer selects the lowest cost configuration. We discuss each in more detail below.

A. A Control System for QoS

CASH is influenced by prior research that has effectively deployed control systems to meet QoS demands [17, 22, 24, 25, 35, 48, 51, 63]. These control approaches measure runtime QoS feedback and compute a *control signal* indicating a resource allocation that maintains the desired QoS. The CASH runtime first computes the *error* $e(t)$ between the QoS requirement q_0 and the current QoS level $q(t)$:

$$e(t) = q_0 - q(t) \tag{1}$$

The controller eliminates this error. There are tradeoffs between how fast the controller drives the error to zero and its sensitivity to noise. The CASH runtime employs a *deadbeat* controller to eliminate the error as fast as possible [34] and corrects noise in the Estimator. Therefore, CASH computes speedup $s(t)$ given an error signal:

$$s(t) = s(t-1) + \frac{e(t)}{b} \tag{2}$$

where b represents the base QoS of the application; *i.e.,* its QoS when running on one Slice with a 64KB L2.

Eqns. 1 and 2 together comprise a standard control system similar to prior approaches, but with some limitations. It will react to phases, but not as quickly as we might like. Furthermore, it has no ability to adapt to the types of non-convex optimization spaces illustrated in Fig. 1. In the next two section, we extend this type of standard control design to provide these two features.

B. Estimating Application Phase Changes

The CASH runtime must adapt to phases like those discussed in Sec. II. The key parameter for handling phases is the constant value b in Eqn. 2 representing base QoS. Analytically, a change in phase represents a fundamental shift in the value of this parameter. It is not feasible, however, to directly measure base speed as doing so would likely violate QoS. Instead, the CASH runtime continually estimates this value, $\hat{b}(t)$. When the application changes phase, this estimate will change, and the speedup values produced by Eqn. 2 will reflect the phase shift. For example, if the base speed increases by $2\times$ then we would like the speedup produced by the controller to drop by this same factor. Clearly, an accurate estimate of base speed substituted into Eqn. 2 will produce the desired effect.

CASH learns b using a Kalman filter [58] based on the time-varying model:

$$\begin{aligned} b(t) &= b(t-1) + \delta b(t) \\ q(t) &= s(t-1)\,b(t-1) + \delta q(t) \end{aligned} \tag{3}$$

which describes b and q subject to both *disturbance*, *e.g.,* a page fault, and *noise*, *i.e.,* natural variation in application's QoS signal, (respectively: δb and δq).

Denoting the system QoS variance and measurement variance as $v(t)$ and r, the Kalman filter formulation is

$$\begin{aligned} \hat{b}^-(t) &= \hat{b}(t-1) \\ E^-(t) &= E(t-1) + v(t) \\ Kal(t) &= \frac{E^-(t)\,s(t)}{[s(t)]^2 E^-(t) + r} \\ \hat{b}(t) &= \hat{b}^-(t) + Kal(t)\,[q(t) - s(t)\hat{b}^-(t)] \\ E(t) &= [1 - Kal(t)\,s(t-1)]\,E^-(t) \end{aligned} \tag{4}$$

where $Kal(t)$ is the Kalman gain for the QoS, $\hat{b}^-(t)$ is the *a priori* estimate of $b(t)$ and $\hat{b}(t)$ is the *a posteriori* one, and $E^-(t)$ is the *a priori* estimate of the error variance while $E(t)$ is the *a posteriori* one. CASH uses a Kalman filter because it produces a statistically optimal estimate of the system's parameters, and is provably exponentially convergent [10]. Practically, this means that the number of time steps required to detect phase changes (*i.e.,* changes in the estimate of base speed) will be – in the worst case – logarithmic in the difference between the base speeds for two consecutive phases; *i.e.,* convergence time $\propto \log(|b_{phase\ i} - b_{phase\ i+1}|)$. Note that the only required (*i.e.,* not directly measured from the hardware) parameter is r, the measurement noise, which we take as a constant property of the hardware architecture.

C. Learning to Minimize Cost

The combination of the Controller and Estimator ensures QoS and reacts quickly to phases. It does not, however, have a notion of optimality. The LearningOptimizer from Fig. 6 determines the architectural configuration that meets the speedup signal $s(t)$ with minimal cost; *i.e.,* it maps the speedup signal into the minimal cost Slice and L2 configuration while avoiding local optima.

The CASH runtime models the architecture as a set of K configurations where each configuration $k \in K$ is a specific number of Slices and L2 size. Each k has a speedup s_k and cost c_k. CASH schedules configurations over a quantum of τ time units, such that τ_k time is spent in each configuration. CASH formulates the cost minimization problem as:

$$\begin{array}{rcll} minimize(\tau_{idle}c_{idle} + \frac{1}{\tau}\sum_{k\in K}(\tau_k c_k)) & & s.t. & \\ \frac{1}{\tau}\sum_{k\in K}\tau_k s_k & = & s(t) & \\ \tau_{idle} + \sum_{k\in K}\tau_k & = & \tau & \\ \tau_k, \tau_{idle} & \geq & 0,\ \forall k & \end{array} \quad (5)$$

The first line in Eqn. 5 minimizes the total cost of the schedule. The second line ensures that the average speedup of the schedule is equal to the speedup produced by the Controller (Eqn. 2). The third and fourth lines ensure that the work is completed by the deadline (for QoS).

While this system has many variables, we know from the theory of linear programming that, because it has only two constraints, there is a solution where only two of τ_k are non-zero and all others are zero [8]. Specifically, the two configurations that will have non-zero times are *over* and *under* where:

$$\begin{array}{rcl} over & = & \text{argmin}_k\{c_k | s_k > s(t)\} \\ under & = & \text{argmax}_k\{s_k/c_k | s_k < s(t)\} \\ t_{over} & = & \tau \cdot \dfrac{s(t) - s_{under}}{s_{over} - s_{under}} \\ t_{under} & = & \tau - t_{over} \end{array} \quad (6)$$

Thus, if we know s_k and c_k for all k then the optimization problem is easy to solve [30]. c_k is the cost per configuration per unit time and set by a systems administrator (*e.g.,* it could be $ per chip area). The difficulty is finding s_k, which can vary tremendously in different phases of an application. Therefore, the CASH runtime treats the speedup as a time-varying parameter, observes the achieved QoS $q(t)$ and uses a *Q-learning* approach to learn the true speedup online [22, 53]. This approach has the advantage that it is computationally cheap, but treats all configurations as independent. More sophisticated learning methods that capture correlation between configurations and applications (*e.g.,* [40]) will be the subject of future work. Using Q-learning, the learned speedup for configuration k at time t is represented as $\hat{s}_k(t)$:

$$\begin{array}{rcl} \hat{q}_k(t) & = & (1-\alpha)\cdot\hat{q}_k(t-1) + \alpha\cdot q(t) \\ \hat{s}_k(t) & = & \frac{\hat{q}_k(t)}{\hat{q}_0(t)} \end{array} \quad (7)$$

D. Runtime Summary

The CASH runtime puts the Controller, Estimator, and LearningOptimizer together in Algorithm 1. The runtime executes an infinite loop. At each iteration, it updates its estimate of base speed, then uses that base speed to compute a control signal. The control signal is passed to the optimizer, which computes the optimal configurations for achieving the control signal. The runtime puts the architecture into that configuration and then updates its estimates of the speedups that configuration will produce in the future. The runtime computational complexity is $O(1)$, making it low overhead. As discussed above, it is exponentially convergent to the QoS goal despite changes in application base speed. The LearningOptimizer adapts speedup models online to maintain optimality despite application changes and non-convex optimization spaces.

Table I: Base Slice Configuration

Number of Functional Units/Slice	2	ROB size	64
Number of Physical Registers	128	Store Buffer Size	8
Number of Local Registers/Slice	64	Maximum In-flight Loads	8
Issue Window Size	32	Memory Delay	100
Load/Store Queue Size	32		

Algorithm 1 The CASH Runtime.

loop
- Read current QoS $q(t)$.
- Compute $\hat{b}(t)$, the estimate of the current base speed, using Eqn. 3.
- Compute $s(t)$ according to Eqn. 2, but substitute $\hat{b}(t)$ for b.
- Solve for t_{over} & t_{under} using Eqn. 6, but substitute speedup estimates.
- Put the architecture in configuration *over* for t_{over} time.
- Sleep for t_{over} time.
- Put the architecture in configuration *under* for t_{under} time.
- Sleep for t_{under}.
- Compute $\hat{s}_{over}(t)$ and $\hat{s}_{under}(t)$, according to Eqn. 7.

end loop

V. Experimental Setup

A. Simulation Infrastructure

We have created a cycle-accurate simulator, *SSim*, to model the CASH Architecture and measure its effectiveness. SSim models each subsystem of the CASH Architecture

Table II: Base Cache Configurations: L2 cache hit delays depend on the distance to the cache bank

Level	Size(KB)	Block Size(Byte)	Associativity	Hit Delay
L1D	16	64	2	3
L1I	16	64	2	3
L2	64*2	64	4	distance*2+4

(fetch, rename, issue, execution, memory, commit, and on-chip network) along with accurately modeling the Out-of-Order execution and inter-Slice and Slice-to-memory latency. SSim is driven by the full system version of the the Alpha ISA GEM5[6] simulator. Table I shows the base Slice configuration and Table II shows the base Cache configuration for our simulations.

B. Benchmark Applications

We use the complete SPEC CINT2006 benchmark suite [50], a subset of PARSEC benchmarks [5], the apache web server [3], and the postal mail server [12] to explore the CASH architecture. They are representative benchmarks that provide a measure of performance across a wide practical range of workloads. Another reason we select these benchmarks is because there are abundant phases in the programs [15]. Programs with phases can benefit most from the reconfigurable hardware. Each PARSEC benchmark denotes a region of interest (ROI) indicating the performance critical part of the benchmark. We consider only the ROI for PARSEC. We evaluate apache serving webpage requests with a concurrency of 30.

C. Experimental Methodology

To quantify CASH's ability to minimize cost, we construct an *oracle* that returns the lowest possible cost configuration for any performance goal. We construct this oracle by running all applications in every possible configuration of the CASH architecture. We manually determine (by examining the simulation output) any distinct processing phases in the application and we then (through brute force) determine the combination of resources that is lowest cost for any performance goal. This brute force exploration allows us to construct the oracle representing true minimal cost.

VI. Evaluation

This section evaluates CASH's ability to aid IaaS customers by delivering QoS for minimal cost. We first evaluate both architectural and runtime overhead. We then demonstrate CASH providing near optimal cost for a variety of benchmarks with QoS requirements. We next show time-series data demonstrating how CASH quickly converges to the required performance. Finally, we compare CASH to a heterogeneous multicore architecture, which provides statically heterogeneous core types; *i.e.,* determined at design time [29, 32], rather than CASH's fine-grain configurable cores.

A. Overhead of Reconfiguration

One of the challenges to exploiting fine-grain reconfigurability is minimizing the architectural and runtime overheads.

Architectural Overhead: There are four sources of microarchitectural overhead: Slice expansion, Slice contraction, L2 expansion, and L2 contraction. Slice expansion is fast – requiring only a pipeline flush – approximately 15 cycles. Slice contraction takes at most 64 cycles more than expansion to flush local register values to the remaining Slices. Both L2 expansion and contraction require flushing dirty cache lines. Assuming all lines are dirty, a flush takes $(BankSize)/(NetworkWidth)$ cycles; *e.g.,* with L2 bank size of 64KB and network width of 64 bits, flushing the L2 takes $64KB/8B = 8000$ cycles. Note this is the worst case cost, in practice we expect that we will not flush the whole cache as only a small number of lines will be dirty. We use a hash table to map physical address to cache banks, the reconfiguration of which is overlapped with dirty line flushing.

Runtime Overhead: We measure the average cycle count to execute Algorithm 1's 'C' implementation. We note that the execution time of the algorithm is not application-dependent. We therefore time 1000 iterations of the runtime for the x264 benchmark and report the average cycle count. With one Slice, the runtime takes just under 2000 cycles per iteration. Two and three Slices produce times of 1100 and 977 cycles respectively. This low overhead means a runtime executing on even a single Slice could easily service many applications. As we expect CASH architectures to scale to 100s of Slices, the area overhead of dedicating one slice to the runtime is extremely small.

B. Cost Model

CASH supports IaaS customers by making computer resources rentable at a fine-granularity. To demonstrate this, we assign a cost per unit area. Following Amazon's EC2 pricing (which uses a linear model for additional capacity within a resource class), we assume a linear increase in price per unit area for CASH [2]. Specifically, we assume $.013 per hour for the minimal architectural configuration of 1 Slice and a 64KB L2 Cache which is the same price Amazon charges for on-demand usage of their t2.micro configuration. Based on silicon area fed by a Verilog implementation of the CASH hardware, this price structure is equivalent to $.0098 per hour for a Slice and $.0032 per hour for 64KB of Cache. We stress that the absolute value of the price does not affect our conclusions. Our comparisons rely on the ratios of the cost incurred by various architectures and resource managers.

C. Cost Savings for QoS Requirement

Our first case study shows that CASH handles a variety of applications, providing near minimal cost with rare QoS violations. For each application, we first determine the

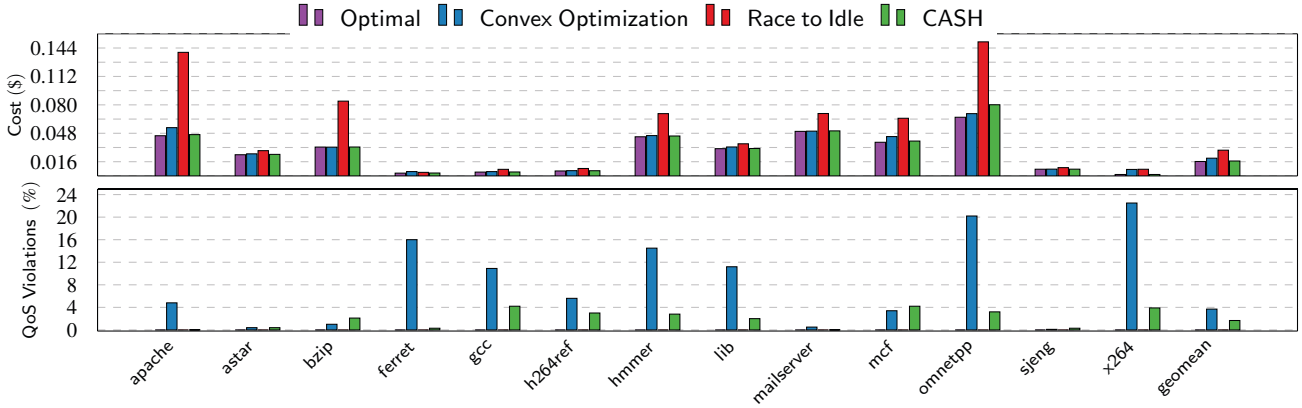


Figure 7: Cost and QoS violations for different fine-grain resource allocators. (lower is better)

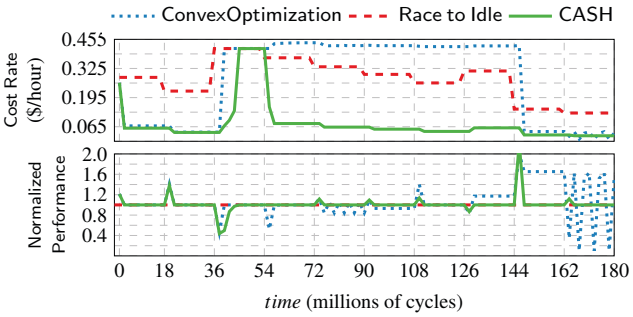


Figure 8: Time series data for x264.

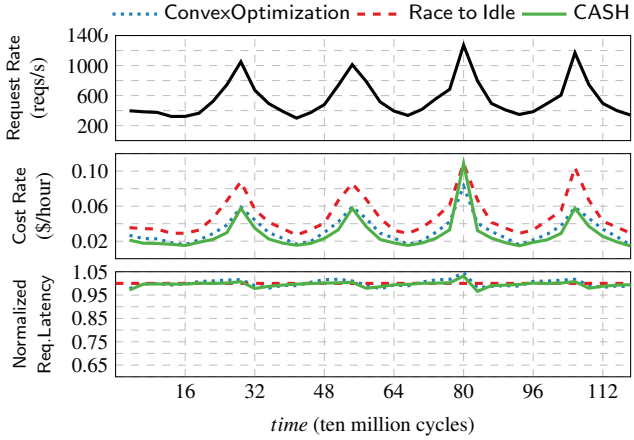


Figure 9: Time series data for apache.

QoS requirement as the highest worst case IPC seen for any application; *i.e.,* we tell the runtime system to always be at or above worst case performance. We then deploy the application on CASH, which dynamically allocates the minimal number of Slices and L2 Cache banks to maintain the QoS goal.

We compare CASH to three other approaches. First, we compare to our oracle which determined the optimal resource allocation for each application and QoS target. Second, we compare to a *convex optimization* scheme that uses control theory (similar to what is described in Sec. IV-A), but does not incorporate learning and must rely on a single convex model that captures average case behavior for the entire application. Third, we compare to the *race-to-idle* approach, which uses the configuration that meets QoS in the worst case and idles (incurring no additional cost) when completing early. For each application **we sample performance 1000 times**, recording both the *total cost* and QoS violations.

Fig. 7 presents the data comparing these four resource allocation schemes. In the figure, the top chart shows the total cost for each application while the bottom chart shows the percentage of QoS violations.

The geometric mean of costs for all applications is listed in Table III. This table shows that convex optimization and race-to-idle incur significant costs compared to optimal. Convex optimization is clearly not a good choice for fine-grain resource management as it has high costs and high QoS violations. In contrast, race-to-idle produces no QoS violations (under the assumption that the worst case resource allocation is known *a priori*). These strong QoS guarantees for race-to-idle, however, incur cost over 1.7× the optimal. CASH operates at a reasonable middle ground, as it incurs only 4% more cost than optimal and less than 2% QoS violations. For all applications in this study, CASH delivers the specified QoS at least 95% of the time. We note one application, omnetpp, where convex optimization produces lower cost than CASH, but this comes with the penalty of 20% QoS violations. In this case, convex optimization has reduced cost, but at the penalty of large-scale QoS violations.

D. Adapting to Application and Workload Phases

We present time-series data for individual applications to show how CASH can aid IaaS customers by adapting to

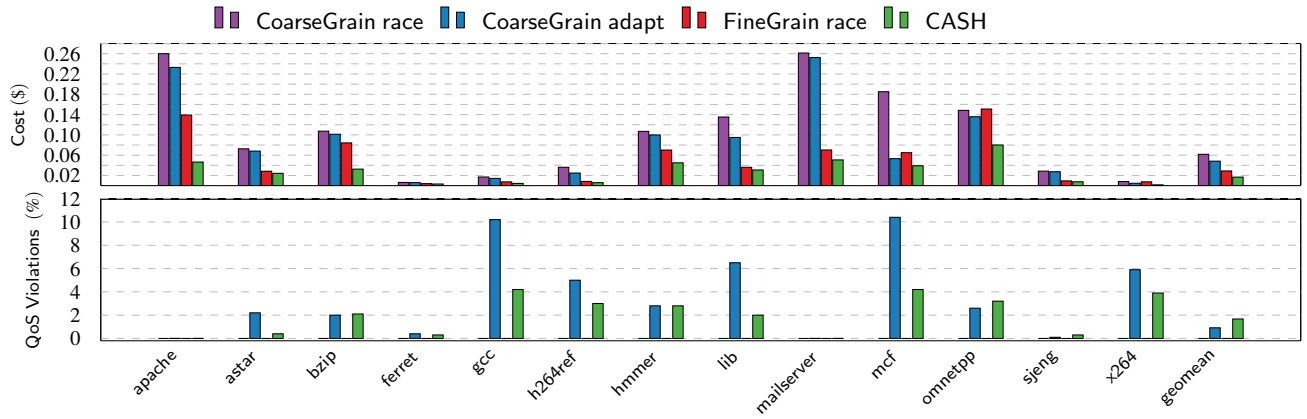


Figure 10: Cost and QoS for combinations of coarse and fine grain reconfigurable architectures and resource allocators.

Table III: Cost comparison for different resource allocators.

	Geometric Mean	Ratio to Optimal
Optimal	$0.0162	1.00
Convex Optimization	$0.0199	1.23
Race to Idle	$0.0289	1.78
CASH	$0.0168	1.03

application and workload fluctuations. We show results for the x264 video encoder and the apache webserver.

We use the same setup as the previous section, but now we report the measured cost rate ($/hour) and performance (normalized to the QoS target) as a function of time. The results are shown in Figs. 8–9. We compare convex optimization, race-to-idle, and CASH.

1) x264: These results show that adaptive, fine-grain resource allocation provides tremendous cost savings for x264 compared to both race-to-idle and convex optimization. If we relate the x264 data from Fig. 8 to that in Fig. 1, we see that between 36 and 54 Mcycles (phase 3 from Fig. 1) has several local extrema and a very expensive true optimal. The convex optimization approach reaches the true optimal at this time, but then stays in the expensive configuration until 144 Mcycles. The CASH runtime, in contrast, detects the application's behavior change and reallocates resources to reduce cost.

2) apache: The results for the apache webserver are shown in Fig. 9. For this experiment, we construct an oscillating stream of requests, typical for web servers (*e.g.,* Wikipedia [56]). In general these oscillations occur over the course of the day. Simulation is too expensive to handle days worth of requests, so we condense the stream into much faster oscillations. The request rate as a function of time is shown in the top chart of Fig. 9. We set a QoS requirement of 110K cycles per request, which corresponds to the smallest possible worst-case latency on CASH. When request rates are low, it will be easy to hit the target latency. When request rates are high it will require more resources.

The cost and delivered QoS are shown in the middle and bottom charts of Fig. 9. All methods adapt to changing request rates and keep the actual latency close to the target. The race-to-idle approach is the most expensive, however, as it always reserves resources to meet the latency in the worst-case. The worst case is only realized around 800 Mcycles, though. Convex optimization saves compared to race-to-idle, but the CASH runtime is the cheapest, providing about an 18% cost reduction compared to convex optimization.

These results demonstrate CASH's advantage for IaaS customers. Fine-grain configurability allows the customers to provision just the right amount of resources. The adaptive runtime allocates for current case, providing considerable savings over the naive approach of reserving for the worst case and giving up resources when they are not needed.

E. Comparison to Heterogeneous Architectures

To demonstrate the advantages of fine-grain configurability for IaaS customers, we compare the CASH approach to a heterogeneous architecture, similar to the ARM big.LITTLE. The big.LITTLE's coarse-grain approach combines fast, expensive cores with slower, cheaper cores. To compare against this coarse-grain approach, we simulate the CASH architecture with one big core and one small core. The big core is the largest configuration needed to meet the QoS demands of all target applications: 8 Slices with a 4MB L2. The little core is the most cost efficient configuration, on average, across all our benchmarks: 1 Slice with a 128KB L2.

We want to isolate the cost savings of both the CASH architecture and runtime. Therefore, we perform resource allocation on both the heterogeneous architecture and the CASH architecture using both race-to-idle and the CASH runtime. This gives four points of comparison: CoarseGrain, race; CoarseGrain, adaptive; FineGrain, race; and CASH, which is fine-grain and adaptive. To be clear, the race-to-idle approach cannot change core configuration, but the CoarseGrain, adaptive approach uses the CASH runtime

to dynamically shift between big and little cores. We run each benchmark using these combinations of architectures and management systems. The results are shown in Fig. 10.

The figure shows cost in the top chart, and the average number of QoS violations in the bottom. These results show the combined power of CASH's fine-grain configuration and an adaptive runtime. The geometric mean of cost for CoarseGrain, race is 0.062$. Replacing race-to-idle with adaptive resource allocation creates the CoarseGrain, adaptive data point, with geometric mean cost $0.048. FineGrain, race produces geometric mean cost $0.029, while CASH's geometric mean cost is $0.017. Adaptation reduces geometric mean cost by about 25%. Fine-grain reconfigurability, by itself, reduces costs by more than 50%. **CASH's combination of fine-grain configurability and adaptive resource management reduces the geometric mean of cost by over 70% compared to racing-to-idle on a heterogeneous architecture.**

These results summarize the case for the CASH approach. The finer-granularity the architecture, the better it is able to match the needs of the application. Combining such a fine-grain architecture with an intelligent management system allows even greater cost savings.

VII. Related Work

A. SMT

Simultaneous multi-threading (SMT) improves out-of-order core efficiency by time multiplexing the processor pipeline [16]. CASH also allows resources that are not used by one core to be repurposed, but CASH is a fundamentally different design point from SMT, as CASH focuses on spatial partitioning. For instance, CASH supports cores with many ALUs and a small cache, few ALUs with a large cache, or many other combinations that are not possible in SMTs. Additionally, in multicore SMTs, core size is chosen at fabrication time, while CASH can be configured at runtime. Therefore CASH does not have to face the classic problems that SMT architectures face around resource thrashing; e.g., destructive cache interference when two threads share the same SMT core.

B. Configurable Architectures Without Policies

The CASH architecture leverages many ideas from Distributed ILP work and fused-core architectures. Distributed ILP removes large centralized structures by replacing them with distributed structures and functional units. This idea has been explored in Raw [55], Tilera [4, 23, 59, 60], TRIPS [46, 47], and Wavescalar [54]. Distributed tiled architectures distribute portions of the register file, cache, and functional units across a switched interconnect. The CASH architecture adopts a similar network topology for communicating instruction operands between function units in different Slices. Unlike Raw and Tilera, we do not expose all of the hardware to software, do not require recompilation, and use dynamic assignment of resources, dynamic transport of operands, and dynamic instruction ordering. TRIPS [9, 46] has an array of ALUs connected by a Scalar Operand Network and uses dynamic operand transport similar to the CASH architecture. Unlike CASH, TRIPS requires compiler support, and a new EDGE ISA which encodes data dependencies directly. In contrast, CASH uses a dynamic request-and-reply message based distributed local register file to exploit ILP across Slices without the need of a compiler.

The CASH architecture leverages many ideas from Core Fusion [27] to distribute resources across cores, but unlike Core Fusion, CASH is designed to scale to larger numbers of Slices (cores). CASH distributes the structures that are centralized in Core Fusion, such as structures for coordinating fetch, steering, and commit across fused cores. Also, we take a more Distributed ILP approach to operand communication and other inter-Slice communication by using a switched interconnect between Slices instead of Core Fusion's operand crossbar. Also, the CASH is designed as a 2D fabric which is created across a large manycore with 100's of cores which can avoid some of the fragmentation challenges that smaller configurations like Core Fusion can have. The interconnect of Slices to Cache is also designed to be flexible across the entire machine unlike Core Fusion which has a shared cache. This can lead to better efficiency and provides a way to partition cache traffic and working sets. The CASH architecture is explicitly co-designed with the CASH runtime. Thus, it supports software driven reconfiguration and monitoring, which is not supported by previous designs.

C. Resource management

PEGASUS [36], a feedback-based controller has been proposed to improve energy proportionality of warehouse scale computer systems. Hardware/Software co-design for datacenter power management [13, 14, 39] improves datacenter power proportionality while maintaining QoS. MITTS [65] is a distributed hardware mechanism which shapes memory traffic based on memory request inter-arrival time, enabling fine-grain bandwidth allocation. CASH manages fine-grain computational resources, which is complementary to those techniques. Runtime systems for datacenter resource sharing [61] improve server utilization while guaranteeing QoS. Bubble-Flux measures instantaneous pressure on the shared hardware resources and controls the execution of batch jobs. All above mentioned techniques are policies for system operators/architects, CASH on the other hand, focuses on **policies for Cloud users**. It helps Cloud customers to make rational decisions on virtual core configurations of sub-core configurable architecture.

Scheduling workloads in a IaaS Cloud requires understanding both the software stack and the hardware architecture. Scheduling algorithms for heterogeneous processors have been proposed for MS Bing's index search [45]. While heterogeneous processors improve interactive data

center workloads performance by supporting heterogeneous query lengths, they do not support fine-grain workload heterogeneity. This algorithm is not applicable to fine-grain configurable architecture.

Offline architectural optimization approaches have been proposed [11, 33, 62], but they have limited applicability to online adaptation. Predictive models [15, 49] have been proposed to perform online adaptation using offline training. Flicker [42] also assembles "virtual cores" using dynamic management, but is designed to maximize system throughput for a power budget. CASH tackles the complementary problem of ensuring performance for minimal cost. While similar, these optimization problems have different geometries and require different techniques to solve efficiently. Machine learning based techniques [7, 26] have been proposed for adaptation of shared caches and off-chip memory bandwidth. These approaches maximize throughput, but do not consider constrained optimization problems; *i.e.,* meeting a performance goal for minimal cost. In addition, the neural networks in these approaches requires significant offline training and the delivered performance is sensitive to the training set. In contrast, CASH requires no *a priori* knowledge or training.

Resource-as-a-Service [1] has been introduced as a model to sell individual resources (such as CPU, memory, and I/O resources) in accordance with market-driven supply-and-demand conditions. CASH enables sub-core fine-grain resource selling by proposing a run-time system supporting sub-core configurable architecture.

D. Control-based Resource Management

Like many prior approaches (e.g., [22, 24, 25, 35, 38, 48, 51, 63]), CASH uses control theory to manage resources and meet performance goals. Control theory is a general *technique* that allows formal reasoning about the dynamic behavior of the controlled system [17, 21]. While the technique is general, implementations are often highly specific to individual applications (e.g., an embedded video encoder [38]). Some prior work has provided more general implementations by implementing control systems at the middleware layer [18, 35, 63]. The middleware handles the complicated construction of the control system, but the application writer must be aware of the components on the system that can be controlled and is responsible for modeling those components. Other approaches have provided automated modeling [17] or split the modeling process into a piece that is provided by the application developer and a piece that is provided by the system developer [22, 24, 25]. CASH differs as it does not require application programmer input at all except for the specification of the performance target. The cost of this generality is that the CASH runtime adapts models on the fly and can sometimes miss performance goals.

VIII. Conclusion

We have presented CASH, an architecture and runtime co-designed to support IaaS customers. The CASH architecture makes it possible to rent computational resources at an extremely fine granularity. The CASH runtime allocates these resources to applications to meet QoS requirements and minimize cost. Our results show that CASH successfully addresses the challenges listed in the introduction: supporting fast reconfiguration, meeting QoS goals, adapting to application phases, and handling non-convex optimization spaces. Furthermore, our comparison with coarse-grain heterogeneous architectures shows that CASH's fine-grain configurable approach provides significant cost savings – on average over 70%. We conclude it is both possible and profitable to (1) expose fine-grain architectural tradeoffs to software and (2) dynamically navigate these tradeoff spaces to save IaaS customers' money.

Acknowledgements

This work was partially supported by the NSF under Grants No. CCF-1217553, CCF-1453112, and CCF-1438980, AFOSR under Grant No. FA9550-14-1-0148, and DARPA under Grants No. N66001-14-1-4040 and HR0011-13-2-0005. Henry Hoffmann's effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] O. Agmon Ben-Yehuda et al. "The Resource-as-a-service (RaaS) Cloud". In: HotCloud'12. Boston, MA, 2012, pp. 12–12.
[2] Amazon. *"Amazon EC2 Pricing"*. 2014.
[3] *Apache:HTTP server project*. httpd.apache.org. 2014.
[4] S. Bell et al. "TILE64 Processor: A 64-Core SoC with Mesh Interconnect". In: *ISSCC*. 2008.
[5] C. Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.
[6] N. Binkert et al. "The GEM5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7.
[7] R. Bitirgen et al. "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach". In: MICRO 41. 2008, pp. 318–329.
[8] S. Bradley et al. *Applied mathematical programming*. Addison-Wesley Pub. Co., 1977.
[9] D. Burger et al. "Scaling to the end of silicon with EDGE architectures". In: *Computer* 37.7 (2004), pp. 44–55.
[10] L. Cao and H. M. Schwartz. "Analysis of the Kalman filter based estimation algorithm: an orthogonal decomposition approach". In: *Automatica* 40.1 (2004), pp. 5–19.
[11] J. Chen et al. "Modeling Program Resource Demand Using Inherent Program Characteristics". In: SIGMETRICS '11. 2011, pp. 1–12.
[12] R. Coker. *Postal*. doc.coker.com.au/projects/postal. 2009.
[13] Q. Deng et al. "MemScale: Active Low-power Modes for Main Memory". In: ASPLOS XVI. 2011, pp. 225–238.
[14] Q. Deng et al. "CoScale: Coordinating CPU and Memory System DVFS in Server Systems". In: MICRO-45. 2012, pp. 143–154.

[15] C. Dubach et al. “A Predictive Model for Dynamic Microarchitectural Adaptivity Control”. In: MICRO ’43. 2010, pp. 485–496.

[16] S. J. Eggers et al. “Simultaneous multithreading: a platform for next-generation processors”. In: *IEEE Micro* 17.5 (1997), pp. 12–19.

[17] A. Filieri et al. “Automated design of self-adaptive software with control-theoretical formal guarantees”. In: *ICSE*. 2014.

[18] A. Goel et al. “SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit”. In: *2nd USENIX Windows NT Symposium*. 1998.

[19] X. Gu et al. “Application-driven Energy-efficient Architecture Explorations for Big Data”. In: *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*. ASBD ’11. Galveston Island, Texas: ACM, 2011, pp. 34–40.

[20] F. Guo et al. “A Framework for Providing Quality of Service in Chip Multi-Processors”. In: MICRO 40. Washington, DC, USA, 2007, pp. 343–355.

[21] J. L. Hellerstein et al. *FCCS*. John Wiley & Sons, 2004.

[22] H. Hoffmann. “JouleGuard: energy guarantees for approximate applications”. In: *SOSP*. 2015.

[23] H. Hoffmann et al. “Remote Store Programming”. In: *HiPEAC* (2010).

[24] H. Hoffmann et al. “A Generalized Software Framework for Accurate and Efficient Managment of Performance Goals”. In: *EMSOFT*. 2013.

[25] C. Imes et al. “POET: A portable approach to minimizing energy under soft real-time constraints”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. 2015.

[26] E. Ipek et al. “Self-Optimizing Memory Controllers: A Reinforcement Learning Approach”. In: *ISCA ’08*. 2008, pp. 39–50.

[27] E. Ipek et al. “Core Fusion: accommodating software diversity in chip multiprocessors”. In: San Diego, California, USA, 2007, pp. 186–197.

[28] R. Iyer et al. “QoS policies and architecture for cache/memory in CMP platforms”. In: SIGMETRICS ’07. San Diego, California, USA, 2007, pp. 25–36.

[29] B. Jeff. “Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration”. In: *DAC*. 2012, pp. 1143–1146.

[30] D. H. Kim et al. “Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics”. In: *CPSNA*. 2015.

[31] O. Kocberber et al. “Meet the Walkers: Accelerating Index Traversals for In-memory Databases”. In: MICRO-46. 2013, pp. 468–479.

[32] R. Kumar et al. “Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures”. In: *Computer Architecture Letters* 2.1 (2003), p. 2.

[33] B. C. Lee and D. M. Brooks. “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction”. In: *SIGPLAN Not.* 41.11 (Oct. 2006), pp. 185–194.

[34] W. Levine. *The control handbook*. Ed. by W. Levine. CRC Press, 2005.

[35] B. Li and K. Nahrstedt. “A control-based middleware framework for quality-of-service adaptations”. In: *IEEE Journal on Selected Areas in Communications* 17.9 (Sept. 1999).

[36] D. Lo et al. “Towards Energy Proportionality for Large-scale Latency-critical Workloads”. In: ISCA ’14. Minneapolis, Minnesota, USA, 2014, pp. 301–312.

[37] P. Lotfi-Kamran et al. “Scale-out processors”. In: ISCA ’12. 2012, pp. 500–511.

[38] M. Maggio et al. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *IEEE Trans. Contr. Sys. Techn.* 21.1 (2013), pp. 239–246.

[39] D. Meisner et al. “PowerNap: Eliminating Server Idle Power”. In: ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 205–216.

[40] N. Mishra et al. “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints”. In: *ASPLOS*. 2015.

[41] R. Nagarajan et al. “A Design Space Evaluation of Grid Processor Architectures”. In: *MICRO*. Dec. 2001, pp. 40–51.

[42] P. Petrica et al. “Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems”. In: ISCA ’13. 2013, pp. 13–23.

[43] A. Putnam et al. “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”. In: *ISCA 41*. 2014.

[44] M. K. Qureshi and Y. N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: MICRO 39. 2006, pp. 423–432.

[45] S. Ren et al. “Exploiting Processor Heterogeneity in Interactive Services”. In: *ICAC 13*. USENIX, 2013, pp. 45–58.

[46] K. Sankaralingam et al. “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture”. In: *ISCA*. 2003, pp. 422–433.

[47] K. Sankaralingam et al. “Distributed Microarchitectural Protocols in the TRIPS Prototype Processor”. In: *MICRO-39*. 2006, pp. 480 –491.

[48] A. Sharifi et al. “METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management”. In: SIGMETRICS ’11. 2011, pp. 13–24.

[49] D. C. Snowdon et al. “Koala: A Platform for OS-level Power Management”. In: *EuroSys*. 2009.

[50] SPEC. *”CINT2006 (Integer Component of SPEC CPU2006)”*.

[51] D. C. Steere et al. “A Feedback-driven Proportion Allocator for Real-rate Scheduling”. In: OSDI ’99. 1999, pp. 145–158.

[52] G. Suh et al. “A new memory monitoring scheme for memory-aware scheduling and partitioning”. In: *HPCA*. 2002, pp. 117–128.

[53] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1998.

[54] S. Swanson et al. “WaveScalar”. In: MICRO 36. 2003, pp. 291–.

[55] M. Taylor et al. “Scalar operand networks: on-chip interconnect for ILP in partitioned architectures”. In: *HPCA*. 2003, pp. 341 –353.

[56] G. Urdaneta et al. “Wikipedia workload analysis for decentralized hosting”. In: *Computer Networks* 53.11 (2009), pp. 1830 –1845.

[57] Y. Watanabe et al. “WiDGET: Wisconsin decoupled grid execution tiles”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 2–13.

[58] G. Welch and G. Bishop. *An Introduction to the Kalman Filter*. Tech. rep. TR 95-041. UNC Chapel Hill, Department of Computer Science.

[59] D. Wentzlaff et al. “On-Chip Interconnection Architecture of the Tile Processor”. In: *IEEE Micro* 27.5 (Sept. 2007), pp. 15–31.

[60] D. Wentzlaff et al. “Configurable fine-grain protection for multicore processor virtualization”. In: *International Symposium on Computer Architecture (ISCA)*. 2012, pp. 464–475.

[61] H. Yang et al. “Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers”. In: ISCA ’13. 2013, pp. 607–618.

[62] J. J. Yi et al. “A Statistically Rigorous Approach for Improving Simulation Methodology”. In: HPCA ’03. 2003, pp. 281–.

[63] R. Zhang et al. “ControlWare: A Middleware Architecture for Feedback Control of Software Performance.” In: *ICDCS*. 2002, pp. 301–310.

[64] Y. Zhou and D. Wentzlaff. “The Sharing Architecture: Sub-core Configurability for IaaS Clouds”. In: ASPLOS ’14. 2014, pp. 559–574.

[65] Y. Zhou and D. Wentzlaff. “MITTS: Memory Inter-Arrival Time Traffic Shaping”. In: *ISCA*. 2016.