CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores

Neeraj Kulkarni*
Azure Hardware Architecture
Microsoft
Redmond, WA, USA
neeraj.kulkarni@microsoft.com

Gonzalo Gonzalez-Pumariega
Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
gg387@cornell.edu

Amulya Khurana
Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
ak2425@cornell.edu

Christine A. Shoemaker
Industrial and Engineering Management
National University of Singapore
Singapore
shoemaker@nus.edu.sg

Christina Delimitrou
Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
delimitrou@cornell.edu

David H. Albonesi
Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
dha7@cornell.edu

***Abstract*—Multi-tenancy for latency-critical applications leads to resource interference and unpredictable performance. Core reconfiguration opens up more opportunities for application colocation, as it allows the hardware to adjust to the dynamic performance and power needs of a specific mix of co-scheduled services. However, reconfigurability also introduces challenges, as even for a small number of reconfigurable cores, exploring the design space becomes more time- and resource-demanding.**

We present CuttleSys, a runtime for reconfigurable multicores that leverages scalable and lightweight data mining to quickly identify suitable core and cache configurations for a set of co-scheduled applications. The runtime combines collaborative filtering to infer the behavior of each job on every core and cache configuration, with Dynamically Dimensioned Search to efficiently explore the configuration space. We evaluate CuttleSys on multicores with tens of reconfigurable cores and show up to 2.46× and 1.55× performance improvements compared to core-level gating and oracle-like asymmetric multicores respectively, under stringent power constraints.

***Index Terms*—Heterogeneous architectures, datacenter, reconfigurable architectures, resource management**

I. Introduction

Cost efficiency in datacenters is adversely affected by low resource utilization [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Multi-tenancy can boost server utilization, however, co-scheduling jobs is especially challenging for latency-critical applications, such as websearch, social networks, and ML inference, since it can lead to interference in shared resources (cores, cache, memory bandwidth, network bandwidth, power, etc.), and unpredictable performance. Prior work has proposed techniques to avoid interference by disallowing colocation of contending workloads [2], [3], [4], [5], [8], [11], or techniques to eliminate interference altogether, by leveraging hardware and software resource isolation mechanisms [6], [7], [8], [12], [13], [14], [15], [16], [17].

In multi-tenant systems with latency-critical applications, fine-grained resource allocation allows assigning just enough resources to co-scheduled applications to meet their quality of service (QoS) requirements, while improving resource efficiency by allowing more applications to be co-scheduled. However, prior work is limited to traditional servers where cores cannot be reconfigured to enable fine-grained performance and power adjustments. Core reconfiguration [18], [19], [20] opens up more opportunities for colocation, as it allows the hardware to adjust to the dynamic needs of a specific mix of co-scheduled applications.

DVFS, which is widely used in systems today, is another solution to enable fine-grained performance and power adjustments in cores. However, the movement towards processors with razor-thin voltage margins and the increase in leakage power consumption limit the effectiveness of DVFS in future systems [21], [22], [23], [24], [25], [26]. Reconfigurable cores [18], [19], [20] operate by dynamically power gating core components. Since they reduce both active and leakage power, they can be effective in reducing power consumption in technologies where voltage scaling ranges are limited. Datacenters also suffer from poor energy proportionality [6], [23], with processors exhibitng high idle power, as technology shrinks. Reconfigurable cores with their ability to reduce idle power, also offer a solution to make cloud servers more energy proportional.

We propose to leverage reconfigurable cores to enable co-scheduling of cloud latency-critical, interactive services, and batch applications. This means satisfying the strict QoS requirements of the latency-critical interactive services, and maximizing the throughput of the batch applications, while always remaining under the allowed power budget assigned to the server either by the chip-wide power budget, or by a global power manager [6] running datacenter-wide. Prior work on reconfigurable multicores, such as Flicker [18], is exclusively geared towards batch applications, and leads to QoS violations and unpredictable performance for latency-critical services. It additionally does not handle interference in

*Work was done while the author was a PhD student at Cornell University.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00060

the shared memory hierarchy. On the other hand, fine-tuning architectural parameters also increases the space of allocations a resource manager must traverse to identify suitable resource configurations for an application. As the number of cores and configuration parameters increases, efficiently exploring this space becomes computationally prohibitive. This is even more challenging given that decisions must happen online, as applications and power budgets change.

We design CuttleSys, an online resource manager that combines scalable machine learning to determine the performance and power of each application across all possible core and cache reconfigurations, with fast design space exploration to effectively navigate the large configuration space and arrive at a high-performing solution. First, the system leverages collaborative filtering, namely PQ-reconstruction with Stochastic Gradient Descent (SGD), to infer the performance (tail latency for latency-critical and throughput for batch applications) and power consumption of an application across core and cache configurations without the overhead of exhaustive profiling. Second, it leverages a new, parallel Dynamically Dimensioned Search (DDS) algorithm to efficiently find a per-job, globally-beneficial configuration that satisfies QoS for latency-sensitive workloads, and maximizes the throughput for batch jobs, under the power budget. Both techniques keep overheads low, a couple milliseconds, allowing CuttleSys to reevaluate its decisions frequently and adjust to changes in application behavior.

We make the following contributions:

- We demonstrate, for the first time, the potential of reconfigurable cores for servers running latency-critical applications by characterizing five representative interactive cloud services (Section III).
- We present CuttleSys, an online resource manager that efficiently navigates the large design space and determines suitable core and cache configurations (Section IV).
- We evaluate CuttleSys on 32-core simulated systems with mixes of latency-sensitive [27] and batch applications [28]. We show that at near-saturation load and across different power caps, CuttleSys achieves 2.46× higher throughput than core-level gating and 1.55× higher than an oracle-like asymmetric multicore, while always satisfying QoS for the latency-sensitive applications. We also show that CuttleSys effectively adapts to changes in input load and power budgets online (Section VIII).

II. Related Work

A. Power Management

1) Dynamic Voltage-Frequency Scaling (DVFS): DVFS allows dynamically changing a processor's voltage and frequency, and is widely used in modern multicores.

Batch Workloads: Isci *et al.* [29] propose maxBIPS, an algorithm that selects DVFS modes for each core that maximize throughput under a power budget. Sharkey *et al.* [30] extend this work by exploring both DVFS and fetch toggling, as well as design tradeoffs such as local versus global management. Bergamaschi *et al.* [31] further extend maxBIPS, and compare its discrete implementation to continuous power modes. Chen *et al.* [32] propose co-ordinated predictive hill climbing to control distribution of power among cores, and intra-core resources like IQ, ROB and register files among SMT threads. Papadimitriou *et al.* [33] explore safe Vmin for different applications by exposing pessimistic guardbands and determining the best voltage, frequency, and core allocation at runtime.

Apart from open-loop solutions, there are also multiple feedback-based controllers [6], [34], [35], [36], [37]. Wang *et al.* [34] use Model Predictive Control to maintain the power of a CMP below the budget by controlling the DVFS states, while Bartolini *et al.* [36] propose a distributed solution allocating one MPC-based controller to each core. Ma *et al.* [35] propose a hierarchical solution for many-core architectures that divides the problem by allocating frequency budgets to smaller groups of cores. Intel also supports fine-grained power control through the RAPL [38] interface that allows software to set a power limit, which the hardware meets by scaling voltage/frequency.

Latency Sensitive Workloads: Lo *et al.* [6] propose a feedback-based controller that reduces power consumption in server clusters, while meeting the QoS (Quality of Service) requirements of latency-critical services by adjusting the server power limits using RAPL. Nishtala *et al.* [37] use Reinforcement Learning to find the best core allocations and frequency settings for latency-critical jobs to save energy while meeting QoS. Kasture *et al.* [14] propose Rubik, a fine-grained DVFS scheme for latency-sensitive workloads and RubikColoc, a scheme to co-schedule batch and latency-critical workloads. Adrenaline [39] applies DVFS at a per-query granularity, using application-level information to speed up long queries. Meisner *et al.* [23] explore the efficacy of active and idle low-power modes for latency-critical applications to save power under QoS, and showed that active power modes (DVFS) provide good power-performance trade-offs but cannot achieve energy proportionality by themselves. Motivated by their conclusion, our work explores fine-grained power management techniques that reduce idle power along with active power.

The movement towards processors with razor-thin voltage margins limits the effectiveness of DVFS as technology scaling slows down. A viable and widely-implemented alternative to DVFS is core-level gating (C states), discussed in the next section. Reconfigurable cores enable gating at an even finer granularity allowing further gains over traditional core-level gating. Similar to how core-level gating is used along-side DVFS in modern processors, our technique can augment DVFS by increasing the energy gains for frequency regions where DVFS is not effective [20], [21], [22].

2) Core-Level Gating: Core-level gating powers off individual cores by placing them in a separate domain [21], [40], [41], [42], and has become necessary to reduce power consumption beyond DVFS. Intel CPUs since Skylake [21], [22] support Duty Cycling Control (DCC), which cycles between per-core on ($C0$) and off ($C6$) states at the granularity of tens of microseconds. Below we describe several proposals to use core-level gating to maximize performance under a power budget.

Batch Workloads: Intel processors [21], [22] implement core-level gating only during idle core times using auto-demotion. Ma *et al.* [43] and Huazhe *et al.* [44] integrate core-level gating with DVFS, and propose a controller-based algorithm that employs power gating at coarse granularity, and DVFS at fine granularity. Arora *et al.* [45] develop a linear prediction algorithm for *C*6 for CPU-GPU benchmarks. Pothukuchi *et al.* [46] use MIMO theory, while Rahmani *et al.* [47] use Supervisory Control Theory to dynamically tune architectural parameters to meet performance and power goals. These feedback-based controllers become overly expensive as the decision space expands, taking a prohibitive time to converge.
Latency Sensitive Workloads: Leverich *et al.* [24] propose per-core power-gating to dynamically turn cores on/off based on utilization and QoS. PowerNap [25] and DreamWeaver [26] coordinate deep CPU sleep states to minimize idle power. However, Kanev *et al.* [48] show that deep CPU sleep states, owing to their long wakeup latencies, can also impact tail latency, as latency-sensitive applications have short idle periods. We use core-level gating in this work as a baseline for cores that host batch workloads to meet the power budget.

B. Asymmetric Multicores

Asymmetric multicores improve performance and power by assigning resources to applications based on their dynamic requirements [49], [50], [51], [52], [53], [54], [55], [56].
Batch Workloads: PIE [57] schedules applications in heterogeneous multicores by estimating the performance of an application on out-of-order cores, while running on an in-order core and vice-versa. Liu *et al.* [58] propose a dynamic thread-mapping approach, *maximization-then-swapping*, to maximize performance in power-constrained heterogeneous multicores. However, this relies on application profiling, which can become impractical in large-scale multicores.

Teodorescu *et al.* [59] and Winter *et al.* [60] propose thread scheduling and power management for heterogeneous systems. Teodorescu [59] proposes LinOpt, a linear programming-based approach, while [60] explores the Hungarian algorithm to optimize performance under a power budget. Adileh *et al.* [61], [62] maximizes performance by multiplexing applications between two voltage/frequency operating points to match the power budget. The authors propose a technique to shift "power holes" arising due to core heterogeneity. Navada *et al.* [63] propose the use of non-monotic cores, each optimized for different instruction-level behavior, and steer applications on appropriate core types using bottleneck signatures.
Latency Sensitive Workloads: Petrucci *et al.* [64] show that simply using asymmetric multicores without redesigning system software results in QoS violations. They propose a controller that maps jobs to the least power-hungry processing resources that can satisfy QoS by incrementally assigning more slower or faster cores until QoS is met. Ren *et al.* [65], [66] propose a query-level slow-to-fast scheduler, where short queries run on slower cores and longer queries are promoted to faster cores to reduce their service latency. The latter work [66] also theoretically proves the energy efficiency advantages of asymmetric multicores over homogeneous systems. All of these efforts assume that cores of the desired speed are always available, which is not realistic. Haque *et al.* [67] take into account the fact that there is a limited number of cores of each type. They combine asymmetric multicores with DVFS and implement the slow-to-fast scheduler of [65], [66]. However, asymmetric multicores have a fixed number of core types (generally two), while reconfigurable cores provide a finer granularity of heterogeneity, enabling fine-grained performance/power tuning. We compare CuttleSys against an oracle-like asymmetric multicore in Section VIII.

C. Reconfigurable Architectures

Previous work on reconfigurable cores focuses on batch, throughput-bound workloads. Lee *et al.* show the efficiency advantages and limits of adapting microarchitecture parameters to workloads. Lukefahr *et al.* [68] propose Composite cores, which pair big and little compute engines, and save energy by running applications on the small core as much as possible, while still meeting performance requirements. Padmanabha *et al.* [69] propose trace-based phase prediction for migration of applications in Composite cores.

Chrysso [19] proposes an integrated power manager that uses analytical power and performance models and global utility-based power allocation. The configuration space of a core in our work is significantly larger compared to Chrysso [19], which makes the optimization problem more complex. Resource Constrained Scaling (RCS) [70] also aims to maximize performance in power-constrained multicores. In RCS, the resources of a processor and the number of operating cores are scaled simultaneously, which means that the system can operate in only a few different configurations.

Khubaib *et al.* [71] propose a core architecture that dynamically morphs from single-threaded out-of-order to multi-threaded in-order. FlexCore [72] similarly morphs into 4-way or 2-way out-of-order, or 2-way in-order cores at runtime. Tarsa *et al.* [73] propose post-silicon combining of 2 out-of-order execution clusters, and operate as an 8-wide or a low-power 4-wide engine. Duplexity [74] couples SMT master and lender cores and allows dynamic borrowing of threads among them.

The Sharing Architecture [75] and Core Fusion [76] combine multiple simple out-of-order cores to form larger out-of-order cores. CASH [77] also advances the Sharing Architecture with a runtime to find the best configuration for a single application which minimizes cost and meets QoS, using control theory and Q-learning. CuttleSys accounts for the interference between multiple co-scheduled applications that must all meet performance guarantees, and can be applied to the Sharing Architecture to quickly explore the design space of resource slices when multiple applications are hosted on a multi-tenant server, and arrive at suitable per-job resources.

Zhang *et al.* [20] and Petrica *et al.* [18] propose cores that can be reconfigured by scaling datapath components to save energy beyond DVFS. The dynamic scheme in Flicker [18] optimizes performance for a homogeneous multicore with reconfigurable cores under a power budget. Zhang *et al.* [20] also show

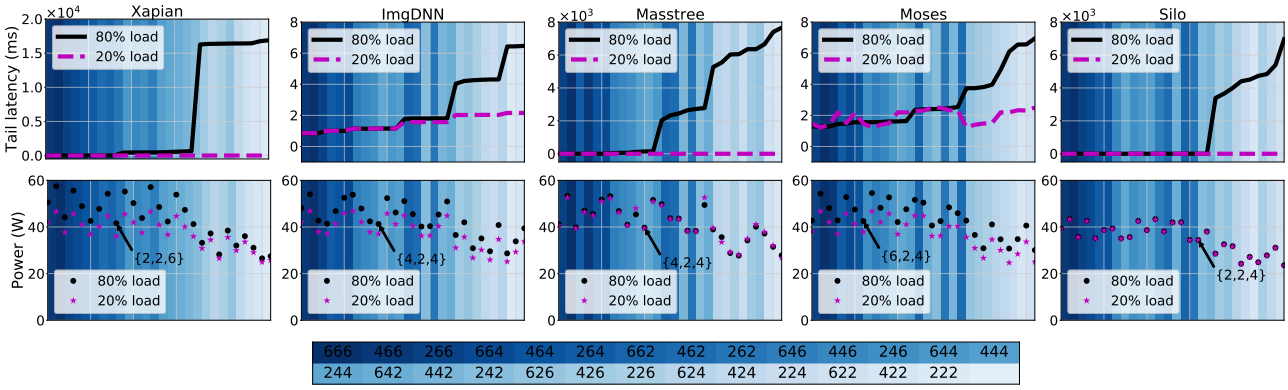


Fig. 1: Characterization of tail latency and power of 5 latency-sensitive applications across core configurations. Colors in the background represent the different core configurations, labeled as {FE,BE,LS}, as shown in the table. Core configurations, from highest to lowest configuration (dark to light color), are ordered by serially decreasing configurations in LS, FE, and BE. For each application, x-axis (core configurations) is sorted according to the tail latency observed at 80% load.

that reconfigurable cores significantly extend the performance-energy pareto frontier provided by DVFS.

However, these systems are limited to batch jobs, and do not consider the implications of reconfiguration on tail latency. Moreover, Zhang *et al.* [20] only consider a single core running one application. In Section VIII-E, we discuss why Flicker cannot be applied directly in this setting, and provide a quantitative comparison between Flicker and CuttleSys.

III. Characterization of Latency-Critical Services

We now quantify the impact of different core configurations on the tail latency of interactive cloud services. We use five applications, `Xapian, Masstree, Imgdnn, Silo, Moses`, and configure them based on the analysis in [27]. We simulate each application on a homogeneous 16-core system using zsim [78], a fast and cycle-level simulator, combined with McPAT v1.3 [79] for a 22nm technology for power statistics. A core is divided into three sections, front-end (FE - fetch, decode, ROB, rename, dispatch), back-end (BE - issue queues, register files, functional), and load-store (LS - LD/ST queues), each of which can be configured to six-way, four-way, and two-way, similar to Flicker [18], except that we adopt a more aggressive superscalar design. These cores dynamically power gate associated array structures in each pipeline region when the configuration is downsized.

Fig. 1 shows the variation of tail latency and power for each service, across core configurations at low and high load. Across all services, at high load, tail latency increases dramatically as the back-end and load-store queue are constrained. On the other hand, at low load, tail latency remains low, even for the lower-performing configurations. Therefore, when load is low, interactive services can leverage reconfiguration to reduce their power consumption, without a performance penalty.

We also observe that the core section that most affects tail latency varies between applications. For Xapian, tail latency is primarily determined by the load-store queue size, with low latency requiring a six-way queue. In the cases of ImgDNN, Silo, and Masstree, tail latencies are low when FE and LS are configured to six- or four-way, while in the case of Moses, tail latency primarily depends on the front-end core section.

At high load, the configuration with the best performance-power trade-off varies across services. For example, `Xapian` consumes the least power in a {2,2,6} configuration while keeping tail latency low, while for `ImgDNN`, `Masstree`, `Moses`, and `Silo`, configurations {4,2,4}, {4,2,4}, {6,2,4} and {2,2,4} consume the least power respectively. This shows that different core configurations are indeed needed by diverse applications. Also, batch applications differ in preferences from latency-critical applications. This variability across loads and applications highlights the need for practical runtimes that identify the best core configurations of each application online.

IV. CuttleSys Overview

We co-schedule latency-sensitive applications with batch workloads on a server with multiple reconfigurable cores, as shown in Figure 2. The last level cache (LLC) and power budget are shared across all cores.

A. Problem Formulation

Our objective is to meet the QoS target for the latency-sensitive application, and maximize the throughput of the co-located batch applications, under a power budget that can change dynamically. Since the applications share the last level cache, the performance of each application depends on the interference in the last level cache caused by other applications. In order to mitigate this interference, CuttleSys also dynamically partitions the LLC among active applications at the granulariy of cache ways [80], [81].

The system consists of N cores. Each core can be configured in m modes. Each application can be assigned one of p cache way allocations. Thus, each application can be executed in $m*p$ configurations. For simplicity, the formulation below assumes one latency-sensitive application colocated with multiple (B) batch applications. The objective function is as follows:

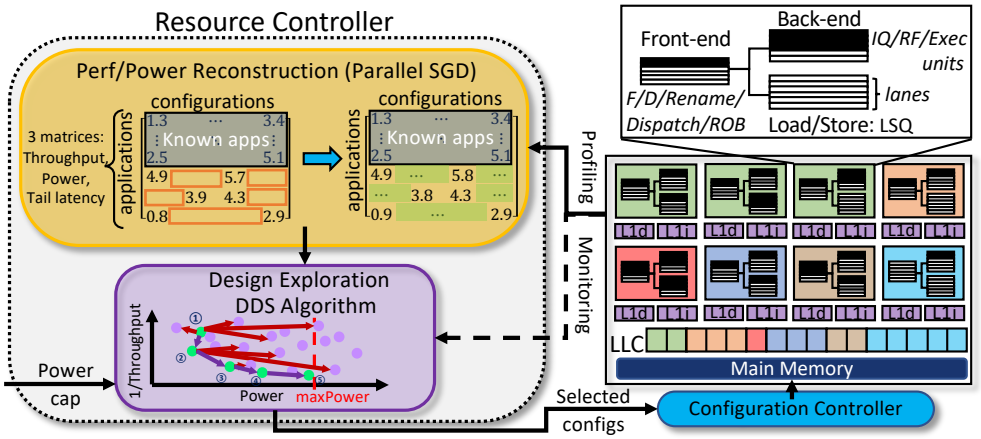


Fig. 2: CuttleSys system overview.

$B_{i,j,k}$ = throughput (BIPS) of batch app i running in core config j and cache allocation k

$T_{0,j,k}$ = tail latency of latency-sensitive app running in core config j and cache allocation k

$P_{i,j}$ = power of app i running in core config j

$C_{i,j,k}$ = cache ways allocated to app i running in core config j and cache allocation k

$I_{i,j,k} = 1$ if app i is assigned to core configuration j and cache allocation k
$= 0$ otherwise

We maximize the geometric mean of throughput:

$$BIPS_{system} = (\prod_{i=1}^{B} \sum_{j,k} B_{i,j,k} * I_{i,j,k})^{1/B} \quad (1)$$

under the following constraints:

$$Power_{system} = \sum_{i=0}^{B} \sum_{j,k} P_{i,j} * I_{i,j,k} \leq maxPower \quad (2)$$

$$Cache\_alloc_{system} = \sum_{i=0}^{B} \sum_{j,k} C_{i,j,k} * I_{i,j,k} \leq cacheWays \quad (3)$$

$$\sum_{j,k} T_{0,j,k} * I_{0,j,k} \leq QoS \quad (4)$$

$$\sum_{j,k} I_{i,j,k} = 1 \; \forall i = 1,..N \quad (5)$$

Eq. 2 states that the total power should be under the budget, while Eq. 3 states that the total allocated cache ways should be no higher than the LLC associativity. We exclude the power overhead of data movement from DRAM, since it is negligible compared to the core power. Eq. 4 addresses the QoS requirement of the latency-sensitive application. Eq. 5 states that each application can be mapped to a single configuration. We use geometric mean as the objective function, since all batch applications have equal priority [82]. Exhaustively exploring the full design space of core configurations and cache allocations ($(m*p)*(m*p)^B$) is impractical as the number of cores/applications increases. This is problematic, since reconfiguration decisions need to happen online, and the optimization problem is non-linear and non-convex in nature.

Our scheme is made practical via two separate, mutually beneficial optimizations:

1) Lightweight runtime characterization to infer the performance ($B_{i,j,k}$ in Eq. 1, $T_{0,j,k}$ in Eq.4), and power ($P_{i,j}$ in Eq. 2), of all applications across all possible m core configurations and p cache allocations; and
2) Fast and accurate design space exploration, given the output from (1) to determine a globally-beneficial solution to the core configuration and cache allocation problem in the limited time available for scheduling.

Previous approaches [18] to determine the impact of reconfiguration require detailed profiling of each active application against large number of resource configurations, which incurs non-trivial profiling overheads, and scales poorly with the number of configuration parameters. This approach is furthermore limited to batch applications, and does not take into account inter-application interference. Instead, we propose to infer performance (tail latency for interactive services and throughput for batch jobs) and power, across all possible core and cache configurations, by uncovering the similarities between the behavior of new and previously-seen applications across configurations. Specifically, we use PQ-reconstruction with Stochastic Gradient Descent [2], [83], [84], [85], a fast and accurate data mining technique that, given a few profiling samples for an application collected at runtime, estimates the application's performance and power across all remaining system configurations, based on how previously-seen, similar applications behaved on them. While SGD has been previously applied in the context of cluster scheduling [2], [3], core reconfiguration places much stricter timing constraints (few ms) and a larger configuration space on SGD, requiring a new, more efficient, parallel approximated SGD implementation.

To quickly explore the design space, we adapt Dynamically Dimensioned Search (DDS) [86], a heuristic algorithm that searches high-dimensional spaces for globally-beneficial solutions. DDS is computationally efficient, applicable to discrete

problems, and especially effective for problems with high dimensionality, such as quickly searching the large space of resource configurations. The combination of SGD and DDS significantly improves performance over previous approaches.

We also note that CuttleSys is an open-loop solution, which searches the design space and finds the best resource allocation in a single decision interval compared to feedback-based controllers, which take significant time to converge. This is especially beneficial for latency-critical applications, as they do not suffer from QoS violations until convergence.

B. Efficient Resource Management

Fig. 2 shows the high-level architecture of CuttleSys, which consists of the *Configuration Controller* and the *Resource Controller*. At the beginning of each decision quantum (100ms by default, consistent with prior work [18]), the *Configuration Controller* profiles performance and power, which are used by the *Perf/Power Reconstruction* module in the *Resource Controller*. The *Configuration Controller* then configures cores and cache ways based on the solution from the *Design Exploration* module for the remainder of the timeslice.

The *Resource Controller* takes as input the collected profiling samples, and the specified Power Cap, and determines the best core/cache configurations. The first step is *Perf/Power Reconstruction*, which uses SGD to estimate the power and performance of an application for all core and cache configurations, based on a small number of samples (Section V). The *Design Exploration* uses SGD's output to determine the best configuration for each job (Section VI).

We describe the timeline of this process below, using Fig. 3. Our approach requires 2 profiling samples, one sample of the highest- and one of the lowest-performing configurations, corresponding to the widest-issue ({6,6,6}) and narrowest-issue ({2,2,2}) configurations respectively with one LLC way per core for the currently running applications, along with the performance and power of some "training" applications in all configurations, as shown in Figure 2. We run applications for the duration of a sample timeframe ($1ms$ as described in Section VIII-A1), for each configuration and measure performance and power (①). QoS for most cloud services is measured at intervals longer than $1ms$ [3], [6], [7], [23], [87], [88]. To obtain meaningful measurements, we measure tail latency over the entire $100ms$ of the previous timeslices. After this online profiling, we run the reconstruction algorithm to estimate the tail latency of latency-sensitive cloud services, the throughput of batch applications, and the power consumption of each application across all $m*p$ configurations (②).

Finally, we apply DDS to quickly search the space of core configurations and cache allocations, and find a solution that meets QoS and maximizes the throughput of batch applications for the given power budget (③). The system then runs in steady state (④) with the selected core and LLC configurations. At the end of the timeslice, power and performance are measured and updated in the SGD matrix to ensure that any predictions deviating from the real metrics are corrected.

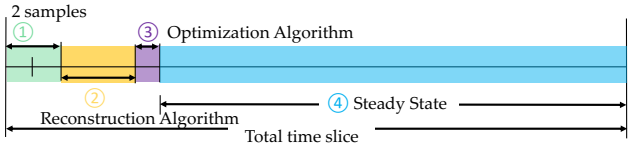


Fig. 3: Timeline showing the steps of characterization, inference, and steady-state operation in CuttleSys.

V. Practical Inference with SGD

The first step in the *Resource Controller* estimates the power, throughput, and tail latency for applications across all core configurations and cache allocations. Previous techniques [18] require long profiling runs to accurately estimate an application's power and performance across configurations. Moreover, since previous work only targeted core configurations, estimating performance for cache allocations too would require an untenable number of profiling samples. Instead, we use the following insight to reduce profiling and improve practicality: the performance and power profile of a new, potentially unknown application may exhibit similarities with the characteristics of applications the system has previously seen, even if the exact applications are not the same.

This problem is analogous to a recommender system [83], [89], [90], [91], [92], [93], [94], where the system recommends items to users based only on sparse information about their preferences. In our case, users are analogous to applications and items to resource configurations (core configurations and cache allocations). A rating corresponds to the power or performance of an application running in the particular core and cache configuration. We construct a sparse matrix R (one each for throughput, tail latency and power) with applications as rows and resource configurations (core-cache vectors) as columns. The rows of matrix R include some "known" applications, along the previously-unseen applications that arrive to the system.

The matrix is initially populated with the performance or power of these "known" applications which have been characterized once offline across all configurations. For all other new applications, the corresponding rows only have two entries obtained through profiling on two core-cache configurations out of the entire design space. The missing entries are inferred using PQ-reconstruction with Stochastic Gradient Descent (SGD) [2], [84], [85], [89], [90]. To reconstruct R, we first decompose it to matrices P and Q, where the product of Q and P^T gives the reconstructed R, as shown in Alg. 1. Matrices Q and P are then constructed using Singular Value Decomposition (SVD), and correspond to $Q = U$ and $P^T = \sum \cdot V^T$ respectively, where U, V are the left and right matrices of singular vectors, and $\sum$ the diagonal matrix of singular values. In Alg. 1, A is the total number of jobs (including known ones), and $m*p$ is the number of resource configurations. The impact of training set size is discussed in Sec. VIII-A2.

There is an obvious trade-off between the maximum number of iterations and the reconstruction accuracy: the fewer the iterations, the lower the overhead, but also the higher the prediction inaccuracy. We have conducted a sensitivity study

Algorithm 1 Reconstruction Algorithm

Initialization:
  $\mathbf{Q} \leftarrow$ random(A,m*p); $\mathbf{P} \leftarrow$ random(m*p,m*p)
  $\eta \leftarrow$ learning rate; $\lambda \leftarrow$ regularization factor
  $maxIter \leftarrow$ max # of iterations
for $l \leftarrow 1$ to $maxIter$ **do**
  for $i \leftarrow 1$ to A **do**
    for $j \leftarrow 1$ to $m*p$ **do**
      $\varepsilon_{ij} \leftarrow R_{ij} - \mathbf{Q_j}.\mathbf{P_i^T}$
      $\mathbf{Q_j} \leftarrow \mathbf{Q_j} + \eta(\varepsilon_{ij}\mathbf{P_i} - \lambda\mathbf{Q_j})$
      $\mathbf{P_i} \leftarrow \mathbf{P_i} + \eta(\varepsilon_{ij}\mathbf{Q_j} - \lambda\mathbf{P_i})$
$R \leftarrow Q \times P^T$

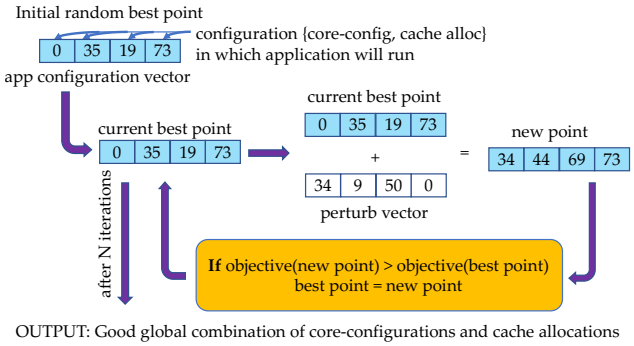


Fig. 4: The DDS design space exploration algorithm.

to select convergence thresholds for SGD. To further reduce overheads, we have also limited the number of iterations.

For the currently-running applications, we obtain two samples of the highest- and lowest-performing core configurations with the ways equally allocated at runtime. We also get additional samples for these applications by monitoring power, throughput, and tail latency for the configurations from previous steady states. To predict the throughput and power for the remaining configurations ($m*p-2$, initially but fewer as we get more points from previous steady states) and tail latency for the remaining configurations ($m*p-1$ initially), we run three instances of the reconstruction algorithm, one each for throughput, tail latency, and power. We run these three reconstructions in parallel to minimize overheads.

To further accelerate reconstruction, we have implemented a parallel reconstruction algorithm that executes SGD without synchronization primitives [95], [96]. This introduces a small, upper-bounded inaccuracy (approximately 1%), while improving its execution time by 3.5×.

VI. Fast Design Exploration with DDS

Once SGD recovers the missing performance and power of each job across all core configurations and cache allocations, the system employs Dynamically Dimensioned Search (DDS) to quickly explore the space, and select appropriate core configurations and cache partitions. DDS [86] is specifically designed to navigate spaces with high dimensionality, especially in cases where computing the objective function is expensive. This makes it a good fit for CuttleSys's tight timing constraints.

The operation of DDS is shown in Fig. 4. The algorithm explores new points in the design space by perturbing a small number of dimensions from the current best point in each iteration, with the number of perturbed dimensions decreasing as the search progresses, and eventually converging to a globally-beneficial solution. Fig. 4 shows an example of DDS for a simple 4-core system running four applications on four cores. The application configuration vector is a N-dimensioned decision variable, where the i^{th} dimension denotes the configuration assigned to the i^{th} application. The configuration assigned can be any number from 0 to $m*p-1$. The algorithm starts with a set of random points, and selects the point that has the highest value for the target objective as the current best point. In the given example, the current best point has threads 0, 1, 2 and 3 assigned to configurations 0, 35, 19, and 73 respectively. The current best point is then perturbed to explore new points. If the new point has a higher objective, it replaces the previous best point, and the process repeats until the algorithm arrives at a globally-beneficial combination of core configurations and cache allocations. The perturbation vector determines the number of dimensions to be perturbed and the perturbation magnitude for each dimension. DDS searches across more dimensions in the beginning, and narrows down to fewer dimensions later. The perturbation quantity is equal to $r \cdot (\#confs) \cdot \mathcal{N}(0,1)$, where r is a perturbation parameter.

A. Handling Optimization Constraints

The optimization problem of Sec. IV has three constraints: a) power (Eq. 2), b) cache (Eq. 3), and c) QoS (Eq. 4).

Since latency-critical jobs are load-balanced, all cores assigned to them use the same configuration. This simplifies the core configuration search to scanning through the predicted tail latency of the $m*p$ configurations. We select the lowest cache allocation, and the core configuration that consumes the least power while meeting QoS. DDS then explores points for the batch jobs, while keeping the configuration of cores and cache ways assigned to latency-critical applications fixed.

To handle the power and cache constraints of Eq. 2 and 3, we use an objective function that penalizes the points that consume more power and/or more cache than allowed as follows:

$$objective(\mathbf{x}) = BIPS_{system}(\mathbf{x}) - penalty\_power * (maxPower - Power_{system}(\mathbf{x})) - penalty\_cache * (maxWays - Cache\_alloc_{system}(\mathbf{x}))$$

We choose a soft penalty approach to handle the power constraint in the objective function, so that points with slightly higher power are not heavily penalized.

If no configurations are found which meet the QoS of the latency-critical service, CuttleSys reclaims cores from the batch workloads, one per timeslice, and yields them to the latency-critical service, until QoS is met. The cores are similarly incrementally relinquished by the latency-critical applications when QoS is met with latency slack.

B. Parallel DDS

To further speed up the design space exploration, we have designed a new parallel DDS, shown in Alg. 2.

Algorithm 2 Parallel DDS Algorithm

```
1:  Initialization:
2:      maxIter ← max # of iterations
3:      r ← perturbation parameter
4:  l_c = get_config_LC()
5:  Initial rand points x={l_c,..,l_c,x_K,...,x_N}
6:  x^best ← argmax{obj(x)|x ∈ random points}
7:  for i ←1 to maxIter do
8:      x^localbest = x^best
9:      for j ← 1 to pointsPerIteration do
10:         p ← 1 − log(i)/log(maxIter)
11:         add dimensions to {P} with probability p
12:         for d ∈ {P} do
13:             x^new[d] = x^localbest[d] + r·(#confs)·N(0,1)
14:             if x^new[d] ∉ [0,#confs) then
15:                 reflect the perturbation
16:         if obj(x^new) > obj(x^localbest) then
17:             x^localbest = x^new
18:     barrier_wait()
19:     if threadID == 0 then
20:         x^best ← argmax{obj(x)|x ∈ {x^localbest}}
21:     barrier_wait()
```

In the first phase, we initialize the algorithm's parameters. Line 2 sets the maximum number of iterations (*maxIter*) of the algorithm. As *maxIter* increases, the quality of the solution obtained improves, but at the same time the time required to run the algorithm also increases. We explore this trade-off in Section VIII, and select the appropriate number of iterations.

In parallel DDS, to avoid different threads exploring the same points (obtained from perturbation of the same best point), and to explore a larger space of configurations, we use four different values for the perturbation parameter; **r** = (r1, r2, r3, r4). In an N-core system, the first $N/4$ threads of the parallel algorithm set $r = r1$, the next $N/4$ threads set $r = r2$, etc.

Line 4 gets the resource configuration that satisfies the QoS for latency-critical (LC) applications. Lines 5-6 show the randomly-chosen points the algorithm starts with, selecting the best among them as the initial best point. In parallel DDS, for a current best point, each thread generates *pointsPerIteration* number of new points, and finds the best point among them, as shown in Lines 9-17. The number of dimensions to be perturbed is determined by the probability function, as seen on Lines 10-11, while Line 13 shows the quantity by which the dimensions are perturbed. If the value of a dimension in the newly-generated point is out of bounds, the algorithm mirrors the value about the maximum or minimum bound, to bring the point back within the valid range (Lines 14-15).

DDS chooses the new point as the next best point if $obj(\mathbf{x}^{new}) > obj(\mathbf{x}^{best})$ (Lines 16-17). After each core has computed *points PerIteration* points, a single core aggregates all the per-core best points, picks the best one, and uses the selected configuration in the next iteration (Lines 18-21). DDS concludes after *maxIter* iterations, and returns the best combination of core configurations and LLC allocations.

If the power cap is not met even when all cores running batch jobs are in the lowest configuration, we turn off cores, in descending order of power, until the power budget is met.

VII. Experimental Methodology

We simulate 32-core multicores with reconfigurable cores. The core's architectural parameters are shown in Table I, and are scaled according to the selected configuration, similar to [18]. Since we assume six-, four-, and two-way in each of the front-end, back-end, and load/store queue sections, we have a total of $3^3 = 27$ (m=27) configurations. Our cores are also similar to the large cores in AnyCore [97], which evaluates the performance-energy overheads of reconfiguration.

Front end	BP: gshare + bimodal, 64 entry RAS, 4KB BTB 144 entry ROB 6-wide fetch/decode/rename/retire
Execution core	out-of-order, 6-wide issue/execute 192 integer registers, 144 FP registers 48 entry IQueue, Load Queue, Store Queue 6 Integer ALUs, 2 FP ALU 1 Int/FP Mult Unit, 1 Int/FP Div Unit
Memory heirarchy	L1 I-Cache: 32KB, 2-way, 2 cycles L1 D-Cache: 64KB, 2-way, 2 cycles L2 Cache: 64MB, shared, 32-way, 20 cycles 200 cycle DRAM access latency
Technology	22 nm technology, 0.8V Vdd, 4GHz frequency

TABLE I: Configuration of the 32-core simulated system.

Based on the RTL analysis of frequency, energy, area overheads in [97], we assume 1.67% frequency and 18% energy penalty per cycle for our reconfigurable cores compared to fixed ones. Reconfigurable cores also consume 19% higher area. In our experiments, we consider fixed-power scenarios, where the power budget is kept constant across the designs (core gating of symmetric and asymmetric multicore, and reconfigurable cores). Under the power-capped scenarios, even if more cores can be packed in fixed-core designs (core gating-based and asymmetric multicores), they cannot be turned on due to power constraints. The performance benefits of CuttleSys are achieved at the cost of 19% more area.

A. Simulation Infrastructure and Workloads

We use zsim [78] for performance statistics combined with McPAT v1.3 [79] in 22nm technology for power statistics. We simulate 32-core systems, with 50% cores assigned to a latency-critical (LC) application and 50% cores are assigned to batch jobs at time $t=0$. The core allocation can change at runtime. Batch jobs are multi-programmed mixes from SPECCPU2006 (`perlbench`, `bzip2`, `gcc`, `mcf`, `cactusADM`, `namd`, `soplex`, `hmmer`, `libquantum`, `lbm`, `bwaves`, `zeusmp`, `leslie3d`, `milc`, `h264ref`, `sjeng`, `GemsFDTD`, `omnetpp`, `xalancbmk`, `sphinx3`, `astar`, `gromacs`, `gamess`, `gobmk`, `povray`, `specrand`, `calculix`, `wrf`), while

the LC services are selected from TailBench [27] (`Xapian, Masstree, ImgDNN, Moses, Silo`). We co-schedule each of the TailBench services with 10 multiprogrammed (16-app) mixes from SPECCPU2006, for a total of 50 mixes. We use one LC service for simplicity, however, CuttleSys is generalizable to any number of LC and batch services, as long as the system is not oversubscribed.

The reconstruction algorithm requires the power and performance of a small number of representative applications to be collected offline, on all core configurations and cache allocations. We randomly selected 16 (discussed in Section VIII-A2) of the above SPECCPU2006 applications for offline training at the beginning, excluding significant platform redesigns. Each of the multiprogrammed workloads is constructed by randomly selecting one of the remaining SPECCPU2006 benchmarks to run on each core, to ensure no overlap between the training and testing datasets. Each SPECCPU2006 benchmark runs with the reference input dataset.

To find the maximum load each Tailbench service can sustain, we simulate it on a 16-core system and incrementally increase the queries per second (QPS), until we observe saturation. We use the QPS at the knee-point before saturation as the maximum load to avoid the instability of saturation [8]. These max QPS are: a) `Xapian`: 22kQPS, b) `Masstree`: 17kQPS, c) `ImgDNN`: 8kQPS, d) `Moses`: 8kQPS, and e) `Silo`: 24kQPS.

The system's maximum power is the average per-core power across all jobs on reconfigurable cores scaled to 32 cores. We evaluate the system across power caps.

B. Baseline Core-Level Gating

We compare our design with core-level gating as it is widely employed in current systems for power gating. To meet QoS the cores running latency-sensitive applications are always turned on. To determine which cores to turn off, core gating requires estimations of the power and performance of all applications. To do this, we profile the applications for one *sample_time*. We explore the following approaches for selecting the cores to turn off: a) descending order of power; b) ascending order of power; c) ascending order of BIPSperWatt; and d) ascending order of BIPS. From our experiments, we found that turning off cores based on descending order of power achieves the best performance for core-level gating. When turning off the last core required to meet the power budget, we search among the active cores and gate the one that meets the power budget with the smallest slack. We also consider core-gating with LLC way-partitioning using [80], since the technique is already available in real cloud servers [7]; the choice of cache partitioning is orthogonal to the techniques in CuttleSys.

Quantitatively comparing against core-level gating using the geometric mean of throughput is problematic, since when a core is gated, fewer applications run to completion. Thus, we compare the total number of instructions (useful work) executed over the same amount of time.

C. Asymmetric Multicores

Asymmetric multicores, which comprise cores with different performance and energy characteristics, have been proposed as an alternative to homogeneous multicores in order to improve energy efficiency [49], [50], [51], [52], [53], [54], [55]. Heterogeneity allows each application to receive resources that are suitable to its requirements and thus, improve the overall throughput, while still operating under a power budget. In asymmetric multicores, each type of core (typically a high-end and a low-power core type [98]), and the number of cores of each type are statically designed. In contrast, reconfigurable multicores enable finer configuration granularity by providing a higher number of core types. Furthermore the number of cores in each configuration can be decided at runtime.

We compare CuttleSys with a heterogeneous system with two types of cores: big cores, equivalent to the {6,6,6} configuration, and small cores, equivalent to the {2,2,2} configuration. While typically the number of cores are statically fixed, we compare against an oracle-like system, which selects the best number of big and small cores that meets the QoS of latency-critical applications, and maximizes the throughput of batch applications under a given power budget. For the oracle system, we also ignore any scheduling overheads that the threads incur to migrate between cores of different types.

VIII. Evaluation

A. CuttleSys Scheduling Overheads

CuttleSys incurs three types of overheads: (i) for the initial *application profiling* that gives the controller a sparse signal of the application's characteristics, (ii) for the *reconstruction algorithm* that infers performance and power on all non-profiled configurations, and (iii) for the DDS space exploration (Fig. 3). Table II shows these overheads.

Performance/Power sampling		SGD reconstruction	DDS search
Single run	Total time	4.8 ms	1.3 ms
1 ms	2 ms		

TABLE II: Characterization and optimization overheads.

1) Profiling: We empirically set a monitoring period of 1*ms* as a advantageous trade-off between reducing profiling overheads and increasing decision accuracy, similar to [18]. We profile all cores in parallel for 2*ms* (① of Fig. 3), 1*ms* each in the widest-issue $\{6,6,6\}$ and narrowest-issue $\{2,2,2\}$ configurations with one way of LLC allocated to each core, and measure performance and power consumption. To avoid power overshoot by running all cores in the highest configuration, half of the cores run in the widest-issue configuration, and the other half in the narrowest-issue configuration in the first 1*ms* and vice-versa in the second 1*ms*. Note that even core-level gating incurs an overhead of 1*ms* for one profiling period.

2) Reconstruction Algorithm: Reconstruction requires characterizing offline a few "known" applications. We select the fewest jobs (16) needed to keep accuracy over 90% for all running jobs. If the training set included 24 jobs instead, inaccuracy drops to 8%, while execution time for reconstruction increases by 18%. On the other hand, decreasing the training set to 8 applications increases inaccuracy to 20%.

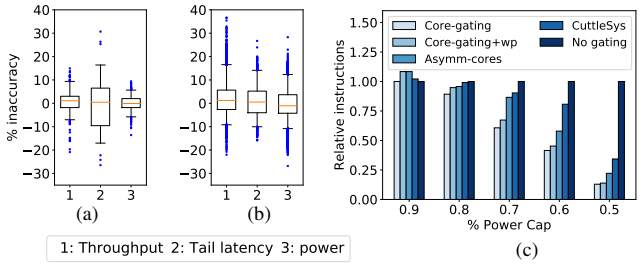


Fig. 5: Box plots of the error between the measured and predicted performance and power by SGD across configurations (a) in isolation and (b) with colocation. (c) Instructions with CuttleSys vs. core-level gating over 1s across power caps.

We run three instances of the reconstruction algorithm (one each for throughput of batch jobs, tail latency of LC applications, and power for all jobs). Reconstructing the throughput for batch jobs takes longer, as it needs to find the missing values for all combinations of core and LLC configurations for 16 applications, while reconstructing the tail latency needs to estimate the missing values for all configurations of 1 job at a time. Inferring performance and power for all possible LLC allocations (32 in our case) increases the overhead and impacts accuracy, even though many allocations would not be feasible in practice, as all 32 cores need to share the 32 ways. Therefore, we limit the LLC allocations for each job to 1/2, 1, 2, and 4 ways. If two jobs are allocated 1/2 ways each, both are assigned the same LLC way. Any interference between them is handled by updating the reconstruction matrix with the measured values during runtime. The three reconstructions all run in parallel on the same server.

3) DDS Algorithm: As described in Section VI, the $\#confs$ is set to 107, since we consider four LLC allocations for each core configuration. We have performed sensitivity studies to find the parameters of parallel DDS that achieve the best trade-off between runtime and accuracy. We arrived at the parameter values shown in Figure 6.

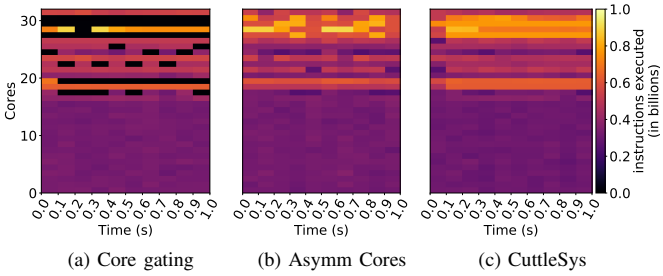
initial random points	50
r = [r1,r2,r3,r4]	[0.2,0.3,0.4,0.5]
penalty_wt	2
pointPerIteration	10
maxIter	40

Fig. 6: DDS parameters.

B. CuttleSys Inference Accuracy

CuttleSys uses three instances of the parallel SGD algorithm to reconstruct the throughput, tail latency, and power of co-scheduled applications across resource configurations.

To isolate the prediction accuracy of SGD, we run all test applications in isolation for the full time slice in all core configurations, which avoids both interference from co-scheduled jobs and inaccuracies from limited profiling time. For the throughput, power, and tail latency estimation, we profile on two configurations per job, and infer the remaining 106 entries. Fig. 5(a) shows the estimation errors for throughput, tail latency, and power across the 12 "testing" SPEC applications and 5 Tailbench applications at 80% load. Fig. 1 shows that some configurations incur very high tail latency, and are not selected during runtime. For these configurations, exact latency prediction is less critical, as long as the prediction shows that QoS is violated. We observe that the 25^{th} and 75^{th} percentiles are within 10%, while the 5^{th} and 95^{th} percentiles are less than 20% for throughput, tail latency, and power. The error for tail latency is higher, as we predict services one at a time and only use 2 sample runs to predict the remaining 106 configurations.



(a) Core gating (b) Asymm Cores (c) CuttleSys

Fig. 7: Instructions executed in each time slice (0.1s) on all cores with core-level gating, asymmetric cores, and CuttleSys.

We now examine the inaccuracy at runtime, which also includes application interference and inaccuracies due to limited profiling. Fig. 5(b) shows box plots of errors in throughput, tail latency, and power. The median is close to zero and the 25^{th} and 75^{th} percentiles are within 10% in all cases. However, the 5^{th} and 95^{th} percentiles for tail latency increase, as do the outliers for throughput. This is due to (a) applications changing execution phases, making the profiling runs not representative of steady state behavior, and (b) contention between jobs. Since CuttleSys updates the reconstruction matrix with the measured metrics, it accounts for changes at runtime.

C. Core Gating and Asymmetric Multicores

Fig. 7 shows the number of instructions executed on all cores in each timeslice over 1s with core-level gating and CuttleSys under a 70% power cap. In the case of core-level gating, cores that consume the most power are turned off to meet the power budget and do not execute any instructions. In the case of asymmetric multicores, though all cores remain active, some jobs execute on small cores. We assume an unrealistic, oracle-like asymmetric multicore, where the number of big and small cores is determined to be the optimal, for a given workload, in each timeslice. To meet QoS, the latency-sensitive applications usually execute on big cores. For 70% power cap, an additional 7 out of 16 batch applications execute on the big cores, while the remaining 9 applications execute on the small cores. CuttleSys also keeps all cores active, but portions of the cores might be turned off to meet the power budget.

Fig. 5(c) quantitatively compares the total number of instructions executed by batch applications in (1) core-level gating without way-partitioning; (2) core-level gating with way-partitioning; (3) the oracle-like asymmetric multicore; and (4) CuttleSys, relative to no gating (all cores run in highest configuration) with no cache partitioning, for each power cap. QoS is satisfied for all Tailbench applications across all runs for core-level gating, oracle-like asymmetric multicore, and CuttleSys. Results include all overheads of Sec. VIII-A.

For relaxed power caps (90%), all cores can be turned on for the fixed-core multicores (core-level gating and asymmetric multicores), while parts of the cores need to be turned off with Cuttlesys, given the energy overhead of reconfiguration. Thus, CuttleSys performs worse in this case.

As the power caps decrease, however, CuttleSys outperforms core-level gating both without and with way-partitioning by 1.64× and 1.52× on average, and up to 2.65× and 2.46× respectively (Fig. 5(c)). CuttleSys also outperforms the oracle-like asymmetric multicore by 1.19× on average, and up to 1.55× for the most stringent power cap. As power caps decrease, core-level gating turns off additional cores, while the oracle-like multicore executes more jobs on smaller cores. The fine granularity of reconfigurable cores provides additional power/performance operating points, which permit better fine-tuning during power-constrained scenarios. These gains amortize the energy and scheduling overheads of CuttleSys.

CuttleSys provides modest throughput gains over the oracle-like asymmetric multicore for relaxed power caps, as more batch jobs can execute on big cores in the asymmetric multicore. In real systems [98], the number of small and big cores is fixed. CuttleSys outperforms a typical multicore with 50% big and 50% small cores by 1.70×, 1.65× and 1.50× at 90%, 80% and 70% power caps respectively. The performance of this 50-50 multicore is the same as that of the oracle-like asymmetric system at 60% and 50% power cap, since all the batch applications run on small cores.

D. Dynamic behavior of CuttleSys

We now show CuttleSys's behavior under varying load and power caps, and an example of core relocation.

1) Varying Load: We vary the input load of the latency-critical application by simulating a diurnal pattern, while maintaining the power budget at 70% of max. Fig. 8a shows the input load of the latency-critical application, its tail latency with respect to QoS, the throughput of batch applications, the total power consumed by the system, and the core configurations for batch applications for a colocation of Xapian with a mix of 16 SPEC jobs. When load is low, cores running Xapian are configured to {4,2,4}, as shown by the background color.

As load increases, the tail latency also increases and violates QoS. Subsequently, CuttleSys configures the cores allocated to Xapian to the {6,6,6} configuration in the next time slice, after which QoS is met, and to {6,2,6} in the following time slice. Four cache ways are allocated to Xapian throughout the experiment. Under high load, Xapian consumes a significant fraction of the power budget, leaving less power for the SPEC applications. The cores running SPEC jobs therefore have to run in lower-performing configurations, and as a result achieve lower throughput. There is a brief interval in $t \in [0.3, 0.4]s$ where the system violates its power budget. This is because the input load of Xapian increases in the middle of CuttleSys's decision interval, and the system needs to wait until the next interval before reconfiguring the cores. While this may briefly consume more power than required, it avoids ping-ponging between configurations due to short load spikes. When the load decreases, CuttleSys again reconfigures Xapian's cores to {4,2,4}, and set the remaining cores to higher configurations, thus increasing the throughput of SPEC jobs.

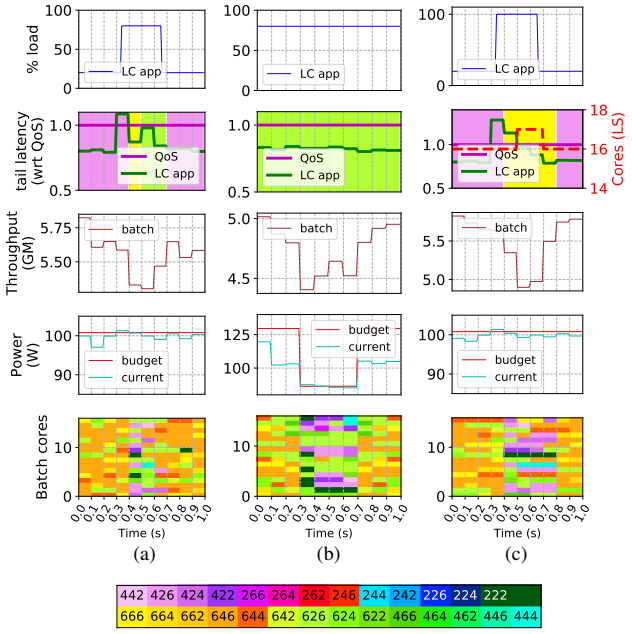


Fig. 8: CuttleSys under (a) varying input load, (b) varying power budget, and (c) example of core relocation. The table shows the colors corresponding to core configurations.

2) Varying Power Budget: We now vary the power cap over time when running Xapian and a mix of SPEC applications, while maintaining a constant 80% load for the latency-critical application. The power budget is set to 90% and reduced to 60% at $t = 0.3s$. In this case (Fig. 8b), the cores running Xapian are configured to {6,2,6} and four cache ways for the entire duration of the experiment. When the power cap is reduced, Xapian still needs the same amount of power to meet its QoS, leaving a lower power budget for the SPEC workloads, which are configured to lower-performing configurations, decreasing their throughput. When the power cap is set back to 90% at $t = 0.7s$, the SPEC cores revert back to the higher configurations.

3) Core Relocation: Fig. 8c demonstrates an example co-scheduling Xapian with a mix of SPEC applications, where CuttleSys relocates cores to the latency-critical application to meet its QoS. As the load increases after $t = 0.3s$, Xapian suffers a QoS violation, after which its allocated cores are reconfigured from {4,2,4} to the widest-issue configuration {6,6,6}. However, that is not sufficient to meet QoS in this case. Thus, CuttleSys reclaims a core from the batch applications, and assigns it to Xapian, at which point QoS is met. After the load drops back down to 20%, tail latency also drops. Since now the latency slack is high enough (20% unless otherwise specified), the extra core is yielded back to the batch applications. As a result of the core relocation, the SPEC jobs time-multiplex on the reduced number of cores allocated to them, achieving lower throughput, which is recovered when the core is returned.

E. Comparison with Flicker

We compare CuttleSys with Flicker [18], which is the most relevant prior work and state-of-the-art for reconfigurable multicores. Flicker was designed for multicore architectures running multi-programmed mixes of exclusively batch jobs. It proposed 3MM3 sampling [99] with RBF surrogate fitting [100], [101], [102], [103], [104] to characterize the impact of core configurations, and Genetic Algorithm (GA) for space exploration. Flicker relies on detailed per-configuration profiling, and is limited to core configurations, still allowing interference through the memory hierarchy. 3MM3 requires sampling nine core configurations, which are then used by RBF surrogate fitting to get the complete performance and power profiles across all core configurations. To get a meaningful sample for tail latency, the system needs to run for at least 10ms.

We evaluated Flicker in two ways: a) we set the profiling period to 10ms and profile the applications for a total of 90ms, search the best configuration that meets the QoS and power budget and maximizes the throughput using GA (takes 2ms), and run the system in that configuration for the remaining 8ms; b) Flicker only manages batch applications, and we set the cores assigned to latency-critical jobs to the highest – {6,6,6} – configuration, which reduces the power budget available for batch jobs. In this case, since we only predict throughput and power, we can directly apply the 3MM3 and RBF techniques over 1*ms* samples. Overall, we profile for 9*ms*, and run GA for 2*ms*. In both cases, we have to run the latency-critical service in lower configurations for extended periods of time. Since QoS is defined with respect to the 99^{th} percentile latency, even 1*ms* of slow requests is enough to violate QoS. As a result, we see extensive QoS violations by over an order of magnitude for the first methodology, and by 1.5× for the second.

We now compare the individual techniques in Flicker and CuttleSys. Flicker requires 9 profiling samples, while SGD only uses 2. To ensure a fair comparison, we show the prediction error of the RBF-based approach in performance and power in Fig. 9 when using 3 samples from the full 100ms timeslice (the algorithm was unable to converge when using two samples). The error is dramatically higher for Flicker with 3 samples, with outliers reaching up to 600%. Thus, with the same amount of information, the SGD-based reconstruction clearly outperforms RBF.

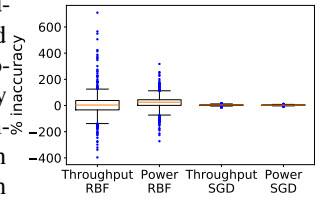


Fig. 9: Performance, power errors with SGD & RBF.

Next, we compare the exploration algorithms, DDS and GA [18]. Fig. 10a shows a subset of points in the entire space, as well as the points explored by DDS and GA. Black dots represent the points explored by GA, and pink dots the points explored by DDS. DDS explores more points on the pareto-optimal front and thus, obtains a higher-quality configuration compared to GA, shown by blue and yellow stars respectively, under a given power budget, shown by the dotted green line.

To quantitatively compare DDS with GA, we applied GA during the optimization phase instead of DDS, and used SGD for reconstruction. Fig. 10b shows the comparison of the gmean of throughput of CuttleSys with SGD and GA across power caps. Using DDS for optimization offers a performance improvement of up to 19% compared to GA for a 32-core system. This can be attributed to the GA algorithm being relatively slow in exploring a highly-dimensional search space compared to DDS. Also, the optimization algorithm must explore a higher number of configurations $27*4=108$ (including LLC allocations), compared to only 27 core configurations in [18]. We also note that the performance improvement is higher at lower power caps, as a large subset of configurations does not violate the power budget, and DDS can quickly explore the large space. As power constraints become more stringent, fewer configurations are valid, enabling GA to find the best configurations in a given amount of time. The improvement is the smallest for a 50% power cap as at that point, all cores often have to operate in their lowest configurations, and may even need to be switched off to meet the power budget.

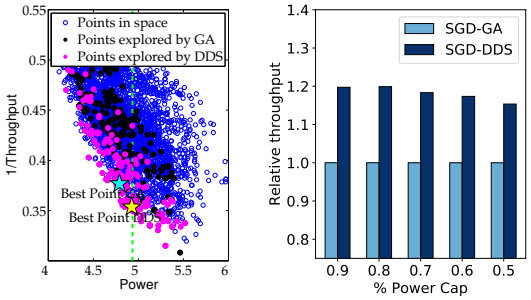


Fig. 10: (a) Comparison of DDS vs GA's ability to explore the design space. (b) Throughput with DDS and GA under different power caps using SGD for inference.

IX. Conclusions

We present CuttleSys, an online and practical resource management system for reconfigurable multicores, which quickly infers the performance and power consumption of each co-scheduled application across all core configurations and cache allocations, and arrives at a suitable configuration that meets QoS for latency-critical services, and maximizes throughput for batch workloads, under a power budget.

We evaluated CuttleSys across a set of diverse latency-critical and batch workloads, and showed that the system meets both the QoS and power budget at all times, while achieving significantly higher throughput for the batch applications than previous work, including core-level gating and Flicker. We also quantified the inference errors of the reconstruction algorithm in CuttleSys, and showed that they are low in all cases.

Acknowledgements

We sincerely thank Shuang Chen, Yu Gan, Yanqi Zhang, Nikita Lazarev, Mingyu Liang, Zhuangzhuang Zhou, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was partially supported by NSF CAREER Award CCF-1846046, NSF NeTS CSR-1704742, and gifts from Google, Facebook, VMWare, and Microsoft.

REFERENCES

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis lectures on computer architecture, 2013.

[2] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[3] ——, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[4] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2013, p. 607–618.

[5] J. Mars and L. Tang, "Whare-Map: Heterogeneity in "Homogeneous" Warehouse-Scale Computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2013, p. 619–630.

[6] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-Scale Latency-Critical Workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[7] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[8] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2019, p. 107–120.

[9] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[10] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozych, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2012.

[11] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[12] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *Proceedings of the 38th annual International Symposium in Computer Architecture*, 2011.

[13] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-critical Workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[14] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[15] F. Romero and C. Delimitrou, "Mage: Online and Interference-Aware Scheduling for Multi-Scale Heterogeneous Systems," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT18)*, November 2018.

[16] C.-H. Hsu, Q. Deng, J. Mars, and L. Tang, "SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-scale Datacenters," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2018, pp. 535–548.

[17] N. Kulkarni, F. Qi, and C. Delimitrou, "Pliant: Leveraging Approximation to Improve Datacenter Resource Efficiency," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 159–171, 2018.

[18] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2013, pp. 13–23.

[19] S. S. Jha, W. Heirman, A. Falcón, T. E. Carlson, K. Van Craeynest, J. Tubella, A. González, and L. Eeckhout, "Chrysso: An Integrated Power Manager for Constrained Many-core Processors," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, 2015, pp. 19:1–19:8.

[20] W. Zhang, H. Zhang, and J. Lach, "Dynamic Core Scaling: Trading off Performance and Energy beyond DVFS," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct 2015, pp. 319–326.

[21] "6th Generation Intel Processor Families for S-Platforms," August 2018.

[22] "8th and 9th Generation Intel Core Processor Families and Intel Xeon E Processor Family," October 2018.

[23] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power Management of Online Data-Intensive Services," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 319–330.

[24] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis, "Power Management of Datacenter Workloads Using Per-Core Power Gating," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 48–51, Jul. 2009.

[25] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *Proceedings of the 14th international ASPLOS*, 2009.

[26] D. Meisner and T. F. Wenisch, "DreamWeaver: Architectural Support for Deep Sleep," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2012, pp. 313–324.

[27] H. Kasture and D. Sanchez, "Tailbench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," in *IEEE International Symposium on Workload Characterization*, 2016.

[28] "SPEC CPU 2006," https://www.spec.org/cpu2006/.

[29] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 347–358.

[30] J. Sharkey, A. Buyuktosunoglu, and P. Bose, "Evaluating Design Tradeoffs in On-chip Power Management for CMPs," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*. New York, NY, USA: ACM, 2007, pp. 44–49.

[31] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, I. Nair, G. Dittmann, G. Janssen, N. Dhanwada, Z. Hu, P. Bose, and J. Darringer, "Exploring Power Management in Multi-core Systems," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 708–713.

[32] J. Chen and L. John, "Predictive coordination of multiple on-chip resources for chip multiprocessors," *Proceedings of the International Conference on Supercomputing*, pp. 192–201, 01 2011.

[33] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive Voltage/Frequency Scaling and Core Allocation for Balanced Energy and Performance on Multicore CPUs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 133–146.

[34] Y. Wang, K. Ma, and X. Wang, "Temperature-Constrained Power Control for Chip Multiprocessors with Online Model Estimation," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2009, pp. 314–324.

[35] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable Power Control for Many-Core Architectures running Multi-Threaded Applications," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 449–460.

[36] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "Thermal and Energy Management of High-Performance Multicores: Distributed and Self-Calibrating Model-Predictive Controller," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 1, pp. 170–183, Jan 2013.

[37] R. Nishtala, V. Petrucci, P. Carpenter, and M. Själander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," 12 2019.

[38] "Intel® 64 and IA -32 Architectures Software Developer's Manual, System Programming Guide, Part 2," 2016.

[39] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and Reining

in Tail Queries with Quick Voltage Boosting," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 271–282.

[40] "2nd Generation Intel Core Processor Family Desktop," January 2011.

[41] "Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge," *Hot Chips: A Symposium on High Performance Chips*, 2012.

[42] R. Kumar and G. Hinton, "A family of 45nm IA processors," in *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, Feb 2009, pp. 58–59.

[43] K. Ma and X. Wang, "PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2012, pp. 13–22.

[44] H. Zhang and H. Hoffmann, "Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2016, pp. 545–559.

[45] M. Arora, S. Manne, I. Paul, N. Jayasena, and D. M. Tullsen, "Understanding Idle Behavior and Power Gating Mechanisms in the Context of Modern Benchmarks on CPU-GPU Integrated Systems," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 366–377.

[46] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 658–670.

[47] A. M. Rahmani, B. Donyanavard, T. Mück, K. Moazzemi, A. Jantsch, O. Mutlu, and N. Dutt, "SPECTR: Formal Supervisory Control and Coordination for Many-core Systems Resource Management," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2018, pp. 169–183.

[48] S. Kanev, K. Hazelwood, G. Wei, and D. Brooks, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014, pp. 31–40.

[49] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 64–75.

[50] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures," in *Proceedings of the 3rd Conference on Computing Frontiers*. New York, NY, USA: ACM, 2006, pp. 29–40.

[51] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: A Scheduler for Heterogeneous Multicore Systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, Apr. 2009.

[52] J. Chen and L. K. John, "Efficient Program Scheduling for Heterogeneous Multi-Core Processors," in *2009 46th ACM/IEEE Design Automation Conference*, July 2009, pp. 927–930.

[53] D. Koufaty, D. Reddy, and S. Hahn, "Bias Scheduling in Heterogeneous Multi-core Architectures," in *Proceedings of the 5th European Conference on Computer Systems*. New York, NY, USA: ACM, 2010, pp. 125–138.

[54] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 177–188.

[55] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, "ACFS: A Completely Fair Scheduler for Asymmetric Single-isa Multicore Systems," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2015, pp. 2027–2032.

[56] J. Chen, A. A. Nair, and L. K. John, "Predictive Heterogeneity-Aware Application Scheduling for Chip Multiprocessors," *IEEE Transactions on Computers*, vol. 63, no. 2, pp. 435–447, 2014.

[57] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 213–224.

[58] G. Liu, J. Park, and D. Marculescu, "Dynamic Thread Mapping for High-Performance, Power-Efficient Heterogeneous Many-Core Systems." in *ICCD*. IEEE Computer Society, 2013, pp. 54–61.

[59] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 363–374.

[60] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-core Architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2010, pp. 29–40.

[61] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout, "Mind The Power Holes: Sifting Operating Points in Power-Limited Heterogeneous Multicores," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 56–59, Jan 2017.

[62] ——, "Maximizing Heterogeneous Processor Performance Under Power Constraints," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 29:1–29:23, Sep. 2016.

[63] S. Navada, N. Choudhary, S. Wadhavkar, and E. Rotenberg, "A Unified View of Non-Monotonic Core Selection and Application Steering in Heterogeneous Chip Multiprocessors," 01 2013, pp. 133–144.

[64] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, and L. Tang, "Octopus-Man: QoS-driven Task Management for Heterogeneous Multicores in Warehouse-Scale Computers," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 246–258.

[65] S. Ren, Y. He, S. Elnikety, and K. S. McKinley, "Exploiting Processor Heterogeneity in Interactive Services," in *ICAC*, January 2013.

[66] S. Ren, Y. He, and K. S. McKinley, "A Theoretical Foundation for Scheduling and Designing Heterogeneous Processors for Interactive Applications," in *International Symposium on Distributed Computing (DISC)*. European Association for Theoretical Computer Science, October 2014.

[67] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. McKinley, "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, October 2017.

[68] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite Cores: Pushing Heterogeneity Into a Core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 317–328.

[69] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Trace Based Phase Prediction for Tightly-coupled Heterogeneous Cores," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2013, pp. 445–456.

[70] H. R. Ghasemi and N. S. Kim, "RCS: Runtime Resource and Core Scaling for Power-Constrained Multi-core Processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. New York, NY, USA: ACM, 2014, pp. 251–262.

[71] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt, "MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 305–316.

[72] F. Afram and K. Ghose, "FlexCore: A Reconfigurable Processor Supporting Flexible, Dynamic Morphing," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, Dec 2015, pp. 30–39.

[73] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-Silicon CPU Adaptation Made Practical Using Machine Learning," in *Proceedings of the 46th International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–26.

[74] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing Server Efficiency in the Face of Killer Microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 185–198.

[75] Y. Zhou and D. Wentzlaff, "The Sharing Architecture: Sub-Core Configurability for IaaS Clouds," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2014, p. 559–574.

[76] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2007, pp. 186–197.

[77] Y. Zhou, H. Hoffmann, and D. Wentzlaff, "CASH: Supporting IaaS Customers with a Sub-Core Configurable Architecture," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, p. 682–694.

[78] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[79] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and ManycoreAArchitectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[80] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[81] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2013, pp. 308–319.

[82] E. H. Sibley, P. J. Fleming, and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," 1986.

[83] I. H. Witten, E. Frank, and G. Holmes, *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd Edition.

[84] L. Bottou, *Large-Scale Machine Learning with Stochastic Gradient Descent*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[85] K. C. Kiwiel, "Convergence and efficiency of subgradient methods for quasiconvex minimization," *Mathematical Programming*, vol. 90, no. 1, pp. 1–25, Mar 2001.

[86] B. A. Tolson and C. A. Shoemaker, "Dynamically Dimensioned Search Algorithm for Computationally Efficient Watershed Model Calibration," *Water Resources Research*, vol. 43, no. 1, pp. n/a–n/a, 2007, w01413.

[87] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 69–82.

[88] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[96] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the Wild: A Unified Analysis of HOG WILD! -style Algorithms," in *Proceedings*

[89] A. Rajaraman and J. Ullman, "Textbook on Mining of Massive Datasets. Rightscale." 2011, https://aws.amazon.com/solution-providers/isv/rightscale.

[90] R. Bell, Y. Koren, and C. Volinsky, "The BellKor 2008 Solution to the Netflix Prize," Tech. Rep., 2007.

[91] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *Proceedings of the International Conference on Computational Statistics (COMPSTAT)*. Paris, France, 2010.

[92] K. C. Kiwiel, "Convergence and Efficiency of Subgradient Methods for Quasiconvex Minimization," in *Mathematical Programming (Series A) (Berlin, Heidelberg: Springer) 90 (1): pp. 1-25, 2001.*

[93] A. Gunawardana and C. Meek, "A Unified Approach to Building Hybrid Recommender Systems," in *Proc. of the Third ACM Conference on Recommender Systems (RecSys)*. New York, NY, 2009.

[94] R. Burke, "Hybrid Recommender Systems: Survey and Experiments," *User Modeling and User-Adapted Interaction*, vol. 12, no. 4, pp. 331–370, Nov. 2002.

[95] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*. USA: Curran Associates Inc., 2011, pp. 693–701.
of the 28th International Conference on Neural Information Processing Systems - Volume 2. Cambridge, MA, USA: MIT Press, 2015, pp. 2674–2682.

[97] R. B. R. Chowdhury, A. K. Kannepalli, S. Ku, and E. Rotenberg, "AnyCore: A synthesizable RTL model for exploring and fabricating adaptive superscalar cores," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 214–224.

[98] "big.LITTLE Technology: The Future of Mobile," https://www.arm.com, 2013.

[99] C. F. J. Wu and M. S. Hamada, *Experiments: Planning, Analysis, and Optimization*. John Wiley and Sons, Inc., 2009.

[100] H.-M. Gutmann, "A Radial Basis Function Method for Global Optimization," *Journal of Global Optimization*, vol. 19, no. 3, pp. 201–227, 2001.

[101] J. Mueller, C. Shoemaker, and R. Piche, "SO-MI: A Surrogate Model Algorithm for Computationally Expensive Nonlinear Mixed-integer Black-box Global Optimization Problems," *Computers and Operations Research*, May 2013.

[102] R. G. Regis and C. A. Shoemaker, "Combining Radial Basis Function Surrogates and Dynamic Coordinate Search in High-dimensional Expensive Black-box Optimization," *Engineering Optimization*, May 2013.

[103] ——, "A Stochastic Radial Basis Function Method for the Global Optimization of Expensive Functions," *INFORMS Journal on Computing*, Fall 2007.

[104] ——, "Local Function Approximation in Evolutionary Algorithms for the Optimization of Costly Functions," *IEEE Transactions on Evolutionary Computation*, October 2004.