MAPPER: Managing Application Performance via Parallel Efficiency Regulation*

SHARANYAN SRIKANTHAN†, SAYAK CHAKRABORTI, PRINCETON FERRO†, and SANDHYA DWARKADAS, Department of Computer Science, University of Rochester, Rochester, NY, USA

State-of-the-art systems, whether in servers or desktops, provide ample computational and storage resources to allow multiple simultaneously executing potentially parallel applications. However, performance tends to be unpredictable, being a function of algorithmic design, resource allocation choices, and hardware resource limitations.

In this article, we introduce MAPPER, a manager of application performance via parallel efficiency regulation. MAPPER uses a privileged daemon to monitor (using hardware performance counters) and coordinate all participating applications by making two coupled decisions: the degree of parallelism to allow each application to improve system efficiency while guaranteeing quality of service (QoS), and which specific CPU cores to schedule applications on. The QoS metric may be chosen by the application and could be in terms of execution time, throughput, or tail latency, relative to the maximum performance achievable on the machine. We demonstrate that using a normalized parallel efficiency metric allows comparison across and cooperation among applications to guarantee their required QoS. While MAPPER may be used without application or runtime modification, use of a simple interface to communicate application-level knowledge improves MAPPER's efficacy. Using a QoS guarantee of 85% of the IPC achieved with a fair share of resources on the machine, MAPPER achieves up to 3.3× speedup relative to unmodified Linux and runtime systems, with an average improvement of 17% in our test cases. At the same time, MAPPER violates QoS for only 2% of the applications (compared to 23% for Linux), while placing much tighter bounds on the worst case. MAPPER relieves hardware bottlenecks via task-to-CPU placement and allocates more CPU contexts to applications that exhibit higher parallel efficiency while guaranteeing QoS, resulting in both individual application performance predictability and overall system efficiency.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Software and its engineering → Scheduling**; **Multiprocessing/multiprogramming/multitasking;**

Additional Key Words and Phrases: Parallel systems, performance counters, sharing-aware scheduler, resource contention

*New article, not an extension of a conference paper. MAPPER source code is available at https://github.com/URCS-systems/MAPPER. This work was supported in part by U.S. National Science Foundation grants CNS-1319353, CNS-1618497, and CNS-1900803; by a Google Faculty Research Award; and by a gift from Futurewei.

†Sharanyan Srikanthan and Princeton Ferro were at University of Rochester when participating in the work.

Authors' address: S. Srikanthan, S. Chakraborti, P. Ferro, and S. Dwarkadas, Department of Computer Science, University of Rochester, Rochester, NY, 14627; emails: sharanyansrikanth@gmail.com, schakr11@cs.rochester.edu, princetonferro@gmail.com, sandhya@cs.rochester.edu.

1544-3566/2022/03-ART18 $15.00
https://doi.org/10.1145/3501767

ACM Reference format:
Sharanyan Srikanthan†, Sayak Chakraborti, Princeton Ferro†, and Sandhya Dwarkadas. 2022. MAPPER: Managing Application Performance via Parallel Efficiency Regulation*. *ACM Trans. Arch. Code Optim.* 19, 2, Article 18 (March 2022), 26 pages.
https://doi.org/10.1145/3501767

1 INTRODUCTION

Present-day computer systems, whether in the cloud, the datacenter, or in desktops, have an abundance of compute, memory, storage, and communication resources. These systems are typically able to host multiple, simultaneously active, potentially parallel applications successfully. However, overall system utilization efficiency is often low especially when applications have low parallel efficiency, or is achieved at the expense of individual application performance [13, 21, 22, 44, 45]. Extracting the best performance from a given system even when an individual parallel application has exclusive use of the system often requires considerable user expertise [1, 34].

The parallel efficiency of an application (speedup relative to the use of a single hardware context normalized by the number of hardware contexts employed) is impacted by algorithmic design and by system capability and management. Algorithmic design can result in issues such as load imbalance, contention for synchronization, inter-task dependencies, and data sharing. The impact of these factors on performance is dependent on system capability and management. System capability is a function of available resources. Demand for resources must be managed at the system level. Resource demand from both individual parallel applications and from other simultaneously executing applications can result in resource contention and saturation. Resource contention and saturation often lead to performance degradation due to the resulting queuing delays. Thus, system-level resource management, the focus of this article, can have a significant impact on both overall system efficiency and individual application performance.

Utilizing resources efficiently and fairly has received considerable attention [6, 8, 14, 20, 21]. However, transparently achieving good performance on any given system for individual applications (performance portability) while simultaneously ensuring overall system resource utilization efficiency has remained an elusive goal. In this article, we present MAPPER,[1] a system that manages application performance via parallel efficiency regulation to achieve the above simultaneous goals without the need for user-level expertise and understanding of system properties. MAPPER monitors and coordinates all participating applications by making two coupled decisions: how much parallelism to afford to each application, and which specific CPU cores to schedule applications on. We focus on providing both good individual application performance (using a well-defined notion of fairness and quality of service) and overall system-wide resource efficiency.
MAPPER contains four distinct elements:

- Hardware performance counter-based characterization of application resource needs and bottlenecks;
- Prioritized compute resource allocation to applications based on resource needs and quality of service/fairness guarantees;
- Task-to-compute-resource mapping (a refinement of the heuristics developed in our sharing-aware mapping work [44, 45]) to alleviate bottlenecks and reduce compute, bandwidth, and memory contention, bandwidth utilization, and latency of data access;

[1]Early results were presented as a poster at the 2019 Symposium on the Principles and Practice of Parallel Programming [46].

- An optional bidirectional interface that provides the resource management system with application-specific progress metrics and provides the application runtime with the number of hardware contexts allocated by MAPPER that the runtime can use to better control the degree of concurrency.

MAPPER monitors applications using *perf* to access relevant low-overhead hardware performance counters, which enables detection, identification, and separation of the causes behind poor performance and parallel efficiency. The performance analysis then informs the policies used in resource allocation to meet combined application and system-wide efficiency goals. Resource allocation is effected by controlling the number and location of hardware execution contexts allocated to an application using Linux cgroups.

In our evaluation, we define **quality of service (QoS)** in terms of a percentage of an application's performance when using a fair share (static resource allocation based on the number and priority of applications) of execution contexts. Importantly, we derive a normalized efficiency metric that allows comparisons across and cooperation among applications to meet individual application QoS and overall system efficiency requirements.

Finally, MAPPER also provides an interface for direct communication with an application's runtime to dynamically effect task control. An application-defined progress metric may be provided through the runtime to MAPPER, allowing MAPPER to more accurately relate application progress with resource allocation and usage. Conversely, MAPPER allows the runtime to query for information on the number of allocated hardware contexts, which the runtime may use to control the degree of concurrency (the number and size of tasks created) of the application based on application scalability. We choose the OpenMP application programming interface to demonstrate the benefits of task control.

Our results using MAPPER demonstrate performance improvements for both standalone and multi-programmed workloads. Using a QoS guarantee of 85% of the IPC achieved when an application receives its fair share of resources, MAPPER achieves speedups relative to Linux of up to 3.3× and a geometric mean speedup of 1.17× across all workloads. When combined with parallel runtime cooperation (specifically, OpenMP), MAPPER achieves a mean speedup of 1.44× compared to Linux. At the same time, MAPPER violates QoS for only 2% of applications in our test suite (compared to 23% for Linux), with the worst case achieving 81% (compared to 9% for Linux) of fair share performance for individual applications. MAPPER achieves this efficiency and QoS in a manner transparent to the application developer and user.

The rest of the article is organized as follows: Section 2 discusses some of the prior work in this domain, which we organize thematically. In Section 3, we introduce our system design and discuss implementation details. Section 4 presents the results of our evaluation, and we conclude with Section 5.

2 PRIOR WORK

Offline approaches: Several works have examined the efficiency of offline performance prediction [6, 8, 14, 20, 25]. Pandia [20] creates an offline workload model requiring multiple runs of the target application to identify the impact on parallel efficiency of the application's resource demand, parallel fraction, data sharing, load balancing, and need for computational resources. The methodology does not consider multiple applications sharing resources and cannot adapt to workloads with heterogeneous, dynamic phases of computation or data-dependent behavior. ReSense [14] uses offline characterization to understand application sensitivity to different shared hardware resources. Sensitivity scores are used to map threads to cores to minimize contention. Parallelism is not dynamically controlled, and offline characterization is not reliable for data-dependent computations. HOLISYN [6] collects hardware performance counters offline to characterize applications and rank them by their resource utilization. Schedules of applications that run together are created by grouping applications with differing resource utilizations to minimize contention. Resources are allocated to minimize overall energy delay product. ESTIMA [8] uses stall cycles from smaller core counts to predict scalability at larger core counts using an offline approach. In contrast, MAPPER uses an online approach and targets both single- and multi-application scenarios.

Runtime control of parallelism: Prior work has focused on separating hardware context allocation from task scheduling for applications using user-level threads [2, 27]. Callisto [21] is a threading library and resource management layer for parallel runtime systems such as OpenMP. Callisto's goal is to maximize hardware utilization while providing fairness in resource utilization. Each application is assigned its fair share of high-priority worker threads that are pinned to unique hardware contexts without attention to application/thread placement. Additional low-priority workers are pinned to other hardware contexts. Low-priority workers compete to run tasks when the high-priority thread on the corresponding context is stalled, thereby improving utilization. To avoid the need for preemption to ensure fairness, worker threads dynamically pick short tasks at a fine granularity from a task pool. Callisto's granularity requirement potentially adds additional coordination overhead and loss of locality. LIRA [11] augments Callisto by using load instruction rate to colocate the high-priority threads of applications with the largest difference in rates onto the same socket for best performance. The load instruction rate may not reflect locality in the program, nor does it differentiate traffic due to capacity or coherence. By contrast, MAPPER does not dictate the use of fine-granularity tasks, thereby avoiding the extra coordination and potential loss of locality induced by the use of a fine task granularity. MAPPER directly takes parallel efficiency and QoS into account when modifying fair-share allocation of hardware contexts to applications. With the help of multiple performance counter-based metrics, MAPPER performs application/thread placement to reduce resource saturation and harmful interference and minimize communication overheads due to data sharing.

Varuna [43] uses Amdahl's law, along with added parameters for degradation or improvement due to environmental factors, to model performance. Parallel and sequential regions are initially profiled sequentially to provide timing inputs to the model. Linear regression of execution time data from three parallel configurations, with periodic recalibration to detect changes in the execution environment, is used to project performance at a particular core count. Varuna controls the degree of parallelism for an application to minimize time or resource consumption, but does not take the impact of placement on resource contention into account. **Feedback-driven threading (FDT)** [47] uses a training phase created via parallel loop peeling (of the first few iterations of a parallel loop) to measure critical section execution time and parallel loop bandwidth needs (using hardware performance counters). These measurements then feed an analytical model to determine the number of threads that will result in the best performance and lowest power consumption. Depending on the application's bottleneck, either *synchronization-aware threading* or *bandwidth-aware threading* is applied. FDT focuses on parallel applications running in isolation and does not take the impact of thread placement into account.

Similar to MAPPER, NuPoCo [10] targets both degree of parallelism *and* thread-to-core mappings using OpenMP as an example platform, but using different techniques. NuPoCo uses a queuing model to maximize overall system utilization when allocating cores at the granularity of a whole socket to colocated applications. MAPPER allows allocation at the granularity of individual cores and uses both QoS and parallel efficiency instead. NuPoCo uses a hill-climbing algorithm to determine the mapping of applications across sockets when memory bandwidth is the bottleneck. MAPPER uses performance counter information to identify whether sharing, memory bandwidth, or CPU utilization is the application's bottleneck and makes decisions accordingly. As the most closely related work, we compare MAPPER against results presented in Reference [10] in Section 4.5.5.

Cluster management: Bubble-up [32] predicts application co-location for memory-intense latency-sensitive workloads by learning memory pressures and sensitivity of each application in a controlled setting. Q-Clouds [33] targets virtualized environments on cloud platforms, monitoring performance in a continuous feedback loop to determine whether additional resources are required to meet an application's SLA. Off line learning is required to train the interference and resource efficiency model. Quasar [13] and Heracles [30] allocate resources to applications using extensive off line characterization and data from multiple sampling runs to predict resource interference. Arachne [38] and ZygOS [37] introduce task management techniques for achieving microsecond latencies. PerfIso [24] colocates service and batch workloads using a variable number of idle cores to absorb variations in load of the service-oriented application. Mesos [22] offers resources to application runtimes based on a dominant resource fairness metric [19]. The runtimes can respond by accepting all or certain resources that were offered based on the requirements of the application. Omega [41] presents all resources to runtime systems, enabling them to compete for currently available resources or plan to acquire more resources in the future. Argo [16, 36], designed for exascale computing, organizes processes into resource groups called "enclaves." These enclaves can be organized into a hierarchy with subenclaves. The analogue of Argo's enclaves in our work is the `cpuset` cgroups controller, which allows a group to claim a subset of the system's processor and memory nodes. POW [17] complements Argo with a dynamic scheduler that redistributes power budgets between nodes by reclaiming wasted power (difference between power consumed and budget allocated) to increase the power budget for nodes whose consumption was closest to the allocation. These related works do not identify or adjust for resource interference.

Power management: Hoder [42] uses a learning-based approach to simultaneously tune degree of parallelism and **DVFS (dynamic voltage and frequency scaling)** for individual OpenMP applications running in isolation. PUPiL [49] uses a hybrid approach to improve multi-application power/performance, using hardware-based DVFS control to meet a power cap while relying on a software module to explore the configuration space using binary search. The power and application-specific performance feedback from the exploration is then used to determine the best configuration in a manner that is agnostic of the underlying bottlenecks. MAPPER works for both individual and multi-application workloads and directly monitors multiple hardware bottlenecks, using the information to both control the degree of parallelism and to map applications to specific hardware resources in a two-step process.

Cache Allocation: Several methods look to dynamic cache partitioning for performance improvement [5, 39]. These methods focus on a single socket and do not alter an application's degree of parallelism. Utility-based cache partitioning [39] uses cost-effective hardware-based per-core cache utility monitoring (based on MPKI) to feed a **last-level cache (LLC)** partitioning algorithm. Dynamic set sampling is used to reduce hardware overheads and make it cost-effective. CDCS [5] proposes a combined approach to thread and data placement for better performance in a **non-uniform cache architecture (NUCA)** with the goal of minimizing access latency. Cache banks are partitioned and managed as virtual caches, allowing software to allocate capacity and distribute data placement across banks at a fine granularity. Threads are placed on cores to minimize access latency to the virtual caches.

3 MAPPER DESIGN AND IMPLEMENTATION

MAPPER consists of five key components: a parallel application launcher; a performance monitor; a resource allocator; a resource mapper; and an interface to parallel runtimes to allow application

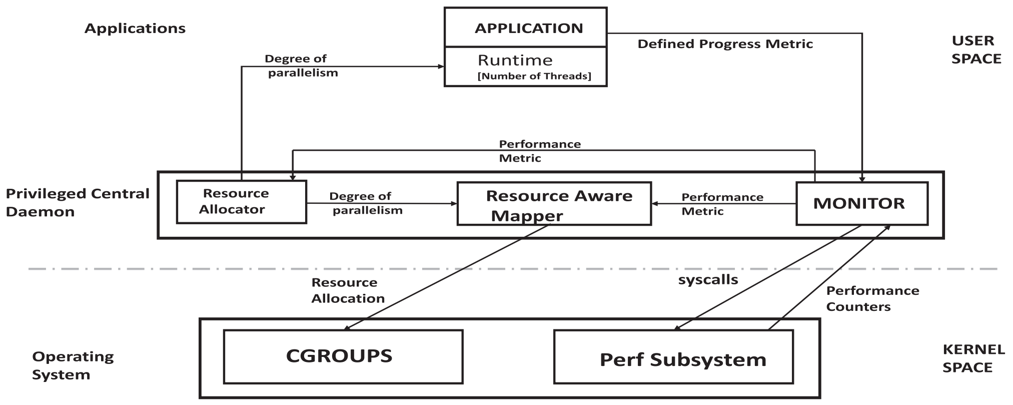

Fig. 1. MAPPER design and information flow.

progress metric definition as well as to provide feedback on allocated hardware contexts. Figure 1 provides an overview of these components and the information flow among them. Application management requires both resource allocation and mapping. The performance monitor, resource allocator, and resource-aware mapper are implemented in a centralized daemon executing in privileged mode. Algorithm 1 outlines the actions performed by the daemon.

ALGORITHM 1: MAPPER Algorithm

```
1: MAPPER Periodicity: default=1 sec
2: minQoS: default = 0.85
3: history: default length = 2
4:
5: Monitor performance:
6: for $appID \in apps$ do
7:    $calculate\_performance\_metrics(perfCounters[appID], prevPerf, currPerf)$
8:    $updateHistory(history, curr\_coreCnt)$
9: end for
10:
11: Allocate resources:
12: for $appID \in apps$ do
13:    $perfChange \leftarrow calcDelta(prevPerf/prev\_coreCnt, currPerf/curr\_coreCnt)$
14:    if $perfChange > perf\_Threshold \vee at\_random$
15:       Explore core count changes informed by performance metrics
16:       Change core count based on FairCoreCnt and minQoS
17:       $sortApplications(NormEff)$
18:       More efficient applications steal $spare\_cores$ from less efficient ones
19:    end if
20: end for
21:
22: Map resources:
23: Sort apps by bottleneck: data sharing > memory bandwidth > compute
24: for $appID \in sorted\ apps$ do
25:    compute $residency\_score$ to preserve locality
26:    map according to bottleneck
27: end for
```

Table 1. Per-Thread Information Obtained from the Performance Monitor

Inter-socket coherence	$R \times RHM/C$
Intra-socket coherence	$R \times (SH + SHM)/C$
Memory bandwidth	$R \times$ LLC-misses$/C$
IPC	instructions$/C$

C = unhalted cycles, R = clock rate, RHM = remote-hit-modified, SH = snoop-hit, SHM = snoop-hit-modified, LLC-misses = last level cache misses [12].

3.1 Application Launcher

To participate in parallel efficiency regulation, applications are launched by the MAPPER application launcher. The launcher runs the application in a child process and registers the child PID with the MAPPER daemon. The daemon establishes a process group for the application under the cgroup `cpuset` subsystem. The process tree in the `/proc` file system under the application's main process PID is traversed to compile the process group for the application. The threads of the application are monitored by the performance monitor. Using `write` calls to the pseudo-filesystem of the `cpuset` subsystem, we can control the CPU mask of all child processes of the application. All child processes belonging to the application are attached to the application-specific subgroup. Dynamically created children are automatically assigned to the same cgroup. The `/proc` file system traversal for registered applications is repeated at each interval to detect and begin monitoring the dynamically created children.

3.2 Performance Monitor

Performance monitoring for an application begins when it is instantiated. MAPPER currently does not leverage information gathered in prior executions and instead adapts to application behavior for the current instantiation and input data. For each registered application instance, the performance monitor gets information about application behavior from low-level hardware performance counters (using the Linux performance event subsystem (*perf* [26])). For each thread of the application, the monitor specifies a set of hardware events that need to be monitored. *perf* periodically reads per-task hardware performance counters and gathers this information over time. This information is used to determine the characteristics of each thread and the application as a whole.

MAPPER uses metrics derived from hardware performance counters commonly available on modern processors to assess application resource bottlenecks [44, 45]. On the Intel Xeon CPU E7-4820, we use five hardware performance counters and six performance events to compute four application per-thread performance metrics (Table 1).

The performance event counts of all threads of an application are aggregated to obtain per-application event counts (resource demand). Whether the application is memory-intensive or data-sharing-intensive (implying a high degree of cross-core data communication) is determined by comparing the application's resource demand (from all of its associated tasks) to system capability (obtained via a one-time offline profiling). Table 2 lists the information currently maintained by the monitor in MAPPER, which includes IPC and the current number of active threads for each application.

In the event that a progress metric is not specified by the application, our runtime system uses IPC values obtained from *perf* (averaged over all active threads in the application) as a measure of progress. IPC as a measure of progress has the inherent flaw of capturing useless work, for example, spinning on a synchronization variable. Techniques to eliminate measurement while spinning (possible if all application synchronization occurs through the runtime) help mitigate these flaws. Our

Table 2. Per-Application Information Supplied from the Monitor and Used by the Resource-aware Mapper

Data sharing bottleneck	Total aggregate coherence activity
Memory bottleneck	Total aggregate memory accesses
Number of threads	Number of active threads
IPC	Average per-thread instructions retired per cycle

experiments with microbenchmarks and OpenMP show that aggregate IPC tracks performance fairly well if the runtime is modified to identify and eliminate spinning from the measurement.

3.3 Resource Allocation

To allocate resources across applications in a fair manner, our implementation places MAPPER's resource allocation and mapping outside the parallel runtime and inside a separate privileged daemon. The MAPPER daemon pools knowledge of individual co-running application's scaling characteristics to better allocate resources to applications that use them more efficiently. This process is carefully moderated to provide well-defined performance guarantees to each application.

The MAPPER daemon is responsible for allocating and mapping resources to registered and simultaneously executing applications. Applications that run in the absence of others will have access to all system resources. Ideally, in these cases, MAPPER will pick the mapping and resource allocation that results in the best performance. For applications that scale well, the best performance is attained when using all available resources. For others, a reduction in resource assignment could result in better performance.

MAPPER relies on information from *perf* to tune and control application parallelism and prevent saturation of resources critical to both application and system-wide performance. For example, applications that saturate memory bandwidth may benefit from a reduction in number of threads to keep queuing delays to a minimum. Applications without any perceived hardware bottlenecks may benefit from an increased number of threads. To avoid repeated bad decisions, the MAPPER daemon maintains a history of application performance at different degrees of parallelism.

In this article, we focus on hardware context allocation while taking resource bottlenecks into account. The MAPPER daemon maintains a per-application count of the number of worker threads. Applications are initially granted a fair share (*FairCoreCnt*) of hardware contexts, depending on the system load. Assuming equal priority, we divide the total available cores uniformly across all running applications; for example, four registered parallel applications with sufficient parallelism to utilize the whole machine will be granted 25% of the hardware contexts.[2]

Using data collected over the application execution, MAPPER calculates the parallel efficiency at different core counts (number of cores used). The performance that applications can fairly expect to achieve (*MaxFairPerf*) is calculated based on the application performance data available for the first *n* core counts, where *n* is the application's fair share of CPU cores.

$$FairCoreCnt = \sum CPU\_Cores / \sum MAPPER\_Applications \tag{1}$$

$$MaxFairPerf = Max(\forall_{k<FairCoreCnt} Perf[k]) \tag{2}$$

$$MinGuaranteedPerf = MinQoS \times MaxFairPerf \tag{3}$$

Each application is guaranteed a minimum performance (Equation 3) defined as being within a specified percentage (*MinQoS*) of *MaxFairPerf* (Equation 2). MAPPER can then determine the number of cores it can spare (to a better scaling application) based on this lower limit on performance.

[2]In our results, we assume that applications have equal priority; other strategies may be possible, depending on application priorities and/or grouping, and could be implemented as an unequal allocation of cores.

We use a value of 85% for *MinQoS* in our experiments (see Section 4 for a sensitivity analysis). To determine the number of spare cores, MAPPER first estimates the difference between the minimum guaranteed performance and the performance at the current core count. This extra performance is calculated as a fraction of the total performance achieved at the current core count. A fraction of cores equal to the fraction of extra to current performance is then estimated as spare cores (Equation 4). This calculation makes the conservative assumption that performance is linear with the number of cores, which is typically not the case, allowing us to ensure that *MinQoS* indeed places a lower bound on performance. It should be possible to adopt other application-specific definitions of progress and their associated notions of minimum QoS. We do not explore them in this article.

$$ExtraPerf = Perf[CurrThr] - MinGuaranteedPerf \tag{4}$$

$$SpareCores = (ExtraPerf * CurrThr)/Perf[CurrThr] \tag{5}$$

In the absence of an application-specific progress metric, **instructions per cycle (IPC)** is used as a measure of progress. We define performance (Perf), a measure of progress, as the sum of the IPCs (or application-specific metric if defined) of each core used in a parallel execution.

An efficiency metric as shown in Equation (6) is calculated for each application. Parallel efficiency (a measure of performance relative to the resources used to achieve it) at n cores is usually defined as $T_1/(T_n \times n)$, where T_1 is the execution time when using one core and T_n is the execution time at n cores: essentially the speedup divided by the core count. The expectation of this efficiency calculation is that single core (sequential) execution efficiency is 1. Since our calculations are dynamic and in terms of progress toward the eventual goal of application completion, we slightly modify this definition. We define parallel efficiency at a particular core count as the execution's progress metric (*Perf*) divided by the core count (shown in Equation (6a)). Since this parallel efficiency is defined in terms of an application-specific progress metric (even baseline IPC varies as a function of the computation), we then use a normalization factor (similar to the expectation that sequential execution has an efficiency of 1). We choose the maximum efficiency across all core counts as a normalizing factor (6b). Using this normalizing factor, a new normalized efficiency is calculated for every core count (6c). A value of 1 at a particular core count indicates that the application is at its highest efficiency (as known to MAPPER at that point in time). If and when MAPPER identifies a more efficient core count, these values will be updated to reflect the change. The normalized efficiency (*NormEff*) allows cross-application comparison when improving overall system efficiency.

The normalized parallel efficiency is generally expected to be close to 1 for scalable applications. It is possible for the normalized efficiency to be significantly less at lower core counts if the application experiences super-linear speedup at higher core counts. Super-linear speedup can occur if the data used for computation by each thread fits within a faster and higher level of the memory hierarchy. This can happen occasionally, since a smaller degree of parallelism can often increase the data handled on a per-thread basis thereby spilling over to slower but bigger caches/memory. It is also possible for parallel efficiency to fall off with increasing number of threads (due to excess synchronization, context switching overheads, higher contention, etc.).

$$\forall_{k \leq CoreCnt} Eff[k] = Perf[k]/k \tag{6a}$$

$$MaxEff = Max(\forall_{k \leq CoreCnt} Eff[k]) \tag{6b}$$

$$\forall_{k \leq CoreCnt} NormEff[k] = Eff[k]/MaxEff \tag{6c}$$

Applications are prioritized according to their parallel efficiency and any free hardware contexts or spare cores from applications that scale more poorly are reallocated to the higher efficiency applications so long as quality of service requirements are met for each application. If an application can benefit from more resources (for example, when it is in an exploratory phase) than it is currently using, then free resources in the system are allocated to it. When free resources are absent, MAPPER attempts to allocate spare cores from an application whose parallel efficiency is less than its own. If any application sees a drop in performance (5%) below *MinQoS* due to granting its resources to other applications, then the reallocation is reverted.

An application running under MAPPER is typically in one of two states: steady or exploratory. Steady state is entered at the end of an exploratory phase, during which a configuration that meets QoS and system efficiency goals is chosen. The runtime system continually monitors performance. Periodically, if performance changes significantly or new hardware bottlenecks are detected (and occasionally even without performance change to capture any gradual changes), then the runtime system transitions the application to exploratory state. The period may be dynamically adjusted based on application behavior [4].

During an exploration phase, MAPPER uses an application's parallel efficiency information (which reflects the existence of bottlenecks) to determine the direction of configuration change. The number of threads in the configuration explored is reduced for applications that saturate memory bandwidth and increased for applications without any perceived hardware bottlenecks. Information on application bottlenecks and behavior may also help reduce the number of unnecessary states explored. In particular, if the application is a source of heavy coherence traffic (inter- or intra-socket), then the degree of parallelism is changed in multiples of cores within a socket to enable localization of coherence traffic within as few sockets as possible. If the new configuration results in improvement (we use a 5% threshold) in performance, then the change is made permanent and exploration continues in the same direction (whether an increase or decrease in active parallelism). If the new configuration results in reduced performance, then the previous configuration is restored and the application is transitioned to stable state.

3.4 Resource Mapping

In addition to decisions on the number of hardware contexts to allocate to each application, MAPPER also carefully maps application threads to specific hardware contexts based on CPU, memory, cache, and interconnect resource needs using a refinement of the heuristics used by the SAM system [44, 45], which includes grouping threads by application. Input to this decision-making strategy is a priority-based sorted list of applications with each application listed using its critical bottleneck. Most modern multiprocessor systems have a number of hardware resources, all of which may or may not be utilized by every application. To arrive at the application's critical bottleneck, we develop microbenchmarks to perform a one-time characterization of the system's capability by stressing hardware resources, one at a time. During this characterization phase, we monitor each microbenchmark using hardware performance counters. Data from the counters help us arrive at thresholds that can be used to detect bottlenecks in application performance.

The applications are sorted according to the priority of the hardware bottleneck exhibited: (Data sharing (coherence traffic) bottleneck > Memory bandwidth bottleneck > Compute (IPC)). Within each category, applications are sorted based on how severely they contribute to the overall bottleneck. Applications with higher contributions are given higher priority for resource mapping. The resource mapper walks this sorted list and determines the mapping strategy using the application's bottleneck information maintained as already shown in Table 2. Mapping strategies decide whether to colocate threads on the same core (if sharing- but not CPU-intensive), on the same socket (if sharing- and CPU-intensive), whether to spread them across sockets (if memory-intensive), or avoid hyperthreads (if CPU-intensive) based on core availability and the application's critical bottleneck.

The above mapping strategy can assign one of many possible topological configurations of hardware contexts to an application. To preserve locality and minimize migrations, MAPPER attempts to minimize differences between successive cpuset mappings. When the system has multiple sockets, this problem is magnified, and so MAPPER generates a sorted list of sockets for each application, reflecting a preference for application thread placement based on the chosen mapping strategy.

For strategies that require colocation, sockets are sorted in descending order of the application's *residency score*, which is the total number of hardware contexts already allocated to that application within each socket. The residency score is used to minimize the data movement and cache warmups induced due to thread migration. When new hardware contexts are allocated to the application, MAPPER will attempt to accommodate them in sockets with the highest scores. If the prior mapping decision spread the application threads across sockets either due to unavailability of hardware contexts (for colocation) or due to change in type of hardware bottleneck, then MAPPER will prefer to colocate them to the highest scored sockets in the current decision-making interval. Naturally, when applications need to relinquish resources they are released from the lowest non-zero scored sockets.

When the mapping strategy necessitates distributing hardware contexts across sockets, the sockets are sorted in ascending order of their residency scores. If additional hardware contexts are allocated to the application, then they are placed on the sockets with lowest scores (wherever possible). Conversely, if hardware contexts are taken away from an application, then those on the highest-scored sockets are freed.

Within each selected socket, contexts that are already occupied by the application are retained when possible to further minimize migrations. Compute-intensive threads are preferably placed on physical cores that are occupied by hardware threads that are not compute bound.

When hardware contexts that satisfy the mapping strategy are not available on any socket, MAPPER attempts to pick hardware contexts that will result in the least amount of resource contention (whether CPU, cache, or interconnect). MAPPER's resource allocation and mapping decisions constitute a complementary two-step process that when combined eliminates the need to explore the entire state space of possible mappings of ${}^{n}P_{c}$, where n is the number of threads and c is the number of contexts.

3.5 MAPPER Runtime Interface: Parallelism Regulation

As described in Section 3.1, applications participating in parallel efficiency regulation are launched by the MAPPER application launcher. If the application runtime so chooses, then it may use the launcher both to communicate additional application-specific progress metrics and to determine current application-specific MAPPER resource allocation. Figure 1 shows the information flow between an application runtime and MAPPER, which takes place via a shared memory segment established between the daemon and the launcher. Runtime systems may choose to use the returned information on resource allocation to change task creation and granularity.

Example Use in OpenMP: Upon application initialization, the number of active threads in OpenMP is set as specified by the user or to any default value set in the system. If no such value is specified, then the runtime determines the number of hardware contexts available in the machine and sets the number of threads to this value. In OpenMP, active application parallelism can be controlled in two different ways. The most direct method to control the active numbers of threads is to change the size of the thread pool that performs the parallel computation. We modify OpenMP to control the thread pools used to execute a parallel region at the beginning of a parallel region (defined by #OMP parallel).

However, once a parallel region is currently executing, threads may no longer be added or removed without changes to the application's parallel structure. We can, however, change the number of active threads for each loop or other OMP task construct within the parallel region. OMP loops may be either statically or dynamically scheduled. To control the parallelism for static loops, we modify the GCC compilation code to introduce a *parallelism_control* function call at the beginning of these constructs. The *parallelism_control* function is called independently by every thread within the parallel region and is used to select the threads from the current worker pool that will participate in the current OMP computation. The remaining threads are queued at a barrier and may be used for the next OMP construct, depending on changes in resource allocation. OMP dynamic loops are implemented by having worker threads request tasks to execute upon completion of the previous task; if no additional tasks are found, the thread queues at a barrier. We modified the code within the OpenMP library so excess worker threads (determined by calling *parallelism_control*) go directly to the barrier without looking for new tasks. Thus, the finest granularity of control within OMP is at the level of these parallel constructs. Our modifications do not restructure the computation, its granularity, or its load balance and are compliant with OpenMP specifications.

We modified the OpenMP parallel constructs to interface with the launcher thread at these safe points to effect changes in the degree of parallelism and thereby reduce potential load imbalance and synchronization overheads. Additionally, when runtimes can change parallelism in response to MAPPER, synchronization and other runtime-related overheads can be reduced, making performance counter information more reflective of the actual computation performed by the worker threads. The alternative would be to modify the runtime to disable performance counters while executing synchronization and other thread creation/deletion code. This would be expensive and can misdirect MAPPER about real CPU contention or data sharing introduced due by spinning synchronization or other runtime code.

We also added support for application-specific progress reporting by using a shared array with dedicated entries per application thread, padded to prevent false sharing. Application developers can use a newly introduced function call within OMP parallel constructs to manipulate this dedicated entry to imply application progress. This information is then aggregated by the modified runtime and passed to MAPPER. We do not use this feature in our evaluation.

4 EVALUATION

4.1 Experimental Setup

Our experiments were conducted on a quad-socket machine with each socket equipped with an Intel Xeon CPU E7-4820 v3 running at 1.90 GHz. The Xeon CPU E7-4820 is from Intel's Haswell processor family and contains 10 physical cores with 2 hardware contexts (logical threads) in each physical core, as well as an on-chip 25 MB of shared last level cache (L3). Each socket is also equipped with 64 GB (four 16 GB DIMMs) of local DRAM memory for a total of 256 GB memory in a four-socket NUMA configuration. We use Linux kernel 4.17.11 and its perf subsystem to access the hardware performance counters. Modifications to OpenMP were performed on top of gcc-8.1.0.

4.2 Workloads and Their Characteristics

To evaluate our MAPPER system, we use different benchmark applications gathered from PARSEC-v3 [7], OMPSCR-v2 [15], and NAS Parallel Benchmarks [3]. We also use increasingly widely used graph-based applications that perform machine learning, data mining, pattern recognition, and computer vision [23, 28, 29, 31, 40, 50]. PARSEC contains applications that perform routing for chip designs, calculate content similarities, video encoding, image processing, and itemset mining.

Table 3. Characteristics of Non-OpenMP Applications

Application	Parallelization Model	Bottlenecks	Standalone Performance
SVD [G]	Data-parallel	High Data Sharing and CPU-intensive	78.53 secs
SGD [G]	Data-parallel	High Data Sharing	61.57 secs
BSGD [G]	Data-parallel	High Data Sharing	64.84 secs
LSGD [G]	Data-parallel	High Data Sharing	66.88 secs
PMF [G]	Data-parallel	Medium Data Sharing and CPU-intensive	130.865 secs
ALS [G]	Data-parallel	Medium Data Sharing and CPU-intensive	187.42 secs
RBM [G]	Data-parallel	High Data Sharing	69.68 secs
Blackscholes [P]	Data-parallel	Low Data Sharing and CPU-intensive	30.4 secs
Canneal [P]	Unstructured	High Data Sharing	53.67 secs
X264 [P]	Pipelined	Low Data Sharing and CPU-intensive	28.22 secs
Ferret [P]	Pipelined	Medium Data Sharing and CPU-intensive	12.18 secs
Freqmine [P]	Data-parallel	High Data Sharing and CPU-intensive	33.476 secs
Swaptions [P]	Data-parallel	Low Data Sharing	16.915 secs
Fluidanimate [P]	Data-parallel	Low Data Sharing and Memory-intensive	24.923 secs
Memcached [C]	Data-parallel	Low Data Sharing and Memory-intensive	p99 lat: 11.75 ms
In-memory-analytics [C]	Data-parallel	Memory-intensive	Throughput: 0.19
Mediastreaming [C]	Pipelined	CPU and Memory-intensive	Net I/O per sec: 299408.9

For [G]raphChi/[P]ARSEC applications performance is measured in terms of execution time (using all available hardware contexts) denoted in seconds. For [C]loudsuite benchmarks each benchmarks has different performance units as denoted.

OMPSCR contains applications implemented using OpenMP that perform kernel computations important to scientific computing including jacobi, gaussian elimination, mandel-brot simulations, and molecular dynamics. NAS Parallel benchmarks include OpenMP applications that compute conjugate gradient, multi-grid mesh solver, 3D Fast Fourier Transformations, and Scalar Penta-diagonal solver. We also evaluate our methodology on the Cloudsuite [18, 35] benchmarks, which range from popular latency-critical workloads like Memcached to memory/network throughput oriented In-memory-analytics and Mediastreaming workloads, respectively. These service-oriented workloads are of interest to cloud service providers to gauge their system performance.

We highlight characteristics of these applications and their standalone performance (using all available hardware contexts) in Table 3. The OpenMP applications are data-parallel with very little data being shared among the threads of the application. With the exception of molecular dynamics, these applications are very scalable and rely on both CPU and memory bandwidth.

4.3 Scheduling Policies

To identify the benefits and tradeoffs in the MAPPER resource allocation and mapping, we compare against a fair share and a hill-climbing approach to resource allocation, as well as against the default Linux scheduler (Linux's Completely Fair Scheduler). In the case of the hill-climbing and MAPPER schedulers, the interval for periodic exploration is set to 1 second and the exploration is performed in increments of four hardware contexts. All schedulers use `cpuset` to control thread-to-core mappings. MAPPER uses resource- and contention-aware policies for mapping, while the fair share and hill-climbing schedulers use the Linux default mapping of application to hardware contexts (which distributes application threads across sockets, avoiding hyperthreads if free hardware contexts are available).

MAPPER scheduler. Our MAPPER policy allocates each new application its fair share of resources. In every iteration, MAPPER generates a resource estimate (number of hardware contexts) for each application as a function of system load (number of other applications and their needs) and application need. The per-application resource estimate is aided by information on hardware bottlenecks, which also determines how the applications are mapped onto hardware contexts in the machine (Section 3.4). Following the per-application resource estimation, MAPPER determines a per-application final resource allocation based on the parallel efficiencies and QoS measurements of all other co-running applications.

Fair share scheduler. Fair share, as the name suggests, allocates resources equally to all concurrently executing applications. For Modified OpenMP applications, Fair-share provides feedback to the OpenMP runtime so the runtime may choose to match active threads to allocated resources.

Hill-climbing scheduler. Our hill-climbing scheduler starts with a fair-share allocation for each application and iteratively and incrementally explores the impact on IPC or the application-specific progress metric of increasing/decreasing hardware context allocation to an application. Application performance bottlenecks are not examined. Hill-climbing searches for a maxima in the progress metric, with periodic random perturbation to adapt to application and environment changes. Similar to the Fair-share scheduler, for Modified OpenMP applications, hill-climbing provides feedback to the runtime to modify its allocation. Unlike MAPPER, the hill-climbing scheduler is not bottleneck-aware.

Linux completely fair scheduler. The Linux load balancer balances load based on the total number of active threads in the system. This creates an incentive for applications to create more threads than there are hardware contexts. Applications that are composed of more threads will be able to acquire a larger share of CPU time slices.

MAPPER, Fair share, and Hill-climbing adopt the approach of isolating applications on exclusive hardware contexts but retain the load balancing within each application's set of hardware contexts. Comparing MAPPER's performance with Fair share will demonstrate the benefits of resource-aware application management. Comparing MAPPER's performance with Hill-climbing allows us to evaluate the benefits of identifying resource bottlenecks in comparison to using a single IPC metric. Comparing all three schedulers with the default Linux approach highlights the tradeoffs between resource partitioning and sharing.

4.4 Standalone Parallel Applications on MAPPER

When applications run standalone, they have access to all resources in the machine. Using all available resources would be the default mode (standalone performance depicted in Table 3) unless the application exhibits better performance if fewer resources are used. MAPPER attempts to find this performance peak for those applications that do not scale sufficiently to be able to utilize all resources.

Among applications in our workload, MAPPER finds the peak performance for the graph-based applications **Single Value Decomposition (SVD), Stochastic Gradient Descent (SGD), Biased SGD**, and **Alternating Least Squares (ALS)** to be at 20 hardware contexts. The performance improvements obtained are: 4.8%, 4.1%, 8.2%, and 4.8%, respectively. Due to the applications exhibiting significant data sharing, MAPPER also attempts to colocate the applications within as few sockets (one socket on our four-socket 80-core experimental platform) as possible to localize communication. Other applications are run with the full resources in the system.

4.5 Multiprogrammed Workloads

With multiple simultaneously executing applications, the scalability of individual applications is load-dependent in addition to being data- and hardware-dependent. We evaluate the benefits of using MAPPER when applications share resources with each other on the same machine. We split the evaluation into three parts. First, we study the performance of applications whose runtimes are not modified to interact with MAPPER. Next, we investigate the effectiveness of MAPPER when

Table 4. Reallocation Statistics for the Multiple Application Mixes (Figure 2 (Two Application Mixes) & Figure 3 (Three/four Application Mixes))

Application Mixes	MAPPER		HILL		FAIR	
	N	C	N	C	N	C
80 canneal, 80 SGD	0.76	8.06	0.67	7.97	0.061	1.99
80 canneal, 80 BSGD	0.74	7.8175	0.60	7.04	0.10	2.55
80 canneal, 80 LSGD	0.77	7.98	0.59	6.25	0.081	2.56
80 canneal, 80 SVDPP	0.82	9.41	0.63	8.21	0.091	2.8
80 canneal, 82 x264, 80 SVDPP	1.30	10.34	1.45	13.18	0.378	4.2
80 SGD, 80 canneal, 80 LSGD	1.165	8.74	1.41	13.99	0.58	7.65
82 x264, 80 SVDPP, 80 LSGD	1.43	10.96	1.54	19.43	0.71	9.27
80 freqmine, 82 x264, 80 SGD	1.54	13.12	1.33	15.11	0.43	5.76
80 canneal, 80 blackscholes, 80 SGD, 80 SVDPP	1.83	13.13	1.84	16.17	0.21	2.89
80 canneal, 80 SGD, 80 SVDPP, 80 BSGD	1.61	10.71	1.97	19.95	0.21	3.28
80 SGD, 80 SVDPP, 80 BSGD, 80 LSGD	1.60	10.88	1.55	19.87	0.373	6.15
80 x264, 322 ferret, 80 canneal, 80 SVDPP	1.70	10.72	1.70	15.81	0.385	4.62
80 freqmine, 80 ferret, 80 SVDPP, 80 LSGD	1.13	8.16	1.84	18.11	0.385	4.62
80 freqmine, 80 x264, 80 SGD, 80 BSGD	2.14	17.50	1.84	19.49	0.42	6.79
80 SGD, 80 SVDPP, 80 ALS, 80 RBM	1.78	14.73	1.85	21.08	0.54	9.58

N Represents the Number of Times `cpuset` is Called Per Second, C Represents the Number of Context Migrations per Second.

interfaced/integrated with the application runtimes using OpenMP as an example runtime. Finally, we perform an evaluation that combines applications from modified and unmodified runtime systems.

When running multiple applications simultaneously, we run the longest (runtime-wise) application three times. To ensure concurrent execution of the workloads across the entire execution of each constituent application, the shorter applications run continuously in a loop until all iterations of the longest application are completed. The experiment is repeated five times and the arithmetic mean of runtimes is used to calculate speedups. Geometric mean speedups of individual applications are calculated, followed by the geometric mean for the workload mix. **In all the graphs, the whiskers (error bars) indicate the range (minimum and maximum) of speedups achieved by the individual applications in the workload mixes and are NOT a representation of the variance of the mean speedup.**

4.5.1 Parallel Application Evaluation without Runtime Integration. Figures 2 and 3 present the geometric mean of speedups (relative to standalone execution time of each application using the full resources of the machine on Linux) with two-application and 3-4-application workload mixes without runtime integration with MAPPER. The mixes consist of PARSEC and GraphChi [29] benchmarks and except where noted are run with the maximum number of contexts supported by the system. In general, the expectation is that speedups will be less than 1 due to resources being shared (in time and/or space) across multiple applications. Speedups greater than 1 indicate either that the application has higher performance at a lower core/thread count or that MAPPER is able to identify and alleviate bottlenecks as well as reduce communication costs by exploiting locality. *The workloads and mixes were chosen to maximize the range of characteristics explored (parallelization strategies and data and resource sharing behavior) rather than present every data point.*

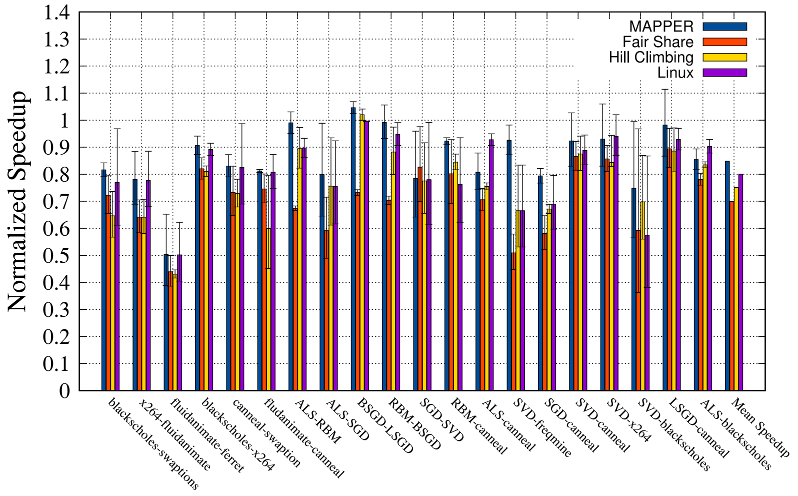

Fig. 2. Geometric mean of speedups (normalized to standalone execution time on Linux; higher is better) achieved by applications in 2-application multiprogrammed workloads using the "MAPPER," "Fair share," "Hill climbing," and "Linux completely fair" schedulers.

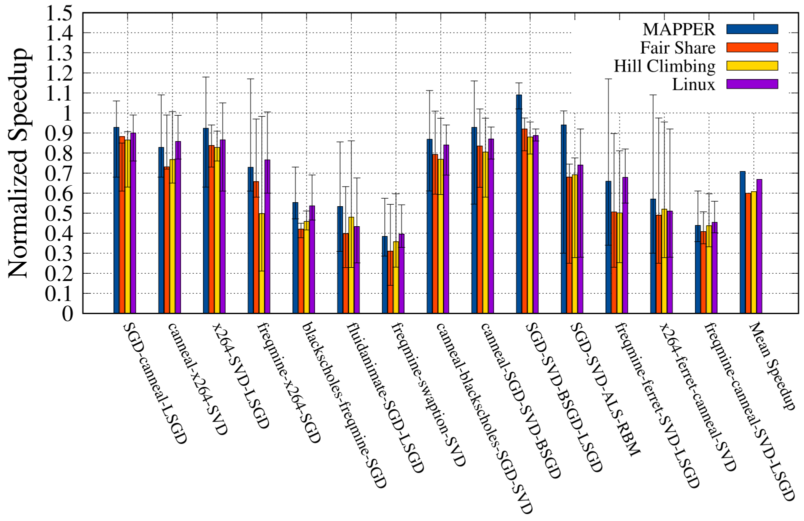

Fig. 3. Geometric mean of speedups (normalized to standalone execution time on Linux) achieved by applications in a 3-4-application multiprogrammed workload using the "MAPPER," "Fair share," "Hill climbing," and "Linux completely fair" schedulers.

Table 4 shows the Number of times *cpuset* is called per second and the number of context migrations per second for application mixes in Figures 2 and 3. These migrations contribute to overheads but also correlate to the performance impact. The specific workload mix, number of hardware context changes due to bottleneck changes, and phased characteristics, combine to affect the overall performance.

Overall, MAPPER is able to modestly outperform Linux (by 6.63%) while Fair share and Hill Climbing perform worse than Linux. The geometric mean of all application mix speedups in Figure 2 is 0.848 for MAPPER, 0.800 for Linux, 0.750 for Hill Climbing, and 0.699 for Fair share. In

Figure 3 the speedups are 0.708 for MAPPER, 0.667 for Linux, 0.607 for Hill Climbing, and 0.599 for Fair share.

GraphChi applications have significant data sharing and therefore incur high inter-socket coherence activity. Consequently, they scale poorly with the addition of threads due to increasing pressure on the inter-socket interconnect and benefit from co-location onto as few sockets as possible. Applications like ALS from GraphChi also have additional memory/memory bandwidth requirements. MAPPER is aware of the poor scaling and inter-thread communication and guides its exploration to reduce the hardware contexts allocated to the application while simultaneously packing them into as few sockets as possible, leading to improved performance and efficiency. The resource allocation and mapping decisions together lead to significantly improved performance.

The best performance with MAPPER is obtained when data sharing-intensive applications are run together. In this scenario, MAPPER is able to localize data sharing and isolate each applications' communication within one socket, thereby improving overall performance. Speedups greater than 1 can be expected due to eliminating the cost of inter-socket communication. The maximum performance gain of MAPPER over Linux is 39% for application mix SVD-freqmine. Within this workload mix, SVD shows the highest individual performance gain over Linux of 60%. The highest individual speedup attained by any application is 12% greater in MAPPER compared to Linux.

x264 and ferret are pipelined applications from the PARSEC [7] benchmark suite. They create a much larger number of threads than hardware contexts for some pipeline phases. As a result, they benefit from the Linux scheduler's load balancer, which distributes the threads across all cores, resulting in x264 and ferret getting a larger share of the compute power, especially when the co-running application scales poorly. The strategy of exclusive division of CPU resources among applications (hardware context isolation) employed by MAPPER, Fair share, and Hill Climbing prohibits computational resource sharing across applications, significantly hurting performance for Fair share and Hill Climbing. MAPPER is able to mitigate the impact of hardware context isolation by detecting the co-running GraphChi application's poor scaling and allocating more CPUs to the PARSEC applications, while also reducing communication latencies for GraphChi workloads by co-locating them on as few sockets as possible. Additionally, when both ferret and x264 are in the workload mix, Linux's load balancer hurts performance when compared to MAPPER.

The worst performance loss observed with MAPPER relative to Linux is 11% for ALS-canneal. Within this workload mix, MAPPER loses 18% performance for ALS over Linux. ALS has significant memory requirements, resulting in the need for high memory bandwidth, but also exhibits phased execution. The tradeoffs of colocation of all threads within a socket for better locality versus distribution for better bandwidth affect its performance.

From a QoS perspective, a QoS target of performance with a Fair-share allocation of resources (primarily compute) is a common default expectation. We use 85% of Fair-share performance as the target QoS for application performance in MAPPER, providing some flexibility in resource allocation to meet the QoS requirements.

Using a QoS guarantee of 85% of the IPC achieved with a fair-share scheduler, MAPPER meets the QoS guarantee for all mixes except one, where canneal achieves 81% of the fair-share scheduler's IPC when co-run with freqmine, SVD, and LSGD. In contrast, Linux violates QoS guarantees in four cases, with fluidanimate being the worst performer when co-running with SGD and LSGD, achieving 73% of fair-share performance. It is important to note that the geometric mean of minimum speedups for Figures 2 and 3 combined are 0.630, 0.529, 0.546, and 0.612 for MAPPER, Fair-share, Hill Climbing, and Linux, respectively. In other words, MAPPER improves the minimum speedup provided to any application in a workload mix by 19% on average when compared to Fair-share. Specifically, MAPPER outperforms Fair-share both via better mapping decisions and by using fewer resources (better resource allocation) when applications do not scale.

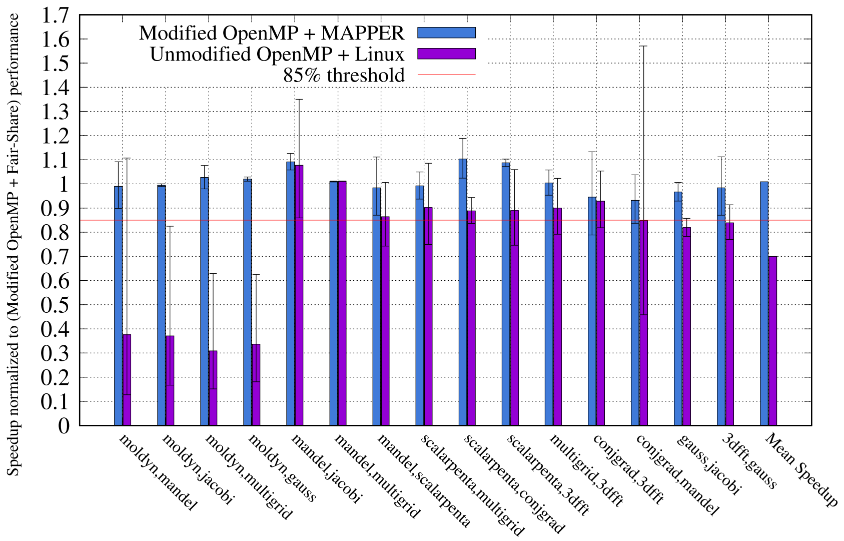

Fig. 4. Geometric mean of speedups (normalized to (Modified OpenMP + Fair-share) execution time) achieved by applications in a multiprogrammed OpenMP workload using modified OpenMP + MAPPER and vanilla OpenMP running directly on Linux (higher is better).

For the remainder of the evaluation, we use Fair share scheduling as the baseline for comparison of both performance and QoS.

4.5.2 Evaluation of Integrated OpenMP Runtime. Figure 4 shows the impact of modifying the OpenMP runtime to integrate with the MAPPER scheduler on the performance of multiple application workloads. Specifically, OpenMP utilizes information on the number of allocated hardware contexts to control the application's task granularity and synchronization. In Figure 4, the speedup of each multi-application workload mix is calculated by taking the geometric mean of each constituent application's execution time normalized to when run using Modified OpenMP + Fair-share. **Note that our normalization baseline also takes advantage of runtime integration.** We use MAPPER's default IPC information collected from hardware performance counters to reason about application progress.

Modified OpenMP with the MAPPER scheduler has a geometric mean speedup relative to Linux of 1.442×. Workload mix moldyn,multigrid achieves the highest gain of 3.33× compared to Linux. These speedup numbers show the importance of runtime integration in managing co-running parallel applications. The performance improvements with our OpenMP applications arise from MAPPER's ability to provide feedback to OpenMP to adjust application parallelism, which in turn reduces oversubscription, synchronization, and other wasteful overheads. There is significant performance benefit that can be realized even for highly scalable applications.

Moldyn benefits significantly from MAPPER's elimination of oversubscription (by oversubscription, we mean applications that use higher thread counts than assigned hardware contexts) when running with other applications (a normalized speedup relative to Fair-Share performance of up to 1.092 on MAPPER when running with mandel, relative to 0.127 on Linux, an order of magnitude improvement). Significant improvements in performance are also observed for gauss, scalarpenta, and multigrid.

MAPPER's QoS-based resource allocation along with hardware context isolation is able to mostly eliminate, or at least mitigate, QoS violations, while improving or maintaining Fair-share

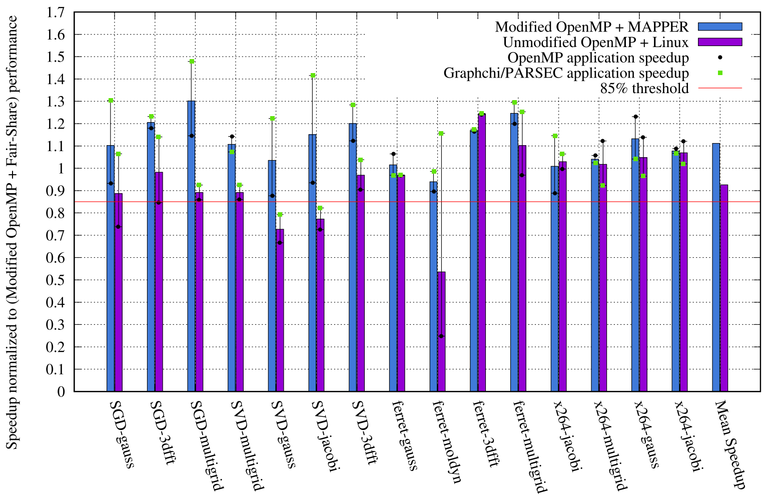

Fig. 5. Geometric mean of speedups (normalized to (Modified OpenMP + Fair-Share) execution time) achieved by applications in multiprogrammed non-OpenMP and OpenMP workload mixes using modified OpenMP + MAPPER and unmodified OpenMP running directly on Linux (higher is better).

performance. Specifically, applications such as conjgrad that exhibit poor scalability allow scalable applications such as mandel and 3D FFT to achieve higher performance than Fair-share. However, this comes at the cost of QoS violations. MAPPER isolates the hardware contexts allocated to each application, but enables mandel and 3D FFT to borrow hardware contexts from conjgrad (up to 4) due to conjgrad's poor scalability. This tradeoff leads to an overall improvement in performance with greatly improved QoS relative to Linux. QoS guarantees (of being within 85% performance of Fair-share) are slightly compromised, however, due to our current resource allocation granularity of four hardware contexts: conjgrad achieves a speedup of 0.836 and 0.788 (relative to Fair-share scheduling) when run with mandel and 3D FFT, respectively, compared to 0.458 and 0.818 for Linux. All other applications meet the minimum QoS guarantee when running with MAPPER.

MAPPER's worst speedup is 0.93 for the workload mix conjgrad,mandel. In contrast, Linux's worst speedup is 0.3 for the workload mix moldyn,multigrid. Benchmark conjgrad achieves the worst individual speedup of 0.79 on MAPPER when running with 3dfft. For Linux, it is moldyn that achieves a speedup of 0.12 when co-run with mandel. MAPPER's worst-case performance is 6% below the QoS target, while the worst case on Linux is 73% below the QoS target. MAPPER violates QoS in only two instances in our test cases compared to Linux's 13 violation instances.

4.5.3 Combined Workload Evaluation. Figure 5 shows the impact of MAPPER for combinations of modified/integrated OpenMP applications and applications using other runtimes. MAPPER achieves a geometric mean speedup of 1.20× compared to Linux. The geometric mean speedup relative to Modified OpenMP + Fair-share is 1.11 for MAPPER and 0.925 for Linux. The geometric mean of the minimum speedups for MAPPER and Linux are 1.028 and 0.822, respectively, and the mean for maximum speedups is 1.201 and 1.043, respectively. Ferret-moldyn achieves the best performance gain of 1.75× compared to Linux.

Data sharing-intensive applications such as SGD and SVD benefit significantly from resource-aware mapping performed by MAPPER. SGD and SVD workload mixes improve by 31% and 34%, respectively, over Linux. The resource-aware mapping enables efficient task placement and resource sharing with the other co-running applications. MAPPER is able to reduce the number of hardware contexts and constrain the (poorly scaling) data-sharing applications to one socket (to localize communication), thereby freeing up resources for the co-running workloads and simultaneously improving the performance of both applications. Jacobi, 3D FFT, and gauss benefit significantly from the availability of more hardware contexts. x264 also benefits from co-location of its threads during phases of computation when the data sharing is high. Unlike the GraphChi applications, x264 is not restricted to a single socket, since the application scales well with increasing number of threads.

MAPPER improves the performance of workload mixes with ferret by 21% over Linux. The speedup of ferret alone (individually within the workload mix) relative to Linux is 0.86, however. This slowdown relative to Linux stems from the fact that ferret running on Linux competes for resources using its 322 threads (compared to other applications' 80 threads). This oversubscription leads to a larger share of CPU time, resulting in significant performance inequities (e.g., ferret-moldyn).

The worst speedup achieved is 0.93 and 0.53 for ferret-moldyn when run using MAPPER and Linux, respectively. The worst individual speedup achieved is 0.88 for gauss in SVD-gauss for MAPPER and 0.24 for moldyn when running with ferret for Linux. MAPPER meets the QoS requirements for all applications. In contrast, Linux violates QoS in four instances, in some cases (as for moldyn above) showing a dramatic reduction in performance relative to Fair-share.

4.5.4 Service-oriented Workloads. Three benchmarks from Cloudsuite [18, 35] are used to evaluate the performance of service-oriented applications on multisocket systems using MAPPER: latency-sensitive Memcached using a 300MB Twitter dataset and a single server with 80 threads; throughput-oriented In-memory-analytics, a recommender system using a Netflix database to rank the top 50 movies for the requester; and network-intensive Mediastreaming, where an Nginx web server is used as a streaming server to host videos of various lengths and qualities.

Note that applications in these workload mixes may demonstrate speedups of more than 1 or close to 1 even in multi-workload scenarios mainly due to non-overlapping resource needs. For example, for Mediastreaming, the resource bottleneck is network bandwidth and I/O compared to data locality and memory for SGD; thus both applications can perform close to or better than standalone performance.

Figure 6 shows the performance of the Cloudsuite benchmarks when multiprogrammed with Parsec and Graphchi benchmarks. Our performance metric is the normalized (to Fair-share behavior) p99 latency for Memcached, throughput (movies ranked/sec) for In-memory-analytics, and network bandwidth (I/O/sec) for Mediastreaming. The geometric mean speedup for MAPPER is 1.0538 compared to Linux's 0.944 normalized to Fair-share performance. Focusing on the Cloudsuite benchmarks, MAPPER achieves a speedup of 0.856 compared to 0.725 with Linux across all mixes. MAPPER meets the QoS requirements of the Cloudsuite benchmarks in all but 1 of 19 cases, specifically, for Memcached (achieving 71% of fair-share performance) when running with BSGD. Linux violates QoS requirements of the Cloudsuite benchmarks in nine cases, with the lowest performance being for Memcached (achieving 58% when co-running with BSGD).

GraphChi/PARSEC workloads are representative of best-effort workloads when co-run with the Cloudsuite benchmarks. The GraphChi/PARSEC workloads attain a speedup of 0.906 with MAPPER versus 0.861 with Linux. MAPPER is able to achieve a 1.18× performance improvement over Linux for Cloudsuite benchmarks in the application mixes and 1.05× for best-effort workloads (PARSEC/GraphChi).

Memcached: MAPPER achieves an average p99 latency that is 24.6% lower than that with Linux for Memcached. For example, when running with BSGD, the recorded p99 latency while running with MAPPER is 17.12 ms compared to 20.69 ms when running with Linux. The higher the

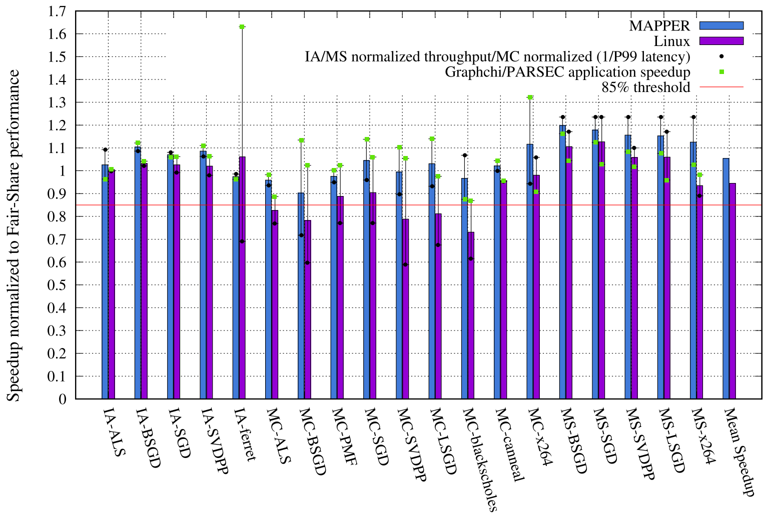

Fig. 6. Performance (compared to Fair-shared performance) for Cloudsuite applications using MAPPER and Linux (IA=In-memory-Analytics, MS=Mediastreaming, and MC=Memcached).

measure of normalized performance, the lower the latency. MAPPER identifies resource bottlenecks, needs, and sharing behavior, which benefits Memcached by better thread placement for localized memory access. Application mixes with Memcached running on MAPPER achieve a geometric mean speedup of 1.178× compared to Linux.

In-memory-analytics: Normalized performance for workload mixes containing In-memory-analytics is 1.02 for Linux and 1.05 for MAPPER. Normalized performance of In-memory-analytics is 1.06 when running with MAPPER vs. 0.926 when running with Linux. In-memory-analytics has 14.44% higher throughput when running with MAPPER compared to when running with Linux. Workloads co-run with In-memory-analytics gain only 2.23% over Linux.

However, when co-running with ferret on Linux, In-memory-analytics loses significant performance (resulting in a large violation of QoS) as a result of Linux's load-balancer providing more compute contexts to ferret due to its large thread count. MAPPER is able to meet the QoS requirements for both applications, avoiding Linux's large performance variance and significant QoS violation. The MAPPER algorithm identifies the memory locality and bandwidth needs of In-memory-analytics. Through its iterative process, it also ensures QoS for In-memory-analytics and does not allow ferret to steal additional compute resources.

Mediastreaming: Mediastreaming and mixes with Mediastreaming performs 11% better on average with MAPPER when compared with the baseline Linux system. When run with x264 on MAPPER, the normalized performance of Mediastreaming is 1.23, compared to 0.89 on Linux. Mediastreaming sees a significant improvement in performance, since MAPPER can identify memory bottlenecks and distribute threads across sockets to better utilize available resources.

Service-oriented workloads are critical workloads that are not tolerant to performance degradation, as it directly impacts business needs. We observe that the geometric mean speedup for service-oriented workloads in the workload mixes is 1.036 for MAPPER compared to 0.873 for Linux when normalized to Fair-share performance. This shows that for workload mixes like the ones depicted in Figure 6, MAPPER is able to allocate resources and map the component applications to simultaneously ensure QoS and improve overall system efficiency.

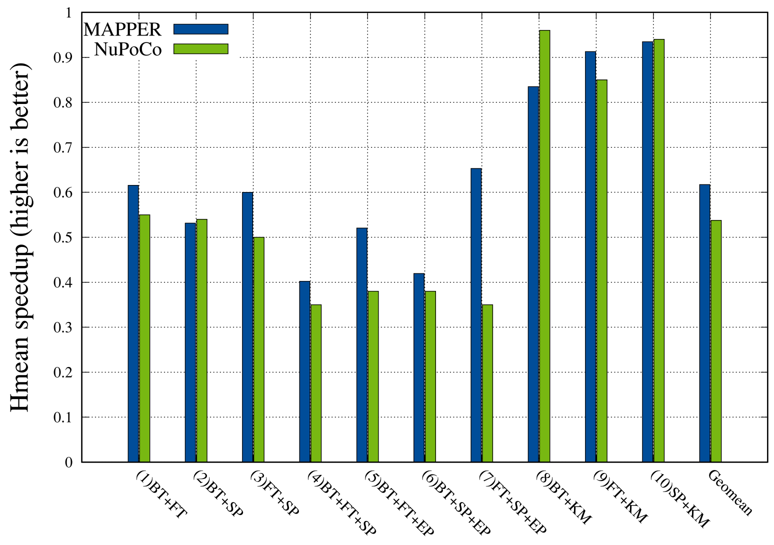

Fig. 7. Hmean of speedup relative to standalone execution for the workload-mix depicted in Figure 12 of Reference [10] for MAPPER and NuPoCo. NuPoCo values have been taken directly from Figure 12 of Reference [10].

4.5.5 Comparison with NuPoCo [10]. To avoid any configuration issues with a port of NuPoCo to our platform resulting in its performance degradation, we compare NuPoCo to MAPPER by replicating the experiments in Figure 12 from Reference [10] using MAPPER. As a result, the evaluation methodology in this section differs from that in the rest of the article. In particular, following NuPoCo methodology, each application in a workload mix is run once with all the applications starting at the same time. As a result, the number of co-running applications decreases as applications complete their execution. Similarly, we use a harmonic mean to match NuPoCo's methodology. We present our results against those presented in Figure 12 of Reference [10] in Figure 7. While a direct comparison is not possible due to differences in platform architecture and environment, a qualitative comparison may be made.

Except for *KM*, all the workloads in the mixes are from the NAS parallel benchmark suite [3]. *BT*, *FT*, and *SP* are CPU and memory-intensive but with differing execution times (*FT* runs for the shortest duration out of these three). *EP* is *embarrassingly parallel* and does not access memory but has a very short execution span. *KM* is *kmeans* from the Rodinia benchmark suite [9] and is run with an input set of 3,000,000 objects and with a cluster parameter of 5.

In Figure 7, we see that MAPPER's speedup is higher than NuPoCo's on average across these workload mixes and is consistently higher for the 3-workload mixes. The latter is likely a direct result of NuPoCo's resource allocation at the granularity of sockets. Performance is better than or comparable to NuPoCo in most cases except for *(8)BT+KM*. This may be attributed to NuPoCo's focus on overall system utilization in contrast to MAPPER's focus on ensuring QoS.

4.5.6 Sensitivity Analysis. We evaluate the impact of varying MAPPER's *minQoS* and decision-making interval using the combined workload mixes from Figure 5. Performance improvements of 29%, 28%, 20%, and 20% were obtained with MAPPER when compared to Linux for QoS thresholds of 0.75, 0.8, 0.85, and 0.9, respectively. For these workload mixes, a QoS of 0.75, 0.8, and 0.85 did not result in any *minQoS* violations. A QoS setting of 0.9 violated *minQoS* for six applications. These violations have a geometric mean of 0.87, ranging from 0.84 to 0.89. This behavior demonstrates the limitations in approximating IPC information as application performance, as well as in adjusting to dynamic application behavior, on MAPPER's ability to manage QoS at high precision.

We also measured the impact of changing the decision making interval for the same combined workload mixes (from Figure 5). For intervals of 20 *ms*, 100 *ms*, and 500 *ms*, speedup over Linux is 11%, 18%, and 20%, respectively (20% with the default 1 *sec* interval). Changes to `cpuset` can take effect in less than 2 *ms*, enabling MAPPER to respond quickly to changes in application behavior. MAPPER is capable of adapting to smaller decision-making intervals and can be combined with dynamic-interval-based control techniques [4] to better respond to unpredictable applications. However, the performance penalty does increase significantly for very fine granularities (less than 100 *ms*), as overheads due to disturbances and migrations start to dominate.

4.5.7 Implementation Overhead. We use Linux's perf subsystem's interface (`perf_event_open` [26]) to monitor desired events on a per thread basis. We find the overhead of using this interface on application performance is negligible and within measurement error (ranging from 0.08%–0.56%). The single-threaded MAPPER daemon is responsible for performance counter setup and reading, for determining resource allocation, and for performing the mapping (the latter including cgroups to effect the mapping). When using a regulation interval of 1 second, MAPPER's daemon takes an average of 5 milliseconds (ms) when monitoring 80 threads, consuming 0.5% of a single CPU's time. Amortized over the 80 hardware contexts in the system, this amounts to an overhead of 0.00625%. Of the 5 ms, 2.7 ms is spent setting up/reading performance counters[3] for each application and constituent thread, with most of the remaining time spent in the mapping function.

5 CONCLUSIONS

We present a system to manage application performance via parallel efficiency regulation (MAPPER) that provides performance portability in both standalone and shared environments. We show that a normalized efficiency metric allows comparisons across and cooperation among applications, enabling better QoS guarantees for each application while at the same time improving system-wide efficiency. MAPPER is able to combine low-overhead hardware counter-based performance monitoring with sharing- and resource-aware mapping and cooperative control of application parallelism to minimize resource contention and data communication costs, eliminate over-subscription, and improve efficiency. MAPPER improves performance by up to 3.3× with average speedups of 1.17× over Linux for multi-programmed workload mixes across all our test cases. At the same time, MAPPER violates QoS for only 2% of the applications (compared to 23% for Linux), while placing much tighter bounds on the worst case. OpenMP application mixes achieve a geometric mean speedup of 1.44× over Linux, while Cloudsuite benchmark mixes perform 1.11× better than Linux. MAPPER is able to improve individual Cloudsuite benchmark performance by 24.6%, 14%, and 11% for Memcached, In-memory-analytics, and Mediastreaming services, respectively. MAPPER achieves these performance improvements while violating QoS (85% of fair-share performance) in only 5% (1) of the workload mixes compared to 47% (9) of the workload mixes for Linux.

REFERENCES

[1] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. 2017. *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores.* Morgan & Claypool Publishers. DOI: https://doi.org/10.2200/S00774ED1V01Y201704DTM045

[2] T. E. Anderson, B. N. Berchad, E. D. Lazowska, and H. M. Levy. 1992. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.* 10 1 (1992), 53–79.

[3] It is important to note the performance of this subsystem can vary across kernel versions [48].

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks. *Int. J. Supercomput. Applic.* 5, 3 (1991), 63–73. DOI:https://doi.org/10.1177/109434209100500306

[4] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. 2003. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, 275–287. DOI:https://doi.org/10.1145/859618.859650

[5] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 538–550. DOI:https://doi.org/10.1109/HPCA.2015.7056061

[6] Major Bhadauria and Sally A. McKee. 2010. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, 189–199. DOI:https://doi.org/10.1145/1810085.1810113

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

[8] Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. 2016. ESTIMA: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM. DOI:https://doi.org/10.1145/2851141.2851159

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. DOI:https://doi.org/10.1109/IISWC.2009.5306797

[10] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. 2018. Maximizing system utilization via parallelism management for co-located parallel applications. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. Association for Computing Machinery, New York, NY. DOI:https://doi.org/10.1145/3243176.3243199

[11] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. 2015. LIRA: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'15)*. ACM. DOI:https://doi.org/10.1145/2768405.2768407

[12] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Retrieved from https://software.intel.com/en-us/articles/intel-sdm.

[13] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, 127–144. DOI:https://doi.org/10.1145/2541940.2541941

[14] Tanima Dey, Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2013. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013). DOI:https://doi.org/10.1145/2555289.2555298

[15] A. J. Dorta, C. Rodriguez, and F. de Sande. 2005. The OpenMP source code repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing*. 244–250. DOI:https://doi.org/10.1109/EMPDP.2005.41

[16] Daniel Ellsworth, Tapasya Patki, Swann Perarnau, Sangmin Seo, Abdelhalim Amer, Judicael Zounmevo, Rinku Gupta, Kazutomo Yoshii, Henry Hoffman, Allen Malony, Martin Schulz, and Pete Beckman. 2016. Systemwide power management with Argo. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 1118–1121.

[17] Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015. Pow: System-wide dynamic reallocation of limited power in HPC. In *Proceedings of the 24th International Symposium on High-performance Parallel and Distributed Computing*. ACM, 145–148.

[18] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[19] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 323–336. Retrieved from http://dl.acm.org/citation.cfm?id=1972457.1972490.

[20] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, 254–269. DOI:https://doi.org/10.1145/3064176.3064177

[21] Tim Harris, Martin Maas, and Virendra J. Marathe. 2014. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM. DOI:https://doi.org/10.1145/2592798.2592807

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, 295–308. Retrieved from http://dl.acm.org/citation.cfm?id=1972457.1972488.

[23] Geoffrey E. Hinton. 2012. A practical guide to training restricted Boltzmann machines. In *Neural Networks: Tricks of the Trade - Second Edition*. Springer, 599–619.

[24] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance isolation for commercial latency-sensitive services. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*. USENIX Association, 519–531. Retrieved from http://dl.acm.org/citation.cfm?id=3277355.3277406.

[25] Nikhil Jain, Abhinav Bhatele, Michael P. Robson, Todd Gamblin, and Laxmikant V. Kale. 2013. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High-performance Computing, Networking, Storage and Analysis (SC'13)*. ACM. DOI: https://doi.org/10.1145/2503210.2503263

[26] Linux kernel manpages. 2018. perf_event_open - set up Performance Monitoring. Retrieved from http://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[27] K. A. Klues. 2015. *OS and Runtime Support for Efficiently Managing Cores in Parallel Applications*. PhD thesis. University of California, Berkeley.

[28] Yehuda Koren. 2008. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 426–434.

[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX, 31–46. Retrieved from https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola.

[30] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, 450–462. DOI: https://doi.org/10.1145/2749469.2749475

[31] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.

[32] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, 248–259. DOI: https://doi.org/10.1145/2155620.2155650

[33] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, 237–250. DOI: https://doi.org/10.1145/1755913.1755938

[34] M. Needham, R. Hui, S. Dwarkadas, and X. Qiu. 2011. Parallelization of gene differential association analysis. *BMC Bioinf.* 12, 374 (Sept. 2011).

[35] Tapti Palit, Yongming Shen, and Michael Ferdman. 2016. Demystifying cloud benchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 122–132.

[36] Swann Perarnau, Rajeev Thakur, Kamil Iskra, Ken Raffenetti, Franck Cappello, Rinku Gupta, Pete Beckman, Marc Snir, Henry Hoffmann, Martin Schulz, and Barry Rountree. 2015. Distributed monitoring and management of exascale systems in the Argo project. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 173–178.

[37] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, 325–341. DOI: https://doi.org/10.1145/3132747.3132780

[38] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 145–160. Retrieved from http://dl.acm.org/citation.cfm?id=3291168.3291180.

[39] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 423–432. DOI: https://doi.org/10.1109/MICRO.2006.49

[40] Ruslan Salakhutdinov and Andriy Mnih. 2008. Bayesian probabilistic matrix factorization using Markov Chain Monte Carlo. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*. 880–887.

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the SIGOPS European Conference on Computer Systems (EuroSys)*. 351–364. Retrieved from http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[42] Janaina Schwarzrock, Charles C. de Oliveira, Marcus Ritt, Arthur F. Lorenzon, and Antonio Carlos S. Beck. 2021. A runtime and non-intrusive approach to optimize EDP by tuning threads and CPU frequency for OpenMP applications. *IEEE Trans. Parallel Distrib. Syst.* 32, 7 (2021), 1713–1724. DOI: https://doi.org/10.1109/TPDS.2020.3046537

[43] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Association for Computing Machinery, New York, NY, 169–180. DOI: https://doi.org/10.1145/2594291.2594292

[44] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data sharing or resource contention: Toward performance transparency on multicore systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, 529–540. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/srikanthan.

[45] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2016. Coherence stalls or latency tolerance: Informed CPU scheduling for socket and core sharing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 323–336. Retrieved from https://www.usenix.org/conference/atc16/technical-sessions/presentation/srikanthan.

[46] Sharanyan Srikanthan, Princeton Ferro, Sandhya Dwarkadas, and Sayak Chakraborti. 2019. Managing application parallelism via parallel efficiency regulation. In *Poster Presented at PPoPP 2019, Washington, DC, United States*. DOI: https://doi.org/details/PPoPP-2019-Posters/4/Managing-Application-Parallelism-via-Parallel-Efficiency-Regulation

[47] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. Association for Computing Machinery, New York, NY, 277–286. DOI: https://doi.org/10.1145/1346281.1346317

[48] Vincent M. Weaver. 2013. Linux perf_event features and overhead. In *Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. Retrieved from http://web.eece.maine.edu/~vweaver/projects/perf_events/overhead/fastpath2013_perfevents.pdf.

[49] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *ACM SIGARCH Comput. Archit. News* 44, 2 (2016), 545–559.

[50] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-scale parallel collaborative filtering for the Netflix prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (LNCS* Vol. 5034). Springer, 337–348.

Received April 2021; revised November 2021; accepted November 2021