SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains*

Rohan Basu Roy, Tirthak Patel, Devesh Tiwari
Northeastern University

***Abstract*—Multi-core architectures have enabled data centers to increasingly co-locate multiple jobs to improve resource utilization and lower the operational cost. Unfortunately, naively co-locating multiple jobs may lead to only a modest increase in system throughput. Worse, some users may observe proportionally higher performance degradation compared to other users co-located on the same physical multi-core system. SATORI is a novel strategy to partition multi-core architectural resources to achieve two conflicting goals simultaneously: increasing system throughput and achieving fairness among the co-located jobs.**

I. INTRODUCTION

Background and Problem. Co-locating workloads on chip multiprocessors (CMPs) is an attractive approach for cloud computing service providers as it enables them to improve resource utilization and lower the capital and operational costs of running their data centers [11], [19], [28], [54], [59], [60], [64], [67], [79], [91]. Co-location of workloads is likely to yield even higher benefits in the future as data centers are continuously observing an increase in the number of throughput-oriented workloads [8], [23]–[25], [80] – these workloads are naturally most suitable for co-location because such jobs are often long-running and do not have strict latency requirements.

While co-location of workloads can potentially increase throughput, it can lead to unfairness where some users or workloads may observe proportionally higher performance degradation than others, compared to their no-interference (*i.e.,* co-location free) execution. Consequently, some prior research works have attempted to optimize for throughput [40], [41], [52], [71], [81], [86], [94], and some other works have focused on achieving high fairness [46], [66], [90]. However, achieving both the goals (throughput and fairness) *simultaneously* for co-located workloads continues to be challenging because throughput and fairness are fundamentally conflicting goals (*system providers often want higher throughput and fairness is preferred by users*) — optimizing one often compromises the other. *Next, we discuss why existing approaches, while useful, do not achieve the full potential toward providing a near-optimal solution in a multi-objective setting.*

Challenges, Existing Approaches and Gaps. The difficulty in achieving optimality stems from two significant challenges: *(1) complexity of the effects of "multiple" shared architectural resources (e.g., cache, memory, etc.) at runtime towards the optimization goal (e.g., system throughput), and (2) co-optimizing for conflicting goals simultaneously at runtime (e.g., throughput and fairness).* An unfair system, even with high throughput, is undesirable since it leads to some users experiencing a disproportionate slowdown, incorrect resource usage charging, delay of scientific discovery, and bad user experience (indirect monetary loss). Fairness of the system can be quantified by widely-used fairness index (e.g., Jain's Fairness Index [35]), which can capture the degree of "similarity" in the performance degradation of co-located jobs compared to co-location-free execution (Sec. II).

Consider a relatively simple scenario where multiple jobs are co-located on a CMP machine and share only "one" resource (*e.g.,* last-level cache), and the aim is to improve one single goal (*e.g.,* system throughput). Prior works have provided practical solutions to the problem of this nature [7], [38], [46], [53], [81], [90], [94]. For example, a recent solution dCAT [90] changes last-level cache (LLC) ways allocation dynamically among co-located workloads to achieve high throughput. The core intuition behind these solutions is to assess the relative "utility" of an additional share of a resource on a particular job's performance. Such an approach classifies some jobs as "donors" and others as "receivers" to improve the throughput.

Adding more shared resources (e.g., memory bandwidth) makes the problem even more challenging, as previous works such as CoPart, Heracles, and PARTIES have demonstrated [12], [52], [66]. These works have taken two different approaches to optimize the partitioning of multiple shared resources. In the first approach, partitioning of individual resources is done separately, but the decisions are communicated to reach a global optimal resource partition. For example, CoPart [66] maintains two separate finite state machines (FSM), one for shared cache and one for memory bandwidth. These FSMs are not joint or linked but are aware of each other's decisions. In the second approach, partitioning of individual resources is performed in one dimension at a time (similar to a gradient descent method). For example, Heracles [52] and PARTIES [12] (the most recent resource partitioning strategy) perform resource partitioning in a gradient descent style where partitioning of one resource is explored first before adjusting the allocations for other resources. CoPart is simple, but may not scale well to multiple resources, unlike PARTIES and Heracles, that can scale well to multiple resources.

However, a source of sub-optimality in both the approaches is that these do not explore partitioning multiple resources simultaneously. Joint exploration of multiple resources simultaneously is necessary since benefits of increasing one resource

*A part of this work was performed during an unprecedented time – around the peak of the COVID-19 pandemic. We were able to conduct this scientific study only because first-line responders and essential workers worked tirelessly to keep our community safe and functioning. We are grateful for their effort. This paper is dedicated to the memories of all the first-line responders and essential workers who sacrificed their lives trying to keep ours safe.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00031

allocation for a given job may not be realized fully until the allocation for other resources are appropriately resized, but this leads to search space explosion (Sec. II).

Finally, while optimizing for one goal is hard enough, adding one more compounds the challenge further. This is especially true if the second goal conflicts with the first goal (*e.g.,* improving both throughput and fairness). Prior works, when optimizing for fairness or throughput, often resort to targeting one goal and settle with best-effort improvement for the other. For example, CoPart [66] focuses on improving fairness and demonstrates that the throughput is not hurt compared to the baseline (unmanaged partitioning of the resources). This is because effectively controlling and finding a balance between conflicting goals online is challenging on a real system (experimental evidence in Sec. II).

SATORI: Key Contributions and Evaluation. SATORI is the first solution that *actively* controls both system throughput and fairness goals *simultaneously*, when multiple workloads are co-located on a CMP machine and share multiple architectural resources. SATORI proposes new solutions to existing gaps of optimality in current approaches.

SATORI develops a new approach grounded in Bayesian Optimization (BO) theory to intelligently explore multi-resource partitioning configuration space. In particular, SATORI's BO based approach enables us to design a practically-feasible technique that can work on a real system in an online fashion. The key intuition behind using BO is to build *simple and just-accurate-enough models for finding near-optimal solutions*, without requiring offline profiling, instrumentation, offline deep learning/reinforcement based training or building complex performance models which may incur high-overhead and may not be portable to new situations depending upon the nature of the offline training dataset [16], [45], [47], [48], [54], [75], [85]. As our evaluation also confirms, tolerating a slight inaccuracy of the model allows SATORI to achieve near-optimal configurations in an online fashion. SATORI enables fast and efficient navigation of large configuration space to find the optimal resource partition configuration (Sec. III-A). It performs joint exploration of multiple resources simultaneously - removing the limitation of existing approaches that maintain one finite state machine for each shared resource or explore one resource dimension at a time.

One of the vital novel insights we discover is that it is beneficial to trade-off one goal for another *temporarily* to achieve higher returns on both the goals over long-term (as discussed in Sec. III-C). To exploit this observation, SATORI intelligently prioritizes one goal over the other on a short-term basis but then strikes a balance between the two goals on a long-term basis – *to the best of our knowledge,* SATORI *is the first solution to observe and exploit this finding*. Unfortunately, dynamic re-prioritization of goals causes the objective function, which is being optimized, to *change* over time, and hence, it becomes challenging and ineffective to apply BO. To address this challenge, SATORI implements a new enhancement that dynamically and separately monitors the system-level throughput and job-level fairness indicators to construct a new overall combined BO objective function targeting multiple goals. This objective function successfully changes the prioritization of the goals, while maintaining the expected behavior of the traditional BO by *bounding the magnitude and period of dynamic re-prioritization of goals (Sec. III-B)*. The inherent design of SATORI makes it suitable for additional concurrent optimization goals.

SATORI evaluation demonstrates its effectiveness across a range of scenarios and parallel workloads. For example, in terms of throughput, SATORI outperforms most recent workload co-location techniques including dCAT, CoPart, and PARTIES by 19%, 17%, and 14%, respectively while achieving higher fairness *at the same time* by 25%, 17%, and 14%, respectively. SATORI performs within 8% of the offline, practically-infeasible oracle scheme. SATORI can be complementary and potentially combined with recent targeted approaches based on economics, resource multiplexing, and machine learning (including [43], [59], [60], [62], [64]).

SATORI is available at https://github.com/rohanbasuroy/satori

II. Characterizing Challenges and Opportunities in Multi-Goal Shared Resource Partitioning

New CMP servers consist of multiple resources that can be shared across workloads (*e.g.,* cores, LLC, memory bandwidth, power). Chip manufactures provide a mechanism to partition these resources among workloads in a dynamically configurable way (*e.g.,* Intel's CAT, MBA, RAPL [1], [15], [33]). Similar to previous works, we leverage these capabilities to partition resources among concurrently co-located workloads. Before analyzing our characterization study, we define the terms used in this paper.

Resource partitioning configuration (or simply, a *configuration*) refers to shares of different jobs of shared architectural resources. For example, consider jobs A and B, which share two resources – cores (6 cores) and LLC (4 cache ways per set). A configuration is defined as one permutation of resource allocation of all available resources to all co-located jobs. Thus, a possible configuration is {3 (50%) cores to job A, 3 (50%) cores to job B, 1 (25%) cache way per set to job A, and 3 (75%) cache ways per set to job B}.

System throughput and fairness objectives can be expressed using multiple metrics as described below. These metrics offer different advantages and have been widely-used in various past works to maximize the system throughput and fairness by optimizing the chosen metric [4], [27], [37], [66], [90].

System throughput can be expressed using multiple metrics. For example, it can be expressed as the geometric mean of the speedups of the co-located jobs as compared to their baseline isolation performance at a given time during their run: $\left(\prod_{i=1}^{N} s_i\right)^{\frac{1}{N}}$, where N is the number of co-located jobs and s_i is the speedup of the i^{th} job. It can also be expressed as harmonic mean of speed ups, or as the sum of instructions per second [13], [20], [69], [88].

Fairness measures the degree of "similarity" in the performances of co-located jobs. It can also be expressed using multiple metrics. For example, we use Jain's Fairness Index [35] as our metric of fairness: $\frac{1}{1+\text{CoV}^2}$, where CoV is the coefficient of variation (standard deviation as a fraction of the mean) of the speedups (from baseline isolation performances) of the co-located jobs. Thus, if the standard deviation of job speedups is small (*i.e.,* the speedups are similar and hence, the jobs are treated highly fairly), the CoV is small, and the fairness index is close to 1. On the other hand, if it is large, the CoV is

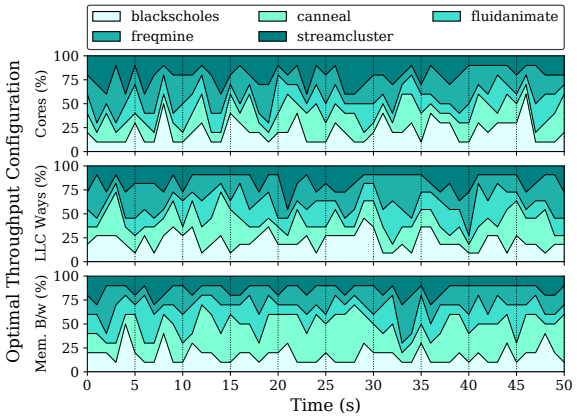


Fig. 1. The configuration which achieves optimal throughput changes significantly and frequently over time, for all shared architectural resources.

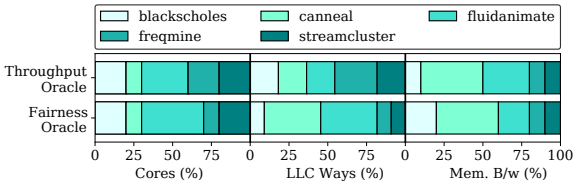


Fig. 2. The optimal configurations for throughput and fairness differ significantly at any point.

large, and the fairness is close to 0. Another fairness metric is: $1 - \text{CoV}$, which evaluates to 1 in the case of perfect fairness and can also be negative in cases of unfairness [21], [73].

Next, we note that finding optimal configuration requires searching a large configuration space. Unfortunately, the configuration search space grows prohibitively as the number of co-located jobs and shared resources increase. Formally, suppose there are N_{res} resources, M_{jobs} co-located jobs, and the r^{th} resource has $U_{\text{units}}(r)$ units of the resource, then the size of the configuration space can be expressed as $S_{\text{conf}} = \prod_{r=1}^{N_{\text{res}}} \binom{U_{\text{units}}(r)-1}{M_{\text{jobs}}-1}$.

For example, if three co-located jobs share just two resources (*e.g.,* number of cores and memory bandwidth), each resource with 10 units, then the total number of possible configurations is $1,296$. When the number of co-located jobs increases to four, the size of the configuration space grows to $7,056$. Simply adding one more shared resource increases the size of the configuration space to 5,92,704. Worse, the optimal configuration does not remain the same over time, even for the same set of co-located jobs. To better understand the magnitude of this challenge, we tracked how the optimal resource configuration changes over time for a five-job mix sharing three shared architectural resources (*i.e.,* cores, last-level cache, memory bandwidth). Optimal resource partitioning is obtained via an exhaustive offline search of all possible configurations to maximize the system throughput.

Fig. 1 shows the optimal configuration for throughput over time for a job mix consisting of 5 jobs from the PARSEC benchmark suite [6] (evaluation methodology details in Sec. IV). We observe that the configuration to achieve optimal throughput can change by more than 20% during the course of the run. In essence, this is because jobs may have different program phases, and each phase may have different sensitivity toward architectural resources. Unfortunately, exploring the full search space to find the optimal resource partitioning configuration requires several hours of exhaustive search. We found that the optimal configuration for fairness also varies dynamically and significantly (results not shown for brevity).

Observation 1. *Search space of architectural resource partitioning configurations grows prohibitively large as the number of shared architectural resources and co-located jobs increase. Also, the optimal resource partitioning configuration is challenging to find and frequently changes over time.*

Next, Fig. 2 demonstrates the differences in the optimal configuration for the two conflicting goals at the same point during the execution. These configurations are significantly different for the two goals (up to 40% difference). Employing throughput optimized configuration achieves only 67% of the fairness obtained using fairness optimized resource partition. Similarly, fairness optimized resource partition achieves only 59% of the throughput obtained with throughput optimized resource partition.

One may hypothesize that the "average" of the optimal resource partitioning configurations for both goals might perform reasonably well for both the goals and might be a good compromise. Note that obtaining this "average" optimal configuration is also practically infeasible since it requires obtaining several hours of exhaustive offline search to obtain the optimal configuration for each goal separately. Even hypothetically assuming that optimal configuration is available instantaneously using oracle knowledge, such a derived "average" optimal configuration performs poorly in both aspects. It achieves only 59% of the *oracle throughput* and 72% of the *oracle fairness*.

Observation 2. *Optimal resource partitioning configurations for conflicting goals (e.g., system throughput vs fairness) can be very different from each other. Finding a resource partitioning configuration that strikes a balance between the conflicting goals is challenging.*

An alternative intuitive method to strike a balance between throughput and fairness is to optimize for throughput half of the time and optimize for fairness the other half, although hypothetically assuming that optimal resource partitioning configuration is available instantaneously. Our results revealed that this approach also leads to significantly sub-optimal results (achieving only 72% of the oracle throughput and 81% of the oracle fairness). But, slight improvement over previous "average" optimal resource partitioning method led us to pose a fundamental question: *do we have an equal opportunity (or difficulty) in finding the balance between throughput and fairness over time?* In other words, is it *easier* to obtain higher throughput at some point with small temporary sacrifice in fairness. At a later point, when it becomes *easier* to obtain higher fairness with small temporary sacrifice in throughput such that there is a net gain in at least one goal when both the points are considered together. Another intuition behind this exploration is the following: since the optimal configurations for both throughput and fairness changes over time, their respective contributions toward throughput and fairness may not remain static either, and this should open up an opportunity

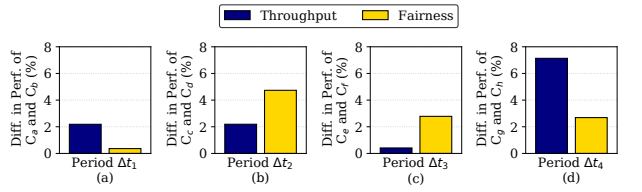


Fig. 3. There exists an opportunity to re-balance the conflicting goals (throughput and fairness) over time and one does not need to put "equal emphasis" on both the goals at all times.

for "temporarily" trading one goal for another.

Fig. 3 (a) shows the differences in throughput obtained with two sample configurations at point Δt_1 and the corresponding difference in fairness for the same two sample configurations at point Δt_1. Fig. 3 (b) quantifies the same metrics but at a different point Δt_2 for two different sample configurations. Note that this example uses the same five-job mix as Fig. 1 using the same methodology. The choice of configurations C_a, C_b, C_c, and C_d and periods Δt_1 and Δt_2 is made for simplicity to provide evidence for the following observation (there are many other similar configurations).

We observe that at both points (Δt_1 and Δt_2), the percentage difference in throughput between the two configurations is the same. However, the corresponding difference in fairness for the same two configurations is in different directions (at Δt_1 it is lower and at Δt_2 it is higher). This suggests that if one prioritizes throughput at Δt_1 and prioritizes fairness at Δt_2, there is a net gain in fairness without sacrificing throughput. Fig. 3(c) and (d) show a similar example where there is a net gain in throughput without sacrificing fairness, by selectively prioritizing one goal over the other at different times.

Observation 3. *There is an opportunity to re-balance conflicting goals over time, i.e., prioritizing one goal over the other for a short duration and vice versa during another period yields higher benefits.*

SATORI bases its design on the above insights and findings.

III. SATORI: Design and Implementation

SATORI leverages Bayesian Optimization (BO) to efficiently navigate the search space to find near-optimal partitioning configurations (Sec. III-A discusses the details of the solution). As a high-level summary, BO builds a low-overhead *proxy model* to predict the performance of different configuration samples and uses an *acquisition function*, which directs BO toward samples to evaluate (principled, intelligent exploration of the search space). BO's underlying proxy model is not wholly accurate by design but becomes iteratively more accurate and suitable for online use. Employing a proxy model underneath and navigating the configuration space with the help of an acquisition function, allows SATORI to adjust partitions for multiple resources simultaneously.

Next, SATORI carefully constructs the objective function that incorporates methods to achieve multiple goals (*i.e.,* throughput and fairness) at the same time. SATORI's objective function is also carefully designed to support one of the key ideas of SATORI: "prioritize" one goal over another temporarily to exploit the opportunity of re-balancing conflicting goals over time (*i.e.,* configurable control). Finally, SATORI's objective function is designed to be extensible and can include additional goals (*e.g.,* energy-efficiency). The BO exploration process optimizes this objective function to find the near-optimal configuration.

Notably, SATORI does not require any programmer hints, recompilation, new architecture support, or instrumentation. SATORI can be deployed readily on platforms where hardware support for architectural resource partitioning is available.

A. How does SATORI employ Bayesian Optimization for efficient search space exploration?

BO is a theoretically-grounded method for solving the problem of maximizing a black-box objective function, f. BO does not need to know the relationship between the input and the objective function. BO incorporates a prior belief of the objective function, and by querying it multiple times, it develops a posterior that better approximates f. Traditionally, the optimization aims to figure out the input x^* that maximizes the objective function $f(x)$ with *minimum possible queries*. Mathematically, BO can be expressed as following (where $\mathcal{X}$ is the search space of interest):

$$x^* = \underset{x \in \mathcal{X}}{\operatorname{argmax}}\, f(x) \tag{1}$$

In the context of SATORI, the input, x, to the objective function, f, is a resource partition configuration which allocates different units of each resource to different co-located jobs. All such possible *resource partitioning configurations* comprise the search space $\mathcal{X}$ and are referred to as *sample points* in the space. The objective function, f, is unknown because SATORI does not know the relationship between the input configuration and the objective function (*i.e.,* the function that expresses the relationship between a given configuration and corresponding outcome in terms throughput and fairness).

However, SATORI can evaluate the value of the objective function, f, given an input x (resource partition configuration) by running the system with the given configuration and observing the performance of all co-located jobs. The value of the objective function indicates the quality of the figure of merit (*e.g.,* throughput, or fairness). However, unlike traditional cases where the optimization problem is to maximize the objective function [3], [31], [32], [57], [58], [72], SATORI requires careful construction of the objective function due to the dynamically changing nature of the objective (dynamic prioritization among multiple goals). This aspect of SATORI and the corresponding solution is discussed later in Sec. III-B. SATORI uses BO to find the optimal value of the objective function with just a few function evaluations (*i.e.,* running a minimum number of configurations).

To identify the global optimum of the objective function, BO intelligently explores the search space by evaluating the objective function with different input samples (configurations). As it explores the sample space, it builds a stochastic model of the unknown objective function and keeps updating it, and uses this knowledge to identify which samples to evaluate next such that it reaches near the global optimum. The two key components that enable this are: (1) the *proxy model*, and (2) the *acquisition function*. BO uses a proxy model, $\mathcal{M}(x)$, to stochastically estimate the performance of different configurations in the space. This stochastic model is improved iteratively during the space exploration process, as more

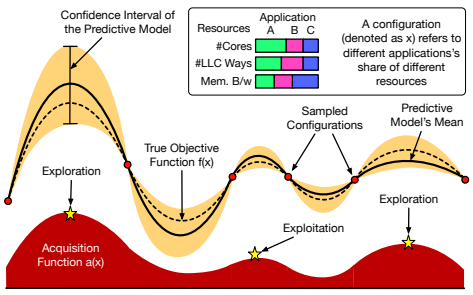


Fig. 4. Visual depiction of Bayesian optimization (BO) components including configuration, true objective function, proxy model, and acquisition function.

Algorithm 1 SATORI BO Engine Algorithm.

```
1: Input: Initial resource configurations (S_init).
2: Run the system with S_init & record throughput and fairness.
3: Record baseline (isolation) performances.
4: while TRUE do
5:     Generate objective function (details in Sec. III-B and III-C).
6:     Update the proxy model M(x).
7:     Compute the acquisition function a(x).
8:     Optimize a(x) and find the next sample to evaluate.
9:     Run the system with the selected configuration x.
10:    Record throughput and fairness for x.
11:        & add these values to the existing set.
12:    if End of a job or Start of new job or Every reset interval
13:        Reset baseline (rerecord isolation performances).
```

points are sampled. Fig. 4 shows a snapshot of the state of a BO execution during exploration where 6 points are already sampled, and BO has estimates for the rest of the points as predicted by the proxy model. We note that the proxy model predicts a range of values for non-sampled points - it consists of both predicted mean and variance. The variance captures the model's "uncertainty" at a given point. SATORI chooses Gaussian Process (GP) as the proxy model, which means that each sample point (configuration), the uncertainty follows a Gaussian (Normal) distribution defined by the mean (μ) and the standard deviation or uncertainty (σ). SATORI uses the Matérn covariance kernel ($\frac{5}{2}$) for its GP proxy model [42], [77].

Next, BO needs to steer itself toward the global optimum by sampling the "most promising points". BO uses the acquisition function to move in the right direction during the space exploration process. That is, an acquisition function $a(x)$ dictates which configuration points to sample next such that BO moves closer to the best performing samples in the space. As shown in Fig. 4, acquisition function returns different values for different non-sampled points at a given stage; mostly, the acquisition function wants the BO to explore different neighborhoods. It strikes a balance between "exploration" (exploring the areas of the configuration space with significant uncertainty) and "exploitation" (exploiting the vicinity of the neighborhood where the predicted mean value of the proxy model is high). SATORI chooses the Expected Improvement (EI) method as the acquisition function that provides a reasonable balance between exploration vs. exploitation at a low evaluation cost [77]. Note that as more points are sampled, the proxy model is updated, and the acquisition function is re-evaluated after each sample. This iterative process of SATORI is summarized in Algorithm 1. Here S_{init} refers to the initial resource configuration in which all available resources are divided equally among the workloads.

B. Constructing SATORI's Objective Function

Unlike the traditional BO implementation, where evaluating the objective function returns a single value (*e.g.,* runtime) to be maximized or minimized, SATORI needs to balance multiple goals (throughput and fairness). But, to make decisions about which configurations to evaluate next, SATORI still needs to assess the "goodness" of a configuration when evaluated (*i.e.,* the system is run for a short duration under a given configuration).

To this end, SATORI designs an objective function (*i.e.,* a performance score that is assigned to a configuration at the end of the period when the system is run under a given resource partition configuration). This function guides SATORI to search in the right direction in the large configuration space. To achieve that effectively, the objective function needs to treat both the goals equally and be comparable. Unfortunately, in general, different goals are calculated using different formulas and have different ranges. For instance, when fairness is calculated as $1 - \text{CoV}$, where CoV is the coefficient of variance of the speedups of the jobs compared to their respective baseline isolation performances, the fairness metric has no lower bound. So we normalize them to obtain the same range of 1 to 0. Also, SATORI needs to be configurable where different weights can be given to different goals. Therefore, the objective function can be expressed as:

$$f(x) = \sum_{i=1}^{K} W_i \times \text{Goal}_i(x) = W_T \times \text{T}(x) + W_F \times \text{F}(x) \tag{2}$$

Here, W_T is the weight on throughput $T(x)$ and W_F is the weight on fairness $F(x)$ of the evaluated configuration x. However, note that this objective function is configurable. It can be reduced to work with just one goal (*e.g.,* throughput), and it can be extended to work with K different goals. While, this objective function works if weights are configured at the start and never changed during the runtime, recall that we need to do dynamic re-balancing of weights to exploit the opportunity of temporary prioritization (discussed in detail in Sec. III-C). This means that the BO engine needs to allow the weights W_T and W_F to change over time. However, traditional BO cannot deal with this as it only optimizes over a single non-dynamically-changing objective function, as shown in Fig. 5. Traditional BO optimizes its static objective function as it collects more configuration samples (3 samples at t_1 to 4 samples at t_2). However, SATORI's objective function changes between two already sampled points (e.g., C_a and C_b). The example shown in the figure has equal weights (0.5) on throughput and fairness at t_1, and the objective function is constructed accordingly (same as traditional BO in the example). However, at t_2, the weights change to place 0.75 weight on throughput and 0.25 on fairness, and the objective function needs to be reconstructed accordingly.

In traditional BO, when the objective function changes, the BO engine has to re-sample already sampled configurations to reconstruct the proxy model for the objective function. However, this incurs a high overhead because all the configurations are required to be re-sampled in every iteration, which is prohibitively infeasible in a real online system.

To combat this problem, SATORI employs a novel technique that maintains a separate record of configuration-dependent

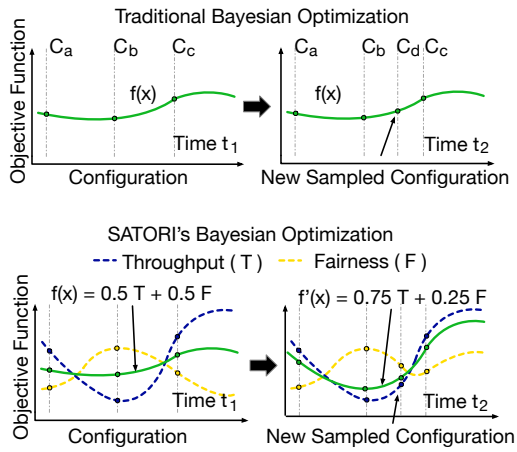


Fig. 5. SATORI maintains separate throughput and fairness performances and constructs a fresh objective function every iteration based on dynamic weights.

performances of every single conflicting goal. The benefits of this strategy are multi-fold: (1) The reduction in the overhead of objective function reconstruction is significant. Instead of re-sampling the configurations, this enables us to use existing goal-specific records to perform a software-based reconstruction of the proxy model. While this has slightly higher overhead than simply updating proxy model like in traditional BO, our evaluation (Sec. V) shows that this overhead does not interfere with the execution of co-located jobs, and in fact, the performance gain outweighs the overhead significantly. (2) This approach makes SATORI portable, customizable, and extensible to multiple objectives without much user-based coding effort since the goals are weighed independently of each other (Eq. 2).

Next, we discuss how we dynamically re-balance the weights set on throughput and fairness.

C. Dynamic Prioritization of Goals in SATORI

SATORI delivers better performance by dynamically setting different priorities on throughput and fairness. However, it still has to ensure that, over the long-term, both throughput and fairness achieve equal priorities. As discussed in Sec. III-B, this is achieved in the form of "weights". On average, both throughput and fairness should receive an equal weight of 0.5. To do so, SATORI uses an equalization period (T_E) over which the weights received by throughput and fairness have to be 0.5 on average. Moreover, SATORI also uses a shorter prioritization period (T_P) over which it can prioritize one goal over the other. Therefore, both throughput and fairness have two weight components: *equalization* and *prioritization*.

As the purpose of the equalization weight is to "equalize", it is set according to the imbalance between the throughput and fairness weights so far in the current equalization period. For instance, if throughput has received more weight so far, then its equalization weight should be decreased as the end of the equalization period approaches. Therefore, SATORI assigns the weights as following:

$$W_{TE} = \frac{1}{2}t_e - \sum_{i=1}^{t_e} W_{T_i} \quad \& \quad W_{FE} = \frac{1}{2}t_e - \sum_{i=1}^{t_e} W_{F_i} \quad (3)$$

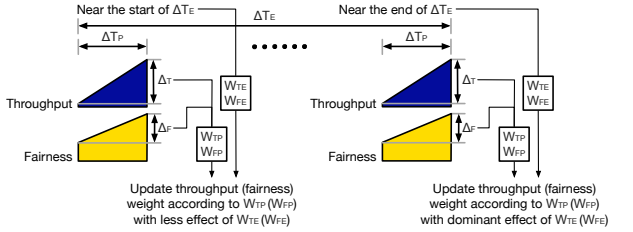


Fig. 6. The prioritization weights depend on improvement in throughput and fairness performances during the prioritization period. The equalization weights are more dominant toward the end of the equalization period.

Here, W_{TE} (W_{FE}) is the equalization throughput (fairness) weight, t_e is the amount of time elapsed in the equalization period, and W_{T_i} (W_{F_i}) is the throughput (fairness) weight during the i^{th} iteration (note that time is discretized here (0.1s intervals as configured in the current implementation) and hence, is represented as a summation). Thus, the equalization weight is set based on the amount of time elapsed and the previous weights set during the current equalization period.

On the other end, the prioritization weight is set based on the percent change in throughput and fairness during the prioritization period. Intuitively, the prioritization weight of throughput (fairness) should be adjusted based on how well the fairness (throughput) goal was achieved in the previous prioritization period. If fairness is given higher priority during the previous period and if it successfully utilized this opportunity, throughput should get this opportunity during the next interval, and vice versa. Therefore, SATORI assigns the prioritization weights as following:

$$W_{TP} = \frac{1}{4} + \frac{1}{2}\frac{\Delta_F}{\Delta_T + \Delta_F} \quad \& \quad W_{FP} = \frac{1}{4} + \frac{1}{2}\frac{\Delta_T}{\Delta_T + \Delta_F} \quad (4)$$

Here, W_{TP} (W_{FP}) is the prioritization throughput (fairness) weight, Δ_T (Δ_F) is the % improvement in throughput (fairness) from the beginning of the prioritization period (T_P) until its end. Evidently, for example, if the % improvement in fairness Δ_F is higher during the previous prioritization period, then the new prioritization weight for throughput W_{TP} will be higher. *Note that* the constants in Eq. 4 help SATORI *limit the prioritization weight between 0.25 and 0.75 so as to not allow weights to be 0 and 1 which can be completely disadvantageous to a single goal and also cause a high degree of unstable weight fluctuations from one sampling interval to another.* We also experimented with favoring the well-performing objective in next period and found that, while reasonably effective, it can underperform the chosen design by approximately 5%.

To summarize, Fig. 6 shows how the equalization weight depends on how far along SATORI is within the equalization period. In contrast, the prioritization weight depends on the improvement in throughput and fairness during the prioritization period. The next task is to combine the equalization and prioritization weights to generate the final throughput (W_T) and fairness (W_F) weights. To combine the two, the key requirement is to gradually increase the importance of equalization weight as the end of the equalization period approaches to ensure the average of equal weights is maintained. This ensures that neither fairness nor throughput is favored

more than the other in the long run, as demonstrated by the following equations:

$$W_T = \frac{t_e}{T_E} W_{TE} + \left(1 - \frac{t_e}{T_E}\right) W_{TP} \quad (5)$$

$$W_F = \frac{t_e}{T_E} W_{FE} + \left(1 - \frac{t_e}{T_E}\right) W_{FP} \quad (6)$$

The influence of the equalization component of the weight is linearly increased as the end of the equalization period approaches. We choose a linear increase as opposed to a stronger form of increase (*e.g.,* exponential) to ensure that the equalization component does not completely overpower the weight near the end, which would result in temporary prioritization opportunities not being adequately exploited. To be aware of phase changes and workload mix changes, SATORI resets its baseline every equalization period by measuring the isolated performances of the co-located workloads. Be it a phase change or a change in the workload mixes, SATORI requires no further initialization. It adaptively configures itself to find the optimal configuration. Note that SATORI does not disqualify previously sampled configurations from re-evaluation (e.g., upon phase or workload-mix change), and reacts to the program phase changes as our evaluation confirms (Sec. V).

Implications of dynamically changing objective function of SATORI. SATORI *constructs a new objective function to exploit the temporal opportunities for maximizing competing goals (dynamic prioritization). It does so by dynamically changing the weight factors.* However, this naturally leads us to ask: what does this do to our BO process? This strategy can make different choices than what a traditional BO-based approach would have made. This is because "moving the goal post" can affect SATORI's belief about the search space and subsequent decisions. The key is to ensure that the expected deviation is bounded. This is why SATORI sets the upper and lower bound on the weight factors as 0.75 and 0.25, respectively, and keeps the equalization and prioritization periods bounded. Our evaluation (Sec. V) shows that this is effective in keeping the BO process controlled and near the traditionally expected behavior, while maximizing the objective function and hence, achieving better performance. We also point out that SATORI predicts a promising configuration to evaluate, but the "predicted expected improvement" of the configuration itself is not directly used. Instead, the configuration is indeed evaluated and compared against previous configuration evaluations. Therefore, the tweaked perception of the underlying BO proxy model does not affect the function and correctness. In fact, our evaluation (Sec. V) also confirms that the variation in observed performance for SATORI is similar to SATORI without dynamic prioritization, while achieving a higher average performance level. Also, the reached performance level remains stable, similar to SATORI without dynamic prioritization unless the phase related characteristics of the workloads change. Finally, we also note that SATORI needs to be run for a whole set of co-located workloads instead of one SATORI for each possible pair of workloads in a given set.

IV. Experimental Methodology

Testbed and Implementation. The experiments are conducted on a Intel Xeon Scalable platform with 14nm Skylake processors. The server has 10 physical cores with 2-way hyperthreading, 85W thermal design power, turbo-boosting, and shared last level cache. SATORI logic is implemented as a lightweight Python-based monitoring service leveraging *pqos* to measure performance, but does not require user intervention, profiling, or additional hardware support and deployable as a user-space system software. SATORI uses *taskset* for setting core affinity, Intel's Cache Allocation Technology (CAT) for LLC cache ways assignments, and Intel's Memory Bandwidth Allocation (MBA) feature for memory bandwidth partitioning via setting *Model Specific Registers (MSRs)*. The effect of SATORI's monitoring and decision making is included in all the results and characterized in the overhead (Sec. V).

TABLE I. PARSEC benchmarks [6] used in this study.

Blackscholes	Option pricing with Black-Scholes Partial Differential Eq.
Canneal	Simulated cache-aware annealing to optimize chip design
Fluidanimate	Fluid dynamics for animation with Smoothed
Freqmine	Frequent itemset mining
Streamcluster	Online clustering of an input stream
Swaptions	Pricing of a portfolio of swaptions

TABLE II. CloudSuite benchmarks [22] used in this study.

Data Analytics	Naive Bayes classifier on Wikipedia entries
Graph Analytics	Page ranking on Twitter data
In-memory Analytics	In-memory filtering of movie ratings
Media Streaming	Nginx server to stream videos
Web Search	Web search algorithm implementation

The instructions per second (IPS) of each workload is collected with *pqos* tool, sampled at a rate of 10 Hz. Methodological pitfalls associate with IPS due to synchronization are avoided for multi-threaded workloads [2], [20], [83]. Fixed-work methodology [29], [41], [66], [90] (using the same number of instructions/runs similar to other partitioning real-system partitioning works [41], [66], [90]) is employed and we have confirmed that these methodological choices do not negatively affect our results or conclusions. The default metrics used are Jain's fairness index and sum of instructions per second as these have been used by other competing techniques [66], [90]. Our evaluation confirmed that SATORI provides similar improvements over competing techniques for other commonly-used objectives metrics. This is because the core ideas and insights behind SATORI design is not metric-dependent and do not favor workloads with specific behavior. SATORI is implemented using Skopt and Hyperopt [5], which are distributed hyperparameter optimization libraries. An online evaluation of the isolated performance of each of the workloads is performed as a baseline for the measurement of throughput and fairness. Prioritization and equalization periods are configured to 1 second and 10 seconds, respectively, by default, but other values provide similar trends (Sec. V).

Workloads. SATORI is evaluated using a set of representative parallel, throughput-oriented workloads chosen from the PARSEC-3.0 benchmark suite [6] (listed in Table I). *Additionally, to demonstrate the effectiveness of* SATORI *on a diverse set of workloads,* SATORI *is also evaluated on the benchmarks from the CloudSuite benchmark suite [22] and the Exascale Computing Project (ECP) benchmark suite [56], [70] (Table II and III).* Unless otherwise mentioned, the presented results correspond to the PARSEC suite.

SATORI is evaluated by co-locating combinations of different applications in a job-mix. 5 workloads out of the 7 PARSEC workloads in Table I are chosen in a mix – resulting in

TABLE III. ECP benchmarks [56], [70] used in this study.

miniFE	Unstructured finite element solver
XSBench	Computational kernel of Monte Carlo neuronic
SWFFT	Fast Fourier transform for HACC (cosmology code)
AMG	Parallel algebraic multigrid solver for linear systems
Hypre	Scalable linear solvers and multigrid methods

a total of 21 ($\binom{7}{5} = 21$) different workload configurations. Relatively high degree of co-location is chosen as it presents more challenging situation for SATORI for optimization. 3 and 2 workloads out of the 5 CloudSuite workloads (Table II) and 5 ECP workloads, respectively, are chosen (Table III) – resulting in a 10 different workload configurations for each benchmark suite. SATORI uses the underlying BO scheme to update its resource allocation configuration every 0.1 seconds.

Competing Resource Partitioning Policies. We compare the performance of SATORI with the following policies:

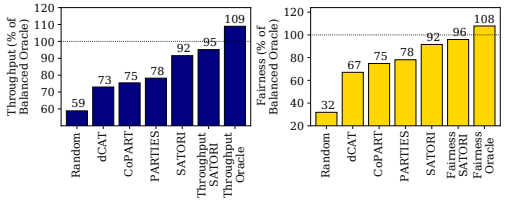
- **Random Search (Random)** samples a configuration stochastically from all possible configurations using a uniform distribution without repetition. The sampled configuration is updated every 0.1 second.

- **dCAT** improves throughput by dynamically partitioning a single shared resource (last-level cache) [90].

- **CoPart** primarily focuses on improving fairness by partitioning two resources – last-level cache and memory bandwidth using two separate FSMs for each resource [66].

- **PARTIES** primarily targets achieving Quality of Service (QoS) of co-located multiple latency-critical workloads [12]. We note that PARTIES is designed for meeting QoS targets of co-located latency-critical jobs. A consequential caveat is that PARTIES should not be necessarily expected to perform for the situation it was not designed for. However, we included PARTIES in our evaluation because the core idea behind PARTIES is partitioning shared resources using a gradient descent method – and, gradient descent method can be applied for resource partitioning among throughput-oriented workloads via iteratively optimizing for throughput to find the optimal resource partitioning configuration, although it was originally applied in the context for latency-critical workloads (our evaluation confirms the promise of gradient descent method in this new context). However, this method requires adjustment to simultaneously optimize for throughput and fairness. We have modified PARTIES to maximize both throughput and fairness, giving equal priority to both.

- **Brute-Force Search (Oracle).** Oracle refers to the brute-force, offline search strategy, which samples all possible configurations and selects the one which maximizes a given goal or a combination of goals. It is calculated every 0.1 seconds to account for the phase changes (Section II). This Oracle strategy has three different variants: *Throughput Oracle.* This solely maximizes throughput and ignores the fairness goal (*i.e.,* $W_T = 1$ and $W_F = 0$). *Fairness Oracle.* This solely maximizes fairness and ignores the throughput goal (*i.e.,* $W_T = 0$ and $W_F = 1$). *Balanced Oracle.* This puts an equal priority on maximizing for throughput and fairness (*i.e.,* $W_T = 0.5$ and $W_F = 0.5$). This strategy acts as the ceiling that SATORI aims to touch when maximizing for both throughput and fairness. Therefore, all results are presented as % of Balanced Oracle (*i.e.,* % distance from the theoretical optimal).



(a) Throughput **(b)** Fairness

Fig. 7. SATORI achieves better throughput and fairness than other techniques (averaged across different job mixes, PARSEC benchmarks).

- **Throughput and Fairness SATORI.** SATORI dynamically changes W_T and W_F during runtime to maintain a balance between the conflicting goals of maximizing throughput and fairness. *Throughput* SATORI refers to a specific variant of SATORI with $W_T = 1$ and $W_F = 0$ (*i.e.,* maximize only throughput). *Fairness* SATORI refers to a specific variant of SATORI with $W_T = 0$ and $W_F = 1$ (*i.e.,* maximize only fairness). These variants are evaluated and analyzed to quantify the limits of SATORI when optimizing a single goal.

V. Evaluation: Results and Analysis

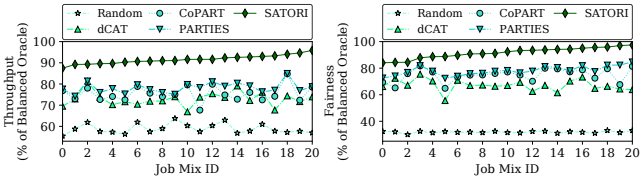
SATORI actively achieves both throughput and fairness goals simultaneously. SATORI provides near-optimal throughput and fairness with significant improvement over competing techniques. Fig. 7 quantifies the performance of all considered techniques (on average for 21 job mixes consisting of 5 jobs as percentage of the performance of Balanced Oracle). We make the following three observations:

First, SATORI outperforms competing techniques, performing 14% points better than the next best technique PARTIES for throughput and fairness. Notably, SATORI achieves 92% of the throughput and fairness of the Balanced Oracle.

Second, the throughput of SATORI with throughput as the only goal (Throughput SATORI) is higher than the throughput obtained by SATORI (which focuses on both goals), and close to the ideal Throughput Oracle. This result shows that SATORI can be configured to maximize a single goal (achieve high throughput, close to the Throughput Oracle). Fig. 7(b) shows similar trends for fairness.

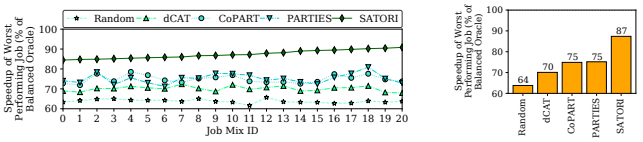
Finally, as expected, Random policy performs the worst, followed by dCAT, CoPart, and PARTIES. This confirms our intuition that controlling multiple resources actively achieves better performance (CoPart>dCAT). Also, our results show that a gradient descent approach (PARTIES) is quite useful for improving the throughput and fairness metrics.

Next, we dig deeper to quantify the effectiveness of SATORI for individual job mixes. We plot the throughput and fairness for all of the 21 job mixes individually in Fig 8. *We observe that* SATORI *provides better performance than competing techniques consistently for all job mixes for both throughput and fairness.* In some cases, SATORI can have up to 20% points better throughput and 10% points better fairness than



(a) Throughput **(b)** Fairness

Fig. 8. SATORI achieves better throughput and fairness than other techniques for every job mix (PARSEC benchmarks). The results are sorted in ascending order of SATORI's performance.



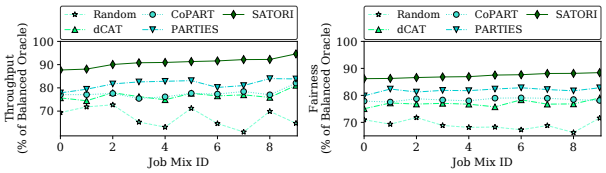
(a) Worst Case Speedup Per Job Mix **(b)** Average of Mixes

Fig. 9. The worst performing job in a mix performs better with SATORI than other techniques across (a) all 21 job mixes and (b) on average.

the next best technique PARTIES, but never worse than competing techniques. In fact, Fig. 9 shows that the worst-performing job in a mix performs much better with SATORI than with other techniques for every single job mix, on average achieving 87% of the Balanced Oracle performance. This result shows that benefits of SATORI are not dependent on specific characteristics found in specific workloads or job mixes (e.g., CPU-intensive, memory-intensive, cache-sensitive, etc.). SATORI works effectively across a variety of workload mixes.

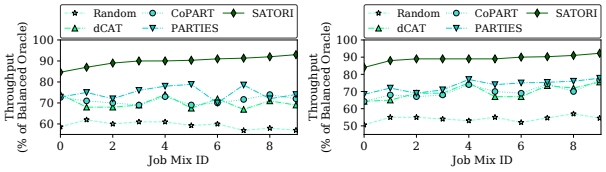
We analyzed the reasons for variation in SATORI's benefits across different job mixes. The mix consisting of canneal, freqmine, streamcluster, swaptions, and vips (among PARSEC benchmark suite mixes) has the highest throughput gain (job mix 20) and replacing freqmine with fluidanimate results in the lowest throughput gain (job mix 0). The reason is the high compute-resource (number of cores) sensitivity of fluidanimate (animation of fluid particles). Another example, Job mix 17 (canneal, freqmine, streamcluster, swaptions, vips) has high throughput gain, however replacing swaptions and streamcluster with blackscholes and fluidanimate results in low throughput gain (job mix 3) as blackscholes and fluidanimate both contend for same resource – memory bandwidth.

To further test the effectiveness of SATORI across different types of workloads, we evaluated different job mixes for CloudSuite and ECP benchmarks (Fig. 10 and 11). We again observe similar performance trends – SATORI significantly outperforms all competing techniques and consistently outperforms across all job mixes for these benchmarks too. CloudSuite and ECP benchmarks were chosen because they represent a diverse set of workloads beyond PARSEC benchmarks – for example, data analytics kernels, graph algorithm, and scientific computing applications (e.g., linear solvers and physical simulation kernels). Our results show that SATORI provides consistent improvement both in terms of throughput and fairness for all job mixes for both CloudSuite and ECP benchmark suites.



(a) Throughput **(b)** Fairness

Fig. 10. SATORI achieves better throughput and fairness than other techniques for every job mix (CloudSuite benchmarks). The results are sorted by SATORI's performance.

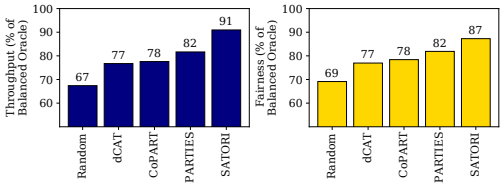


(a) Throughput **(b)** Fairness

Fig. 11. SATORI achieves better throughput and fairness than other techniques for every mix of two out of five co-located ECP benchmarks. The results are sorted by SATORI's performance.

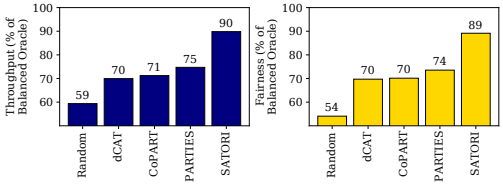
The aggregate results indicate that in terms of throughput, SATORI outperforms the next best technique (PARTIES) by 9% and 15% for CloudSuite and ECP benchmark suites, respectively (Fig. 12 and Fig. 13). In terms of fairness, SATORI outperforms the next best technique (PARTIES) by 5% and 15% for CloudSuite and ECP benchmark suites, respectively. To understand the performance behavior better, we looked at different job-mixes. SATORI achieves the lowest performance for mix 0 of miniFE and SWFFT (ECP benchmark suite). The main reason for this is that miniFE has intensive compute (high IPC and FLOP rate) and last-level cache (high L1 miss-rate) requirements, which makes it difficult to co-locate with SWFFT which has an equally high LLC requirement. On the other hand, SATORI achieves the best performance for mix 9 of AMG and Hypre. Both of them have similar resource requirements for all resources, which while making co-location difficult, also makes the search space easier to navigate for SATORI, resulting in near-optimal results. For all job mixes representing a wide variety of characteristics, SATORI performs better than competing techniques by more than 10% points in terms of both throughput and fairness in most cases.

Next, we investigate the source of SATORI's benefits. In particular, we tested whether SATORI can outperform dCAT and CoPart only because it partitions more resources (dCAT partitions the LLC ways and CoPart partitions the memory bandwidth and LLC ways). *Interestingly, our results revealed that* SATORI*'s approach is more effective than dCAT and CoPart even when partitioning only one or two resources.* When SATORI only partitions the LLC ways, it performs 4% points better than dCAT on throughput and 5% points better on fairness. Further, when SATORI only partitions the LLC ways and memory bandwidth, it performs 7% points better than CoPart on throughput and 4% points better on fairness. Therefore, the higher benefits of SATORI are not merely a result of operating in a multi-resource design space. Note that



(a) Throughput **(b)** Fairness

Fig. 12. SATORI achieves better throughput and fairness (averaged across different job mixes, CloudSuite benchmarks).



(a) Throughput **(b)** Fairness

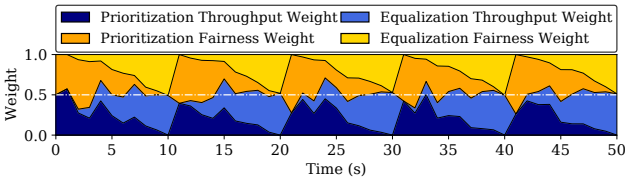
Fig. 13. SATORI achieves better throughput and fairness (averaged across different job mixes, ECP benchmarks).

SATORI and PARTIES partition the same set of resources.

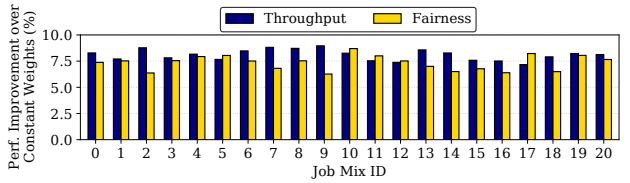
Why does SATORI work so well? Next, we look at the internal functioning of SATORI to provide insight into its performance gain. Fig. 14(a) provides a closer look at how SATORI dynamically re-balances the prioritization and equalization components of fairness and throughput weights. The two blue regions combined indicate the overall throughput weight, and the two yellow regions combined indicated the overall fairness weight. We observe that overall throughput and fairness weights can deviate by up to 50% from the equal weights mark of 0.5 due to temporary prioritization of one goal over the other. However, the equalization component ensures that for the equalization period (a configurable parameter, set to 10 seconds in this example), the weights are 0.5 on average. This is why the equalization component gets more dominant as we approach the end of the equalization window of different lengths (10s, 20s, 30s, etc.).

To understand the effectiveness of the dynamic weight prioritization strategy, we compare SATORI against SATORI implemented with static 0.5 weights for fairness and throughput (Fig. 14(b)). We observe that dynamic prioritization of weights provides up to 10% additional benefit over the static weights strategy for both fairness and throughput. More importantly, dynamic weight prioritization opportunities and benefits are present in all 21 job mixes and lead to improvements in both the goals (*i.e.,* not limited to improving only one goal: throughput or fairness). This indicates that SATORI successfully leverages short-term prioritization of one goal over the other to improve performance significantly.

Next, Fig. 15 helps us explain that SATORI outperforms competing techniques by choosing better configurations. To this end, we quantify the proximity of the configuration obtained by SATORI to the configuration obtained by the Balanced Oracle to demonstrate its near-optimality and comparison with PARTIES. We calculate the distance between the configuration allocated by

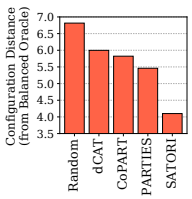


(a) Weight Re-balancing During Runtime

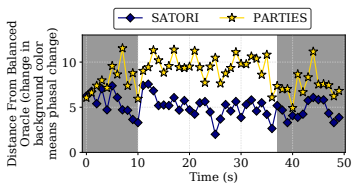


(b) Impact of Dynamic Weights

Fig. 14. (a) SATORI dynamically re-balances equalization and prioritization throughput and fairness weights. (b) Dynamic weight re-balancing improves performance.



(a) Config. Distance



(b) SATORI and PARTIES Config. Distance

Fig. 15. (a) The configurations set by SATORI are the closest to the Balanced Oracle. (b) SATORI dynamically changes configurations better than PARTIES as the phase changes.

different techniques and the configuration allocated by Balanced Oracle. In this example, the configuration has 15 dimensions (5 jobs $\times$ 3 resources), so calculating the distance between two configurations is the same as calculating the Euclidean distance between two vectors of 15 dimensions. Note that the maximum possible distance is 13 in this scenario. Fig. 15(a) shows that SATORI indeed achieves the closest proximity to the Balanced Oracle (averaged over the runtime of a job mix and across different job mixes), with other techniques having at least 1.3$\times$ more distance of SATORI. Fig. 15(b) shows that SATORI is able to find better configuration than PARTIES as the phase changes – distance from Oracle for SATORI is lower than PARTIES, although the optimal configuration changes over time and changes frequently. SATORI outperforms because of both the capabilities: better search compared to gradient descent and dynamic prioritization – the magnitude from individual contribution can range from 2% - 10% depending upon the application, workload mix and temporal phase.

SATORI is not sensitive to the prioritization period and the equalization period within a wide range, and does not require investment in tuning efforts to yield benefits. Fig. 16 shows the effect of these factors on SATORI's performance. It shows the sensitivity of SATORI's performance to prioritization and equalization periods (the two most essential and tunable parameters of SATORI). While the sensitivity is not high, we observe that generally, the throughput and fairness performance degrades as the period length increases. This is because a

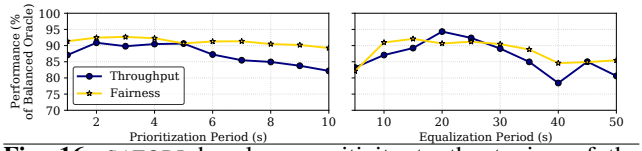


Fig. 16. SATORI has low sensitivity to the tuning of the prioritization period and the equalization period. It does not require large tuning efforts to achieve near-optimal results.

longer prioritization period provides less opportunity to make precise dynamic decisions based on very short phases. That is, if the period is too long, the decisions are made based on performance during longer phases. And, a longer equalization period provides less opportunity to equalize the two goals, thus causing the performances to degrade. However, this only happens for a very long prioritization period (> 5 seconds) and equalization period (> 30 seconds). Overall, SATORI does not require a large amount of tuning effort to set these parameters. Parameter values chosen in a reasonable range perform similarly.

SATORI's design of the new objective function (*moving goal post*) helps it extract higher performance (throughput and fairness), but does not force the underlying BO model to behave unexpectedly. Recall that SATORI constructs a new objective function that changes the objective function itself over time (dynamic prioritization of goals). Using an example, we show that this feature, although non-conventional, keeps the underlying BO engine operations in the neighborhood of the expected BO operations, but yield higher performance. The job mix used for illustration includes a group of five PARSEC benchmarks (blackscholes, canneal, fluidanimate, freqmine, streamcluster) and covers the initial portion of a long-running experiment. Similar observations are drawn from other job mixes, but not shown for brevity.

First, Fig. 17(a) shows the value of the objective function over time. SATORI achieves higher value for the objective function over time compared to the SATORI without prioritization. This is expected since the goal of SATORI is to indeed maximize the objective function – and that yields better performance. However, a natural inquiry is: does this increase the variance in decision that the underlying BO makes? More specifically, does the estimation of SATORI about the proxy model change more drastically than the estimate of SATORI without prioritization? If the difference in the estimations is varying erratically over time, then it can potentially lead to corresponding increased variability in the observed performance.

Fig. 17(b) shows the percentage change in the values of the proxy models (mean of absolute difference of estimate of each configurations from one iteration to the next) in SATORI and SATORI without dynamic prioritization. Fig. 17(b) confirms that the range of percentage change in the proxy model value is similar in SATORI and SATORI without dynamic prioritization. The cross-overs between the two curves occur because the exploration and exploitation phase of SATORI and SATORI without dynamic prioritization are not in synchronization. The final inquiry is: does SATORI result in more variation in the observed performance due to its new objective function design (changing weight factors)? Fig. 17(a) earlier showed that new objective function design helps it maximize the objective function better. Our results (Fig. 18) demonstrate that the

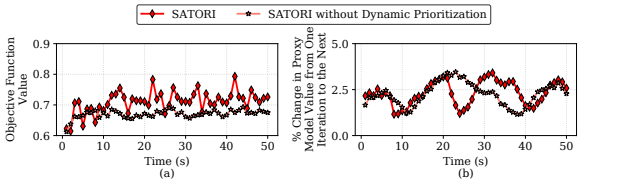


Fig. 17. New objective function construction in SATORI improves the quality of the solution by obtaining (a) higher objective function values, but without (b) unexpected changes in the underlying BO operations.

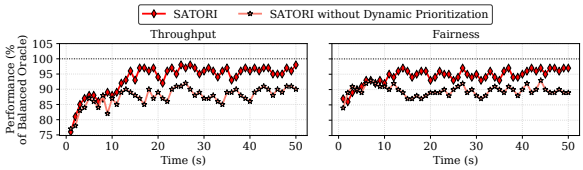


Fig. 18. Variation in the observed performance for both goals is similar for SATORI and SATORI without prioritization.

variation in the observed performance for SATORI is similar to that of SATORI without dynamic prioritization. SATORI curve is always above the SATORI without dynamic prioritization curve and closer to the Oracle, but the changes over time are similar in nature and within the expected region – confirming that bounding the weight factor is also helpful (Sec. III-B). Finally, Fig. 19 shows a sample snapshot to demonstrate that prioritizing the weaker goal empirically has better potential to improve the benefits for both goals over a time period, instead of prioritizing the stronger goal (as discussed in Sec. III).

SATORI's relative improvement increases with higher co-location degree (scalability). As the co-location degree increases from three to seven applications, the %-point difference between SATORI and PARTIES monotonically increases for both throughput and fairness goals (% point difference is 8%, 11%, 13%, 13% and 15% for 3, 4, 5, 6, and 7 co-located applications, respectively). This is because as the co-location degree increases, finding the optimum becomes more challenging due to the larger search space with more number of local maxima, and gradient-descent methods tend to get stuck in a local maximum more than SATORI.

SATORI is practical for real-systems and incurs low overhead. First, we note that SATORI's overhead is not in the critical path of job mix execution as jobs continue to execute while SATORI is making its decisions. Jobs continue to execute using their previous resource allocation configuration until SATORI generates a new decision and implements it. SATORI's most time-consuming component is its BO engine, which includes a GP model update and acquisition function evaluation. We measured that in a 100ms sampling interval, all BO related tasks take only 1.2ms on average.

Further, we note that SATORI is optimized not to be active and incurring computation related overhead at all times. It is invoked only when the performance of a specific job changes significantly or the job mix changes. This ensures that 1.2 ms overhead is incurred only when needed. Further optimizations such as avoiding frequent updates to the GP model after the optimal configuration detection saves GP model overheads that can potentially increase the computational time. In terms of

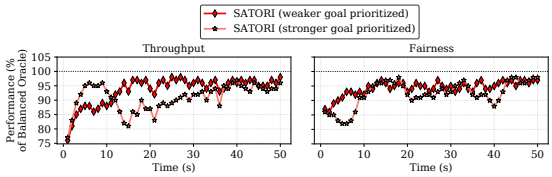


Fig. 19. Prioritizing the weaker performing goal helps SATORI to reach higher levels of throughput and fairness.

computation interference, our experiments reveal that SATORI only executes about 1% of the instructions executed by the job mix on an average. Note that all our performance results related to SATORI presented in this paper automatically include this overhead. BO can be sensitive to the initial configuration set used at the search start; final outcome quality may vary by 1-3%. SATORI mitigates this challenge by building a reasonable set of "good" configurations (e.g., equal resource partitions, less imbalance in partition share across resources for a job) instead of starting from random configurations.

VI. Related Work

Besides multiple state-of-the-art techniques used in our evaluation, most prior works have focused on managing/partitioning one resource to meet one goal such as improving utilization, energy consumption, etc. [8]–[10], [17], [19], [30], [34], [36], [40], [41], [55], [63], [65], [67], [71], [76], [78], [81], [84], [87], [89], [90], [93]. Some techniques can be customized to meet different goals, but not all goals simultaneously. Many recent works have leveraged the new hardware capabilities of LLC ways and memory bandwidth partitioning, power-capping, etc., to use multiple resources to meet a single goal [12], [18], [39], [44], [49], [50], [52], [61], [66], [68], [74], [94].

Our recent work designed for co-locating latency critical jobs, CLITE [68], also takes a BO-based approach to partition shared multi-core resources. CLITE outperforms PARTIES when co-locating multiple latency critical jobs. But, when applied in SATORI's context of multiple throughput-oriented co-located workloads with two competing objectives (fairness and throughput), CLITE performs similar to PARTIES and underperforms SATORI by a similar margin as PARTIES. This is expected since both CLITE and PARTIES are not designed for this problem context; they do not actively and simultaneously control competing objectives, and do not employ the insight of dynamic prioritization of goals.

REF technique, proposed by Zahedi et al. [92], is the first proposal to demonstrate the use of Cobb-Douglas utility for modeling performance, and uses game theory to fairly allocate resources among multiple applications. REF primarily focuses on improving fairness and does not actively maximize throughput (unlike SATORI), but it provides multiple desirable properties including sharing incentives, envy freeness, and Pareto efficiency. In contrast to SATORI, REF requires prior information or profiling about the performance of each application under all possible resource allocations using Cobb-Douglas utility function – while promising, it is often not practical in typical cloud computing scenarios. A game-theoretic approach is more suitable where the set of co-located applications is fixed and tenants need assurance about envy freeness and sharing incentive [51], [92].

Wang et al. [82] proposed, ReBudget, a market-based resource allocation scheme for allocating multi-core resources. ReBudget proposes a novel iterative bidding-pricing procedure where users solve an optimization problem using hill-climbing to maximize their local utility and bid for resources. Then, the market determines their allocations based on multiple rounds of bidding. Extensive simulation-based evaluation, leveraging hardware-support for utility calculation, demonstrates the value of iterative bidding ideas. However, quickly achieving market equilibrium with multiple applications and more than two shared architectural resources is challenging. In contrast, SATORI employs a BO-based approach that works on commodity hardware and exploits the dynamic prioritization of competing goals: fairness and throughput. Combining game-theoretic approaches such as REF and ReBudget with SATORI can provide further opportunities for improving fairness and throughput.

Ghodsi et al. [26] proposed dominant resource fairness (DRF) to maintain fairness among applications by providing an equal share of the most dominant resource to each application. Chowdhury et al. [14] proposed HUG to improve DRF by considering elastic and correlated demands among network links. However, they still assume that the demand vector is supplied by the user, and they do not consider correlated utility – which is critical in the case for multi-core resource partitioning (e.g., memory bandwidth allocation can affect the utility of allocated cache ways) and SATORI's BO-based approach captures these effects. Also, these approaches require a user to define a demand vector of resource share, which may not be readily available or be determined for co-located applications on a CMP. SATORI assumes no input from the user. SATORI does not make such static decisions and instead adapts to the dynamically changing characteristics to utilize the shared resources more efficiently.

VII. Conclusion

Previous works do not *actively and simultaneously* control partitioning knobs while achieving multiple goals. SATORI is the only technique to *actively and simultaneously* control multiple CMP architectural resources to achieve multiple goals. SATORI can effectively handle computing cores, LLC ways, memory bandwidth, and power-cap resources to achieve an optimal balance among throughput and fairness – which also complements recently published approaches for energy-efficiency under co-location based on economics, collaborative filtering and machine learning [43], [59], [60], [62], [64].

Acknowledgment. We are thankful to anonymous reviewers for providing thoughtful feedback despite the pandemic challenges. We are thankful for the support from NSF Award 1910601 and 1753840, and Northeastern University.

References

[1] Intel 64 and IA-32 Architectures Software Developer's Manual.
[2] Alaa R Alameldeen and David A Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.
[3] Omid Alipourfard et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, pages 469–482, 2017.
[4] Ali Bakhoda, John Kim, and Tor M Aamodt. Throughput-effective on-chip networks for manycore accelerators. In *2010 MICRO*, pages 421–432. IEEE, 2010.
[5] James Bergstra et al. Hyperopt: A Python Library for Model Selection and Hyperparameter Optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[6] Christian Bienia and Kai Li. Parsec 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, volume 2011, 2009.

[7] Tarannum Bloch et al. Understanding Live Migration Techniques Intended for Resource Iinterference Minimization in Virtualized Cloud Environment. In *Big data analytics*, pages 487–497. Springer, 2018.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*, pages 285–300, 2014.

[9] Alex D Breslow et al. The case for colocation of hpc workloads. *Concurrency and Computation: Practice and Experience Preprint*, 2012.

[10] Alex D Breslow et al. Enabling fair pricing on hpc systems with node sharing. In *Supercomputing (SC)*, pages 1–12, 2013.

[11] Jian Chen and Lizy Kurian John. Predictive Coordination of Multiple On-Chip Resources for Chip Multiprocessors. In *Proceedings of the international conference on Supercomputing*, pages 192–201. ACM, 2011.

[12] Shuang Chen, Christina Delimitrou, and José 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*, pages 107–120. ACM, 2019.

[13] Xi E Chen and Tor M Aamodt. A first-order fine-grained multithreaded throughput model. In *2009 HPCA*, pages 329–340. IEEE, 2009.

[14] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, 2016.

[15] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[16] Ewa Deelman, Christopher Carothers, Anirban Mandal, Brian Tierney, Jeffrey S Vetter, Ilya Baldin, Claris Castillo, Gideon Juve, Dariusz Król, Vickie Lynch, et al. PANORAMA: An Approach to Performance Modeling and Diagnosis of Extreme-Scale Workflows. *The International Journal of High Performance Computing Applications*, 31(1):4–18, 2017.

[17] Christina Delimitrou and Christos Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Transactions on Computer Systems (TOCS)*, 31(4):12, 2013.

[18] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144. ACM, 2014.

[19] Nosayba El-Sayed et al. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 HPCA*, pages 104–117. IEEE, 2018.

[20] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3):42–53, 2008.

[21] Stijn Eyerman, Pierre Michaud, and Wouter Rogiest. Multiprogram Throughput Metrics: A Systematic Approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):34, 2014.

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[23] Yu Gan and Christina Delimitrou. The Architectural Implications of Cloud Microservices. *IEEE CAL*, 2018.

[24] Yu Gan et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS*, pages 3–18. ACM, 2019.

[25] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *ASPLOS*, pages 19–33. ACM, 2019.

[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[27] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th OSDI*, pages 65–80, 2016.

[28] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE, 2019.

[29] Andrew Hilton, Neeraj Eswaran, and Amir Roth. Fiesta: A sample-balanced multi-program workload methodology. *Proc. MoBS*, 2009.

[30] Derek R Hower, Harold W Cain, and Carl A Waldspurger. PABST: Proportionally Allocated Bandwidth at the Source and Target. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 505–516. IEEE, 2017.

[31] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent Freeh. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *CLOUD, 2018*.

[32] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. Scout: An Experienced Guide to Find the Best Cloud Configuration. *arXiv preprint arXiv:1803.01296*, 2018.

[33] CAT Intel. Improving Real-Rime Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[34] Ravi Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266. ACM, 2004.

[35] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput Fairness Index: An Explanation. In *ATM Forum contribution*, volume 99, 1999.

[36] Vimalkumar Jeyakumar et al. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, pages 297–311, 2013.

[37] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[38] Seungmin Kang et al. Towards Workload-Aware Virtual Machine Consolidation on Cloud Platforms. In *6th ICCUIMC Conference*, page 45. ACM, 2012.

[39] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[40] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610. IEEE, 2015.

[41] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGPLAN Notices*, 49(4):729–742, 2014.

[42] Kenji Kawaguchi, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Bayesian Optimization with Exponential Convergence. In *Advances in neural information processing systems*, pages 2809–2817, 2015.

[43] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A Shoemaker, Christina Delimitrou, and David H Albonesi. Cuttlesys: Data-driven resource management for interactive services on reconfigurable multicores. In *MICRO*. IEEE, 2020.

[44] Bin Li, Li Zhao, Ravi Iyer, Li-Shiuan Peh, Michael Leddige, Michael Espig, Seung Eun Lee, and Donald Newell. CoQoS: Coordinating QoS-Aware Shared Resources in NoC-Based SoCs. *Journal of Parallel and Distributed Computing*, 71(5):700–713, 2011.

[45] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M Shipman, and Chita R Das. D-Factor: A Quantitative Model of Application Slow-Down in Multi-Resource Shared Systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):271–282, 2012.

[46] Xing Lin et al. Towards Fair Sharing of Block Storage in a Multi-Tenant Cloud. In *HotCloud*, 2012.

[47] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding How off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[48] Fang Liu and Yan Solihin. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 37–48. ACM, 2011.

[49] Rose Liu et al. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 10–10, 2009.

[50] Yanpei Liu et al. SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers. In *ISCA*, 2014.

[51] Qiuyun Llull, Songchun Fan, Seyed Majid Zahedi, and Benjamin C Lee. Cooper: Task colocation with cooperative games. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 421–432. IEEE, 2017.

[52] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[53] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. Improving Spark Application Throughput via Memory Aware Task Co-Location: A Mixture of Experts Approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 95–108. ACM, 2017.

[54] Jason Mars et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *MICRO*. ACM, 2011.

[55] David Meisner, Brian T Gold, and Thomas F Wenisch. PowerNap: Eliminating Server Idle Power. In *ACM sigplan notices*, volume 44, pages 205–216. ACM, 2009.

[56] Paul Messina. The Exascale Computing Project. *Computing in Science & Engineering*, 19(3):63–67, 2017.

[57] Takashi Miyazaki, Issei Sato, and Nobuyuki Shimizu. Bayesian Optimization of HPC Systems for Energy Efficiency. In *HiPC*, pages 44–62. Springer, 2018.

[58] Vivek Nair et al. Transfer Learning with Bellwethers to Find Good Configurations. *arXiv preprint arXiv:1803.03900*, 2018.

[59] Iyswarya Narayanan, Adithya Kumar, and Anand Sivasubramaniam. Pocolo: Power optimized colocation in power constrained environments. In *2020 IISWC*, pages 1–12. IEEE, 2020.

[60] Iyswarya Narayanan and Anand Sivasubramaniam. Mediating power struggles on a shared server. In *2020 ISPASS*. IEEE.

[61] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *EuroSys*, pages 237–250. ACM, 2010.

[62] Mehrzad Nejat et al. Coordinated management of processor configuration and cache partitioning to optimize energy under qos constraints. In *2020 IPDPS*, pages 590–601. IEEE, 2020.

[63] Konstantinos Nikas et al. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In *ICPP*, page 15. ACM, 2019.

[64] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

[65] Dejan Novaković and other. Deepdive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *USENIX ATC*, 2013.

[66] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 10. ACM, 2019.

[67] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. HyPart: A Hybrid Technique for Practical Memory Bandwidth Partitioning on Commodity Servers. In *PACT 2018*.

[68] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[69] Ravishankar Rao and Sarma Vrudhula. Efficient online computation of core speeds to maximize the throughput of thermally constrained multi-core processors. In *2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 537–542. IEEE, 2008.

[70] David Richards et al. Quantitative performance assessment of proxy apps and parents. *LLNL Tech. Rep*, 2018.

[71] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

[72] Alexandre Scotto Di Perrotolo. A Theoretical Framework for Bayesian Optimization Convergence, 2018.

[73] Vicent Selfa, Julio Sahuquillo, Crispín Gómez, and María E Gómez. Methodologies and Performance Metrics to Evaluate Multiprogram Workloads. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 150–154. IEEE, 2015.

[74] Akbar Sharifi et al. METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management. In *ACM SIGMETRICS*, pages 13–24. ACM, 2011.

[75] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2015.

[76] Nikolay A Simakov et al. A quantitative analysis of node sharing on hpc clusters using xdmod application kernels. In *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, pages 1–8, 2016.

[77] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[78] Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 517–528. ACM, 2009.

[79] Prashanth Thinakaran et al. Phoenix: A Constraint-Aware Scheduler for Heterogeneous Datacenters. In *2017 ICDCS*. IEEE.

[80] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[81] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F Martínez. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *HPCA*, pages 121–132. IEEE, 2017.

[82] Xiaodong Wang and José F Martínez. Rebudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. *ACM SIGPLAN Notices*, 51(4):19–32, 2016.

[83] Thomas F Wenisch, Roland E Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. Simflex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[84] Joseph P White et al. An analysis of node sharing on hpc clusters using xdmod/tacc_stats. In *Proceedings of the 2014 Extreme Science and Engineering Discovery Environment*, pages 1–8, 2014.

[85] Carl Witt, Marc Bux, Wladislaw Gusew, and Ulf Leser. Predictive Performance Modeling for Distributed Batch Processing using Black Box Monitoring and Machine Learning. *Information Systems*, 2019.

[86] Carole-Jean Wu and Margaret Martonosi. A Comparison of Capacity Management Schemes for Shared CMP Caches. In *7th WDDD Workshop*, volume 15, pages 50–52. Citeseer, 2008.

[87] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *EuroSys*, page 13. ACM, 2018.

[88] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning. In *2014 HPCA*.

[89] Yuejian Xie and Gabriel H Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

[90] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dCat: Dynamic Cache Management for Efficient, Performance-Sensitive Infrastructure-as-a-Service. In *Proceedings of the Thirteenth EuroSys Conference*, page 14. ACM, 2018.

[91] Hailong Yang et al. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. *ACM SIGARCH Computer Architecture News*, 41(3):607–618, 2013.

[92] Seyed Majid Zahedi and Benjamin C Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. *ACM SIGPLAN Notices*, 49(4):145–160, 2014.

[93] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys*, pages 379–391. ACM, 2013.

[94] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.