Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems

Sharanyan Srikanthan Sandhya Dwarkadas Kai Shen
Department of Computer Science, University of Rochester
{srikanth,sandhya,kshen}@cs.rochester.edu

Abstract

Modern multicore platforms suffer from inefficiencies due to contention and communication caused by sharing resources or accessing shared data. In this paper, we demonstrate that information from low-cost hardware performance counters commonly available on modern processors is sufficient to identify and separate the causes of communication traffic and performance degradation. We have developed SAM, a Sharing-Aware Mapper that uses the aggregated coherence and bandwidth event counts to separate traffic caused by data sharing from that due to memory accesses. When these counts exceed pre-determined thresholds, SAM effects task to core assignments that colocate tasks that share data and distribute tasks with high demand for cache capacity and memory bandwidth. Our new mapping policies automatically improve execution speed by up to 72% for individual parallel applications compared to the default Linux scheduler, while reducing performance disparities across applications in multiprogrammed workloads.

1 Introduction

Multicore processors share substantial hardware resources including last-level cache (LLC) space and memory bandwidth. At the same time, a parallel application with multiple tasks[1] running on different CPU cores may simultaneously access shared data. Both data and resource sharing can result in performance slowdowns and symptoms including high data traffic (bandwidth consumption) and high LLC miss rates. To maximize efficiency, multicore platforms on server, desktop, as well as mobile environments must intelligently map tasks to CPU cores for multiprogrammed and parallel workloads.

Despite similar symptoms, data and resource sharing behaviors require very different task→CPU mapping policies—in particular, applications with strong data sharing benefit from colocating the related tasks on cores that are in proximity to each other (e.g., cores on one socket) while applications with high memory demand or large working sets might best be distributed across sockets with separate cache and memory bandwidth resources. The mapping efficiency is further complicated by the dynamic nature of many workloads.

A large body of prior work has devised scheduling techniques to mitigate resource contention [5–7, 14, 15, 17, 18, 21, 22] in the absence of data sharing. Relatively few have investigated data sharing issues [19, 20] for parallel applications. Tam et al. [19] used direct sampling of data access addresses to infer data sharing behavior. While more recent multicore platforms do sometimes provide such sampled tracing capability, trace processing in software comes at a significant cost. Tang et al. [20] demonstrated the behavior of latency-critical datacenter applications under different task placement strategies. Their results reinforce the need to provide a low-cost, online, automated approach to place simultaneously executing tasks on CPUs for high efficiency.

This paper presents our operating system strategy for task placement that manages data sharing and resource contention in an integrated fashion. In order to separate the impact of data sharing from resource contention, we use aggregate information from existing hardware performance counters along with a one-time characterization of event thresholds that impact performance significantly. Specifically, we use performance counters that identify on- and off-chip traffic due to coherence activity (when data for a cache miss is sourced from another core) and combine this knowledge with LLC miss rates and bandwidth consumption to separate sharing-related slowdown from slowdown due to resource contention.

Our adaptive online *Sharing-Aware Mapper* (**SAM**) uses an iterative, interval-based approach. Based on the sampled counter values in the previous interval, as well as measured thresholds in terms of performance impact using microbenchmarks, we identify tasks that share data and those that have high memory and/or cache demand. We then perform a rebalance in order to colocate the tasks that share data on cores that are in proximity and with potentially shared caches. This decision is weighed against the need to distribute tasks with high bandwidth uses across cores that share fewer resources.

SAM improves the execution speed by up to 72% for stand-alone parallel applications compared to the default Linux scheduler, without the need for user input or ma-

[1] In this paper, a task refers to an OS-schedulable entity such as a process or a thread.

nipulation. For concurrent execution of multiple parallel and sequential applications, our performance improvements are up to 36%, while at the same time reducing performance disparities across applications. The rest of this paper presents the design, implementation, and evaluation of our sharing-aware mapping of tasks to CPUs on multicore platforms.

2 Sharing and Contention Tracking

Hardware counters are commonplace on modern processors, providing detailed information such as the instruction mix, rate of execution, and cache/memory access behavior. These counters can also be read at low latency (on the order of a μSec). We explore the use of commonly available event counters to efficiently track data sharing as well as resource contention on multicores.

Our work addresses several challenges. First, predefined event counters may not precisely suit our information tracking needs. In particular, no single counter reports the data sharing activities on multicores. Secondly, it may be challenging to identify event thresholds that should trigger important control decisions (such as saturating uses of a bottleneck resource). Finally, while many events may be defined in a performance counter architecture, usually only a small number can be observed at one time. For example, the Intel Ivy Bridge architecture used in our evaluation platform can only monitor four programmable counters at a time (in addition to three fixed-event counters).

In this paper, we use these low-cost performance counters to infer valuable information on various bottlenecks in the system. Particularly, we focus on intra- and inter-socket coherence activity, memory bandwidth utilization, and access to remote memory (NUMA). We use the counters and microbenchmarks to analyze the effect of these factors on performance. We obtain thresholds for memory bandwidth utilization and coherence activity that result in significant performance degradation. These thresholds further enable our system to identify and mitigate sharing bottlenecks during execution.

2.1 Coherence Activities

Coherence can be a significant bottleneck in large scale systems. In multithreaded applications, access to shared data and synchronization variables trigger coherence activities when the threads are distributed across multiple cores. When data-sharing threads are colocated on the same multicore socket, coherence is handled within the socket using a high speed bus or a ring. When threads are distributed across sockets, the coherence cost increases significantly due to the higher latency of off-chip access.

Despite the gamut of events monitored by modern day processors, accounting for coherence activity in a manner portable across platforms can still be a challenge. Our goal is to identify performance counters that are available across a range of architectural performance monitoring units, and that can help isolate intra- and inter-socket coherence. We use the following four counters—last-level cache (LLC) hits, LLC misses, misses at the last private level of cache, and remote memory accesses.

In multi-socket machines, there is a clear increase in overhead when coherence activities cross the socket boundary. Any coherence request that can be resolved from within the socket is handled using an intra-socket protocol. If the request cannot be satisfied locally, it is treated as a last-level cache (LLC) miss and handed over to an inter-socket coherence protocol.

We use the cache misses at the last private level, as well as LLC hits and misses, to indirectly infer the intra-socket coherence activities. LLC hit counters count the number of accesses served directly by the LLC and do not include data accesses satisfied by intra-socket coherence activity. Thus, by subtracting LLC hits and misses from the last private level cache misses, we can determine the number of LLC accesses that were serviced by the intra-socket coherence protocol.

To measure inter-socket coherence activity, we exploit the fact that the LLC treats accesses serviced by both off-socket coherence as well as by memory as misses. The difference between LLC misses and memory accesses gives us the inter-socket coherence activity. In our implementation, the Intel Ivy Bridge processor directly supports counting LLC misses that were not serviced by memory, separating and categorizing them based on coherence state. We sum the counters to determine inter-socket coherence activities.

We devise a synthetic microbenchmark to help analyze the performance impact of cross-socket coherence activities. The microbenchmark creates two threads that share data. We ensure that the locks and data do not induce any false sharing. Using a dual-socket machine, we compare the performance of the microbenchmark when consolidating both threads on a single socket against execution when distributing them across sockets (a common result of Linux's default scheduling strategy). The latter induces inter-socket coherence traffic while all coherence traffic uses the on-chip network in the former case.

We vary the rate of coherence activity by changing the ratio of computation to shared data access within each loop in order to study its performance impact. Figure 1 shows the performance of consolidating the threads onto the same socket relative to distributing across sockets. At low coherence traffic rates, the consolidation and distribution strategies do not differ significantly in performance, but when the traffic increases, the performance improvement from colocation can be quite substantial.

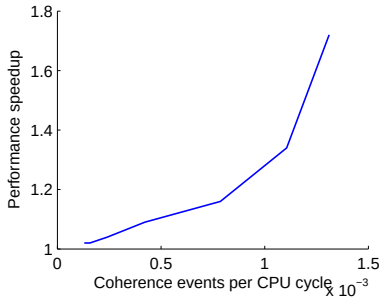

Figure 1: Speedup when consolidating two threads that share data onto the same socket (eliminating inter-socket coherence) in comparison to distributing them across sockets as coherence activity is varied.

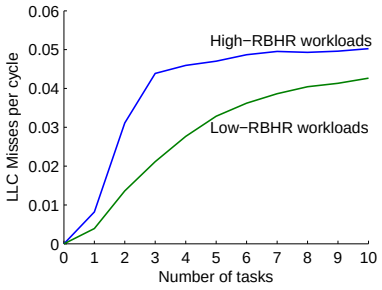

Figure 2: Socket-level memory bandwidth usage (measured by the LLC_MISSES performance counter) for workloads of high and low row buffer hit ratios (RBHRs) at increasing number of tasks.

We use these experiments to identify a per-thread coherence activity threshold at which inter-socket coherence causes substantial performance degradation (which we identify as a degradation of $>5\%$). Specifically on our experimental platform with 2.2 GHz processors, the threshold is 2.5×10^{-4} coherence events per CPU cycle, or 550,000 coherence events per second.

2.2 Memory Bandwidth Utilization

Our approach requires the identification of a memory bandwidth utilization threshold that signifies the resource exhaustion and likely performance degradation. Since all cores on a socket share the access to memory, we account for this resource threshold on a per-socket basis—aggregating the memory bandwidth usage of all tasks running on each particular socket.

One challenge we face is that the maximum bandwidth utilization is not a static hardware property but it further depends on the row buffer hit ratio (RBHR) of the memory access loads. DRAM rows must be pre-charged and activated before they can be read or written to. For spatial locality, DRAMs often activate an entire row of data (on the order of 4 KB) instead of just the cache line being accessed. Once this row is opened, subsequent accesses to the same row incur much lower latency and consume less device time. Hence, the spatial locality in an application's memory access pattern plays a key role in its performance and resource utilization.

Figure 2 shows the aggregate memory bandwidth used with increasing numbers of tasks. All tasks in each test run on one multicore socket in our machine. We show results for both low- and high-RBHR workloads, using a memory copy microbenchmark where the number of contiguous words copied is one cache line within a row or an entire row, respectively. Behaviors of most real-world applications lie between these two curves. We can clearly see that the bandwidth available to a task is indeed affected by its RBHR. For example, on our machine, three tasks with high RBHR may utilize more memory bandwidth than ten low-RBHR tasks do. If we use the maximum bandwidth usage of high-RBHR workloads as the available memory resource, then a low-RBHR workload would never reach it and therefore be always determined as having not used the available resource (even when the opposite is true).

In this paper, we are more concerned with detecting and mitigating memory bandwidth bottlenecks than utilizing the memory bandwidth at its maximum possible level. We can infer from Figure 2 that the difference between the low / high-RBHR bottleneck bandwidths (at 10 tasks) is about 10%. We conservatively use the low RBHR bandwidth as the maximum available without attempting to track RBHR. Our high-resource-use threshold is set at 20% below the above-defined maximum available memory bandwidth. Specifically on our experimental platform with 2.2 GHz processors, the threshold is 0.034 LLC misses per cycle, or 75,000,000 LLC misses per second. A conservative setting might result in detecting bottlenecks prematurely but will avoid missing ones that will actually result in performance degradation.

2.3 Performance Counters on Ivy Bridge

Our experimental platform contains processors from Intel's Ivy Bridge line. On our machine, cores have private L1 and L2 caches, and an on-chip shared L3 (LLC). The Intel Performance Monitoring Unit (PMU) provides the capability of monitoring certain events (each of which may have several variants) at the granularity of individual hardware contexts. Instructions, cycles, and unhalted cycles can be obtained from fixed counters in the PMU. The remaining events must use the programmable counters.

We encounter two constraints in using the programmable counters—there are only four programmable counters and we can monitor only two variants of any particular event using these counters. Solutions to both constraints require multiplexing the programmable counters. We separate the counters into disjoint groups and then

alternate the groups monitored during successive intervals of application execution, effectively sampling each counter group from a partial execution.

In order to monitor intra-socket coherence activity, we use the main event MEM_LOAD_UOPS_RETIRED. One caveat is that since the event is a load-based event, only reads are monitored. In our experiments, we found that since typical long-term data use involves a read followed by a write, these counters were a good indicator of coherence activity. We use three variants of this event, namely L2_MISS (misses at the last private level of cache), L3_HIT (hits at, or accesses serviced by, the LLC), and L3_MISS (misses at the LLC), and use these events to infer the intra-socket coherence activity, as described in Section 2.1.

To obtain inter-socket coherence activity, we use the MEM_LOAD_UOPS_LLC_MISS_RETIRED event. We get the inter-socket coherence activity by summing up two variants of this event—REMOTE_FWD and REMOTE_HITM.

We read the remote DRAM access event count from MEM_LOAD_UOPS_LLC_MISS_RETIRED: REMOTE_DRAM.

We use the LLC_MISSES event (which includes both reads and writes) as an estimate of overall bandwidth consumption. Although the LLC_MISSES event count includes inter-socket coherence and remote DRAM access events, since these latter are prioritized in our algorithm, further division of bandwidth was deemed unnecessary.

In total, we monitor seven programmable events, with three variants for each of two particular events, allowing measurement of one sample for each event across two iterations. Since different interval sizes can have statistical variation, we normalize the counter values with the measured unhalted cycles for the interval.

3 SAM: Sharing-Aware Mapping

We have designed and implemented SAM, a performance monitoring and adaptive mapping system that simultaneously reduces costly communication and maximizes resource utilization efficiency for the currently executing tasks. We identify resource sharing bottlenecks and their associated costs/performance degradation impact. Our mapping strategy attempts to shift load away from such bottleneck resources.

3.1 Design

Using thresholds for each bottleneck as described in Section 2, we categorize each task based on whether its characteristics exceed these thresholds. The four activity categories of interest are: inter-socket coherence, intra-socket coherence, remote DRAM access, and per-socket memory bandwidth demand.

Figure 3 defines the symbols that are used for SAM's task→CPU mapping algorithm illustrated in Figure 4. Reducing inter-socket coherence activity by colocating tasks has the highest priority. Once tasks with high inter-socket coherence traffic are identified, we make an attempt to colocate the tasks by moving them to sockets with idle cores that already contain tasks with high inter-socket coherence activity, if there are any. If not, we use our task categorization to distinguish CPU or memory bound tasks—SAM swaps out memory intensive tasks only after swapping CPU bound tasks, and avoids moving tasks with high intra-socket coherence activity. Moving memory-intensive tasks is our reluctant last choice since they might lead to expensive remote memory accesses.

Our second priority is to reduce remote memory accesses. Remote memory accesses are generally more expensive than local memory accesses. Since a reluctant migration might cause remote memory accesses, we register the task's original CPU placement prior to the migration. We then use this information to relocate the task back to its original socket whenever we notice that it incurs remote memory accesses. To avoid disturbing other tasks, we attempt to swap tasks causing remote memory accesses with each other whenever possible. Otherwise, we search for idle or CPU-bound tasks to swap for the task that generates remote memory accesses.

After reducing remote memory accesses, we look to balancing memory bandwidth utilization. SAM identifies memory-intensive tasks on sockets where the bandwidth is saturated. The identified tasks are relocated to other sockets whose bandwidth is not entirely consumed. We track the increase in bandwidth utilization after every migration to avoid overloading a socket with too many memory-intensive tasks. In some situations, such relocation can cause an increase in remote memory accesses, but this is normally less damaging than saturating the memory bandwidth.

3.2 Implementation Notes

Performance counter statistics are collected on a per task (process or thread) basis with the values accumulated in the task control block. Performance counter values are read on every operating system tick and the values are attributed to the currently executing task.

Our mapping strategy is implemented as a kernel module that is invoked by a privileged daemon process. The collected counter values for currently executing tasks are examined at regular intervals. Counter values are first normalized using unhalted cycles and then used to derive values for memory bandwidth utilization, remote memory accesses, and intra- and inter-socket coherence activity for each running task. These values are attributed to

$\mathcal{C}_T$: *Per task inter-socket coherence threshold*
M_T : *Per task memory utilization threshold*
$\mathcal{R}_T$: *Per task remote memory access threshold*
$\mathcal{S}$: *Set of all sockets*
$\mathcal{C}_i$: *Set of cores in socket i*
$\mathcal{C}_{i,j}^{inter}$: *Inter − socket coherence activity generated by core j on socket i*
$\mathcal{C}_{i,j}^{intra}$: *Intra − socket coherence activity generated by core j on socket i*
$M_{i,j}$: *Memory bandwidth utilization by core j on socket i*
$\mathcal{R}_{i,j}$: *Remote memory accesses by core j on socket i*
$\mathcal{C}ycles_{i,j}^{inter}$: *Cycles count for core j on socket i*
$\mathcal{P}_i^{Mem} : \{\ j \mid j\ \in \mathcal{C}_i \wedge M_{i,j} > M_T\ \}$
$\mathcal{P}_i^{Rem} : \{\ j \mid j\ \in \mathcal{C}_i \wedge R_{i,j} > R_T\ \}$
$\mathcal{P}_i^{Idle} : \{\ j \mid j\ \in \mathcal{C}_i \wedge \mathcal{C}ycles_{i,j} = 0\}$
$\mathcal{P}_i^{inter} : \{\ j \mid j\ \in \mathcal{C}_i \wedge \mathcal{C}_{i,j}^{inter} > \mathcal{C}_T\ \}$
$\mathcal{P}_i^{intra} : \{\ j \mid j\ \in \mathcal{C}_i \wedge \left(\mathcal{C}_{i,j}^{inter} \leq \mathcal{C}_T\right) \wedge \left(\mathcal{C}_{i,j}^{intra} \geq \mathcal{C}_T\right)\}$
$\mathcal{P}_i^{CPU} : \mathcal{C}_i - \mathcal{P}_i^{Mem} - \mathcal{P}_i^{Idle} - \mathcal{P}_i^{inter} - \mathcal{P}_i^{intra}$
$\mathcal{M}_i : \sum_{j \in \mathcal{C}_i} \mathcal{M}_{i,j}$
$Soc_{task(i,j)}$: *Original socket from which the task currently running on socket i, core j was reluctantly migrated.*

Figure 3: SAM algorithm definitions.

the corresponding core and used to update/maintain per-core and per-socket data structures that store the consolidated information. Per-core values are added to determine per-socket values.

Task relocation is accomplished by manipulating task affinity (using `sched_setaffinity` on Linux) to restrict scheduling to specific cores or sockets. This allows seamless inter-operation with Linux's default scheduler and load balancer.

SAM's decisions are taken at 100 mSec intervals. We performed a sensitivity analysis on the impact of the interval length and found that while SAM was robust to changes in interval size (varied from 10 mSecs to over 1 second), a 100-mSec interval hit the sweet spot in terms of balancing reaction times. We find that we can detect and decide on task migrations effectively at this rate. We were able to obtain good control and response for intervals up to one second. Effecting placement changes at intervals higher than a second can reduce the performance benefits due to lack of responsiveness.

4 Evaluation

We assess the effectiveness of our hardware counter-based sharing and contention tracking, and evaluate the performance of our adaptive task→CPU mapper. Our performance monitoring infrastructure was implemented in Linux 3.14.8. Our software environment is Fedora 19 running GCC 4.8.2. We conducted experiments on a dual-socket machine with each socket containing an Intel Xeon E5-2660 v2 "Ivy Bridge" processor (10 physical cores with 2 hyperthreads each, 2.20 GHz, 25 MB of L3 cache). The machine has a NUMA configuration in which each socket has an 8 GB local DRAM partition.

```
// Inter-socket coherence activity found. Need to colocate appropriate tasks.
for every i ∈ S if (P_i^inter != ∅)
    for every j ∈ S ∧ (j != i)
        while ((|P_i^inter + P_i^intra| < |C_i|) ∧ (P_j^inter != ∅))
            while (P_i^Idle != ∅ ∧ P_j^inter != ∅)
                move (P_j^inter[0], P_i^Idle[0])
            while (P_i^CPU != ∅ ∧ P_j^inter != ∅)
                swap (P_j^inter[0], P_i^CPU[0])
            while (P_i^Mem != ∅ ∧ P^inter != ∅)
                swap (P_j^inter[0], P_i^Mem[0]}, let a = P_i^Mem[0]
                // This is a reluctant task migration. Store original
                // socket id to restore task when possible.
                if (Soc_task(j,a) == −1)
                    Soc_task(j,a) = i

// Mitigate any remote memory accesses encountered.
for every i ∈ S if (P_i^Rem != ∅)
    for every j ∈ P_i^Rem, let a = Soc_task(i,j)
        // Swap with a task migrated reluctantly from the current socket.
        if (∃_{k∈C_a}(Soc_task(a,k) == i))
            swap (j, k)
        else if (P_a^Idle != ∅)
            move (j, P_a^Idle[0])
        else if (P_a^CPU != ∅)
            swap (j, P_a^CPU[0])
        else if (P_a^Mem != ∅)
            swap (j, P_a^Mem[0]), let b = P_a^Mem[0]
            if (Soc_task(a,b) == −1)
                Soc_task(a,b) = i

// Balance the memory intensive load to other sockets.
for every i ∈ S if (P_i^Mem != ∅) ∧ (M_i > M_T)
    for every j ∈ S if ((j != i) ∧ (M_j < M_T))
        // Balance memory intensive tasks across sockets.
        // If cores are unavailable, look for other sockets to balance load.
        res = balance(P_i^Mem, P_j^Mem)
        if (res == Balance_Sucessful)
            break
```

Figure 4: SAM task→CPU mapping algorithm.

4.1 Benchmarks

We use a variety of benchmarks from multiple domains in our evaluation. First, we use the synthetic *microbenchmarks* described in Sections 2.1 and 2.2 that stress the inter- and intra-socket coherence and memory bandwidth respectively. We create two versions of the coherence activity microbenchmark—one (Hubench) generating a near-maximum rate of coherence activity (1.3×10^{-3} coherence events per CPU cycle) and another (Lubench) generating coherence traffic close to the threshold we identified in Section 2 (2.6×10^{-4} coherence events per cycle). We also use the high-RBHR *memcpy* microbenchmark (MemBench). MemBench saturates the memory bandwidth on one socket and needs to be distributed across sockets to maximize memory bandwidth utilization.

PARSEC 3.0 [2] is a parallel benchmark suite containing a range of applications including image processing, chip design, data compression, and content similarity search. We use a subset of the PARSEC benchmarks that exhibit non-trivial data sharing. In particular, Canneal uses cache-aware simulated annealing to minimize the routing cost of a chip design. Bodytrack is a computer vision application that tracks a human body through an image sequence. Both benchmarks depend on data that

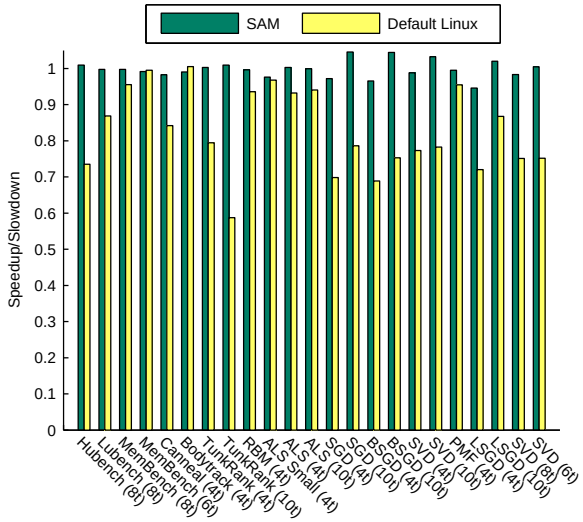

Figure 5: Performance of standalone applications on Linux with SAM and without (default Linux) SAM. The performance metric is the execution speed (the higher the better) normalized to that of the best static task→CPU mapping determined through offline testing.

is shared among its worker threads. We observe that the data sharing however is not as high as other workloads that we discuss below.

The *SPEC CPU2006* [1] benchmark suite has a good blend of memory-intensive and CPU-bound applications. Many of its applications have significant memory utilization with high computation load as well. The memory-intensive applications we use are libq, soplex, mcf, milc, and omnetpp. The CPU-bound applications we use are sjeng, bzip2, h264ref, hmmer, and gobmk.

GraphLab [13] and *GraphChi* [12] are emerging systems that support graph-based parallel applications. Unlike most PARSEC applications, the graph-based applications tend to have considerable sharing across worker threads. In addition, the number of worker threads is not always static and is dependent on the phase of execution and amount of parallelism available in the application. Such phased behaviors are a good stress test to ascertain SAM's stability of control. Our evaluation uses a range of machine learning and filtering applications—TunkRank (Twitter influence ranking), Alternating Least Squares (ALS) [23], Stochastic gradient descent (SGD) [11], Singular Value Decomposition (SVD) [10], Restricted Bolzman Machines (RBM) [8], Probabilistic Matrix Factorization (PMF) [16], Biased SGD [10], and Lossy SDG [10].

4.2 Standalone Application Evaluation

We first evaluate the impact of SAM's mapping decisions on standalone parallel application executions. Figure 5 shows the performance obtained by SAM and the default Linux scheduler. The performance is normalized to that of the best static task→CPU mapping obtained offline for each application.

Application	Coherence	Remote memory	Workload characteristic
Hubench (8t)	4	0	Data sharing
Lubench (8t)	4	0	Data sharing
MemBench (8t)	0	0	Memory bound
MemBench (6t)	0	0	Memory bound
Canneal (4t)	8	3	CPU bound
Bodytrack (4t)	2	0	CPU bound
TunkRank (4t)	6	1	Data sharing
TunkRank (10t)	13	1	Data sharing
RBM (4t)	12	0	CPU bound
ALS (4t) Small Dataset	1	0	CPU bound
ALS (4t)	6	3	CPU bound
ALS (10t)	9	1	CPU bound
SGD (4t)	12	2	Data sharing
SGD (10t)	14	3	Data sharing
BSGD (4t)	8	2	Data sharing
BSGD (10t)	20	3	Data sharing
SVD (4t)	12	1	Data sharing
SVD (10t)	24	7	Data sharing
PMF (4t)	20	0	CPU bound
LSGD (4t)	6	3	Data sharing
LSGD (10t)	18	3	Data sharing
SVD (8t)	22	6	Data sharing
SVD (6t)	18	6	Data sharing

Table 1: Actions taken by our scheduler for each standalone application run. *Coherence* indicates the number of task migrations performed to reduce inter-socket coherence. *Remote memory* indicates the number of task migrations performed to reduce remote memory accesses. *Workload characteristic* classifies each application as either CPU bound, data sharing intensive, or memory bound.

We can see that SAM considerably improves the application performance. Our task placement mirrors that of the best configuration obtained offline, thereby resulting in almost identical performance. Improvement over the default Linux scheduler can reach 72% in the best case and varies in the range of 30–40% for most applications. This performance improvement does not require any application changes, application user/deployer knowledge of system topology, or need for recompilation.

Parallel applications that have nontrivial data sharing among its threads benefit from SAM's remapping strategy. Figure 6(A) shows the average per-thread instructions per unhalted cycle (IPC). SAM results in increased IPC for almost all the applications. Figure 6(B) demonstrates that we have almost eliminated all the per-thread inter-socket coherence traffic, replacing it with intra-

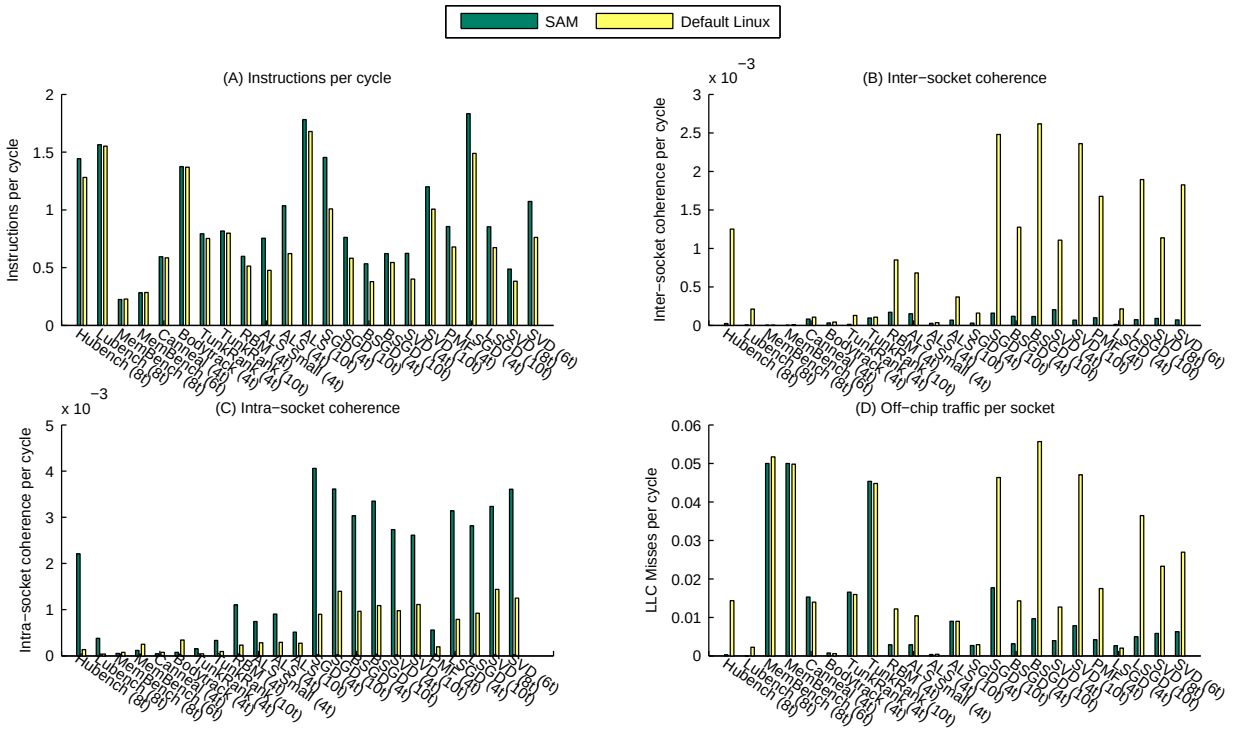

Figure 6: Measured hardware metrics for standalone applications. Fig. (A): per-thread instructions per unhalted cycle (IPC); Fig. (B): per-thread inter-socket coherence activity; Fig. (C): per-thread intra-socket coherence activity; Fig. (D): per-socket LLC misses per cycle. All values are normalized to unhalted CPU cycles.

socket coherence traffic (Figure 6(C)). Figure 6(D) uses LLC misses per socket per cycle to represent aggregate per socket off-chip traffic. We can see that off-chip traffic for applications with high data sharing is reduced. Table 1 summarizes the mapping actions (whether due to inter-socket coherence or remote memory access) taken for every workload shown in Figure 5.

Interestingly, although Figure 6(B) shows that TunkRank (Twitter influence ranking) has fairly low average inter-socket coherence activity, it shows the most performance boost with SAM. In this application, there are phases of high coherence activity between phases of heavy computation, which SAM is able to capture and eliminate. The application also scales well with additional processors (high inherent parallelism). The better a data-sharing application scales, the higher the speedup when employing SAM. With SAM, TunkRank scales almostlinearly on our multicore platform.

While most workloads achieve good speedup, in some cases SAM's performance is only on par with (or slightly better than) default Linux. Such limited benefits are due to two primary reasons. First, applications may be CPU bound without much coherence or memory traffic. ALS, RBM, and PMF are examples of such applications. Although inter-socket coherence activity for these applications is above threshold, resulting in some migrations (see Table 1) due to SAM's affinity control, the relative impact of this colocation is small. Second, some workloads are memory intensive but contain low inter-socket coherence activity. Linux uses a static heuristic of distributing tasks across sockets, which is already the ideal action for these workloads. In such cases, SAM does not have additional room to improve performance.

Table1 does not contain a column for migrations to balance memory bandwidth because for standalone applications, SAM does not effect such migrations. First, for purely memory intensive workloads (MemBench), the default strategy to distribute tasks works ideally and therefore we do not perform any more migrations. Second, for workloads that are both memory intensive and with high data sharing, SAM prioritizes inter-socket coherence activity avoidance over balancing memory bandwidth utilization.

Graph-based machine learning applications exhibit phased execution and dynamic task creation and parallelism (mainly in the graph construction and distribution stages). The burst of task creation triggers load balancing in Linux. While Linux respects core affinity, tasks may be migrated to balance load. Linux task migrations do not sufficiently consider application characteristics and may result in increased remote DRAM accesses, inter-socket communication, resource contention, and bandwidth sat-

uration. SAM is able to mitigate the negative impact of these load balancing decisions, as demonstrated by the non-trivial number of migrations due to coherence activity and remote memory accesses performed by SAM (Table 1). SAM constantly monitors the system and manipulates processor affinity (which the Linux load balancer respects) to colocate or distribute tasks across sockets and cores.

4.3 Multiprogrammed Workload Evaluation

Table 2 summarizes the various application mixes we employed to evaluate SAM's performance on multiprogrammed workloads. They are designed to produce different levels of data sharing and memory utilization. Two factors come into play. First, applications incur performance penalties due to resource competition. Second, bursty and phased activities disturb the system frequently.

In order to compare the performance of different mixes of applications, we first normalize each application's runtime with its offline optimum standalone runtime, as in Figure 5. We then take the geometric mean of the normalized performance of all the applications in the workload mix (counting each application once regardless of the number of tasks employed) to derive a single metric that represents the speedup of the mix of applications. We use this metric to compare the performance of SAM against the default Linux scheduler.

The performance achieved is shown in Figure 7. Naturally, due to competition for resources, most applications will run slower than their offline best standalone execution. Even if applications utilize completely difference resources, they may still suffer performance degradation. For example, when inter-socket coherence mitigation conflicts with memory load balancing, no scheduler can realize the best standalone performance for each co-executing application.

SAM outperforms the default Linux scheduler by 2% to 36% as a result of two main strategies. First, we try to balance resource utilization whenever possible without affecting coherence traffic. This benefits application mixes that have both applications that share data and use memory. Second, we have information on inter-socket coherence activity and can therefore use it to colocate tasks that share data. However, we do not have exact information to indicate which tasks share data with which other tasks. We receive the validation of a successful migration if it produces less inter-socket and more intra-socket activity after the migration. Higher intra-socket activity helps us identify partial or full task groupings inside a socket.

For the mixes of microbenchmarks (#1–#3), SAM executes the job about 25% faster than Linux. SAM's relative performance approaches 1—indicating comparable performance to the case that all microbenchmarks reach respective optimum offline standalone performance simultaneously. SAM is able to preferentially colocate tasks that share data over tasks that are memory intensive. Since the memory benchmark saturates memory bandwidth at fairly low task counts, colocation does not impact performance.

Multiprog. workload #	Application mixes
1	12 MemBench, 8 HuBench
2	14 MemBench, 6 HuBench
3	10 MemBench, 6 HuBench, 4 CPU
4	2 libq, 2 bzip2, 2 sjeng, 2 omnetpp
5	2 libq, 2 soplex, 2 gobmk, 2 hmmer
6	2 mcf, 2 milc, 2 sjeng, 2 h264ref
7	2 milc, 2 libq, 2 h264ref, 2 sjeng
8	2 mcf, 2 libq, 2 h264ref, 2 sjeng, 4 TunkRank
9	10 SGD, 10 BSGD
10	10 SGD, 10 LSGD
11	10 LSGD, 10 BSGD
12	10 LSGD, 10 ALS
13	10 SVD, 10 SGD
14	10 SVD, 10 BSGD
15	10 SVD, 10 LSGD
16	8 SVD, 8 LSGD
17	20 SGD, 20 BSGD
18	20 SGD, 20 LSGD
19	20 LSGD, 20 BSGD
20	20 LSGD, 20 ALS
21	20 SVD, 20 SGD
22	20 SVD, 20 BSGD
23	20 SVD, 20 LSGD
24	16 BSGD, 10 MemBench, 14 CPU
25	16 LSGD, 10 MemBench, 14 CPU
26	16 SGD, 10 MemBench, 14 CPU

Table 2: Multiprogrammed application mixes. For each mix, the number preceding the application's name indicates the number of tasks it spawns. We generate various combinations of applications to evaluate scenarios with varying data sharing and memory utilization.

The speedups obtained for the SPEC CPU benchmark mixes relative to Linux are not high. The SPEC CPU mix of applications tend to be either memory or CPU bound. Since Linux prefers to split the load among sockets, it can perform well for memory intensive workloads. We perform significantly better than the Linux scheduler when the workload mix comprises applications that share data and use memory bandwidth.

A caveat is that Linux's static task→CPU mapping policy is very dependent on the order of task creation. For example, different runs of the same SPECCPU mix of applications discussed above can result in a very different sequence of task creation, forcing CPU-bound tasks to be colocated on a socket and memory-intensive tasks

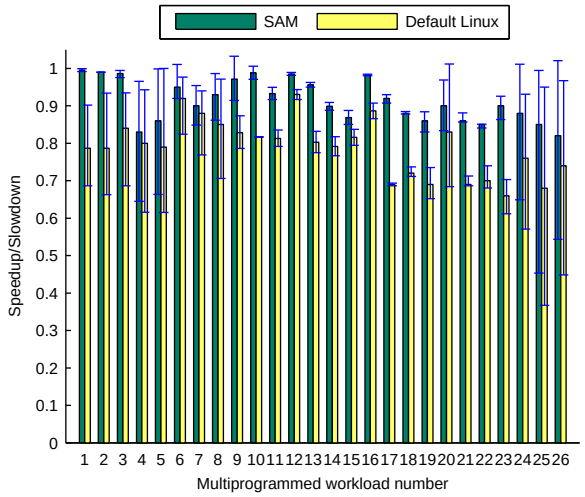

Figure 7: Speedup (execution time normalized to the offline optimum standalone execution: higher is better) of multiprogrammed workloads with (SAM) and without (default Linux) SAM. We show the geometric mean of all application speedups within each workload as well as the max-min range (whiskers) for the individual applications.

to be colocated on the other. We use the best-case Linux performance as our comparison base. Since SAM observes and reacts to the behavior of individual tasks, its scheduling performance does not depend heavily on the task creation order.

Figure 7 also plots the minimum and maximum speedups of applications in each workload mix using whiskers. The minimum and maximum speedups are important to understand the fairness of our mapping strategy. The application that has the minimum speedup is slowed down the most in the workload mix. The application that has the maximum speedup is the least affected by the contention in the system. We can see that SAM improves upon the fairness of the Linux scheduler in a significant fashion. The geometric mean of the minimum speedup for all the workload mixes for SAM is 0.83. The same for the default Linux scheduler is 0.69. Similarly, the geometric mean of the maximum speedup for all workload mixes for SAM and Linux are 0.96 and 0.87 respectively. From this analysis, we can conclude that in addition to improving overall performance, SAM reduces the performance disparity (a measure of fairness) among multiple applications when run together.

Figure 8(A) plots the per-thread instructions per cycle for the mixed workloads. We can see that largely, the effect of our actions is to increase the IPC. As with the standalone applications, Figure 8(B) shows that SAM significantly reduces inter-socket coherence activity, replacing it with intra-socket coherence activity (Figure 8(C)). Note that for some workloads, SAM shows reductions in both intra- and inter-socket coherence. This is likely due to working sets that exceed the capacity of the private caches, resulting in hits in the LLC when migrations are effected to reduce inter-socket coherence.

Figure 8(D) uses LLC misses per cycle to represent aggregate per socket off-chip traffic. Our decision to prioritize coherence activity can lead to reduced off-chip traffic, but this may be counteracted by an increase in the number of remote memory accesses. At the same time, our policy may also reduce main memory accesses by sharing the last level cache with tasks that actually share data (thereby reducing pressure on LLC cache capacity). We also improve memory bandwidth utilization when possible without disturbing the colocated tasks that share data. Workload mixes #24–#26 have a combination of data-sharing, memory-intensive, and CPU-bound tasks. In these cases, SAM improves memory bandwidth utilization by moving the CPU-bound tasks to accommodate distribution of the memory-intensive tasks.

In our experiments, both hardware prefetcher and hyperthreading are turned on by default. Hyperthreads add an additional layer of complexity to the mapping process due to resource contention for logical computational units as well as the private caches. Since SAM utilizes one hardware context on each physical core before utilizing the second, when the number of tasks is less than or equal to the number of physical cores, SAM's policy decisions are not affected by hyperthreading.

In order to determine the interaction of the prefetcher with the SAM mapper, we compare the relative performance of SAM when turning off prefetching. We observe that on average, prefetching is detrimental to the performance of multiprogrammed workloads with or without the use of SAM. The negative impact of prefetching when using SAM is slightly lower than with default Linux.

4.4 Overhead Assessment

SAM's overhead has three contributors: accessing performance counters, making mapping decisions based on the counter values, and migrating tasks to reflect the decisions. In our prototype, reading the performance counters, a cost incurred on every hardware context, takes takes 8.89 μSecs. Counters are read at a 1-mSec interval. Mapping decisions are centralized and are taken at 100 mSecs intervals. Each call to the mapper, including the subsequent task migrations, takes about 9.97 μSecs. The overall overhead of our implementation is below 1%. We also check the overall system overhead by running all our applications with our scheduler but without performing any actions on the decisions taken. There was no discernible difference between the two runtimes, meaning that the overhead is within measurement error.

Currently, SAM's policy decisions are centralized in

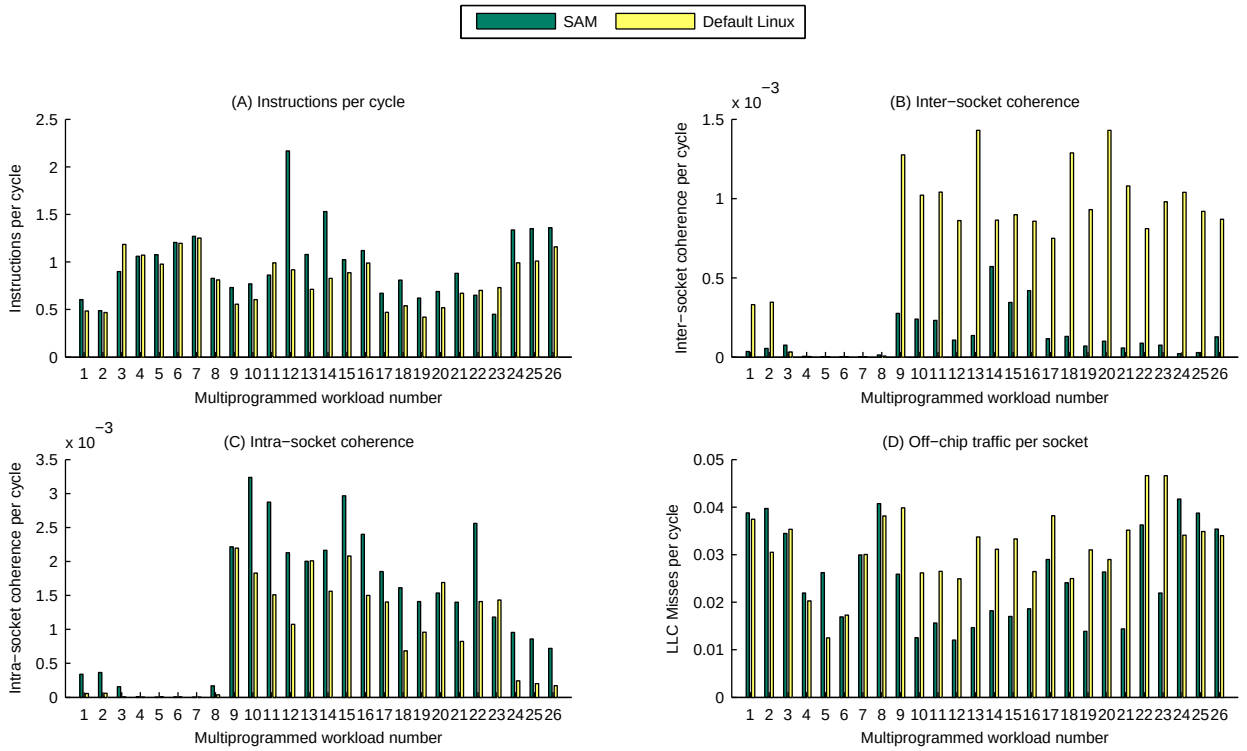

Figure 8: Measured hardware metrics for multiprogrammed application workloads. Fig. (A): per-thread instructions per unhalted cycle (IPC); Fig. (B): per-thread inter-socket coherence activity; Fig. (C): per-thread intra-socket coherence activity; Fig. (D): per-socket LLC misses per cycle. All values are normalized to unhalted CPU cycles.

a single daemon process, which works well for our 40-CPU machine, since most of the overhead is attributed to periodically reading the performance counters. Data consolidation in the daemon process has a linear complexity on the number of processors and dominates in cost over the mapping decisions. For all practical purposes, SAM scales linearly with the number of processors. In the future, if the total number of cores is very large, mapping policies might need to be distributed for scalability.

4.5 Sensitivity Analysis

We study the sensitivity of SAM's performance gains to the thresholds identified in Section 2 for determining high coherence activity and memory bandwidth consumption.

Too low a coherence activity threshold can classify applications with little benefit from colocation as ones with high data sharing; it can also result in misclassifying tasks that have just been migrated as ones with high data sharing due to the coherence activity generated as a result of migration. For some workloads, this may reduce the ability to beneficially map tasks with truly high data sharing. Too high a coherence activity threshold can result in not identifying a task as high data sharing when it could benefit from colocation.

We found that SAM's performance for our workloads was relatively resilient to a wide range of values for the threshold. The coherence activity threshold could be varied from 10% of the value determined in Section 2 to 2 times this value. Using threshold values below the low end result in a loss of performance of up to 9% for one mixed workload. Using threshold values $>$2 times the determined threshold for the mixed workloads and $>$3 times for the standalone applications results in a loss of up to 18% and 30% performance respectively for a couple of workloads.

SAM's resilience with respect to the memory bandwidth threshold is also quite good, although the range of acceptable values is tighter than for coherence activity. For the memory bandwidth threshold, too low a value can lead to prematurely assuming that a socket's memory bandwidth is saturated, resulting in lost opportunities for migration. Too high a value can result in performance degradation due to bandwidth saturation. In the case of the SPECCPU benchmarks, lowering our determined threshold by 30% resulted in both sockets being considered as bandwidth saturated, and a performance loss of up to 27% due to lost migrations opportunities. Similarly, for MemBench, raising the memory threshold by 25% leads to non-recognition of bandwidth saturation, which produces up to 85% loss in performance.

5 Related Work

The performance impact of contention and interference on shared multicore resources (particularly the LLC cache, off-chip bandwidth, and memory) has been well recognized in previous work. Suh et al. [18] used hardware counter-assisted marginal gain analysis to minimize the overall cache misses. Software page coloring [5, 22] can effectively partition the cache space without special hardware features. Mutlu et al. [15] proposed parallelism-aware batch scheduling in DRAM to reduce inter-task interference at the memory level. Mars et al. [14] utilized active resource pressures to predict the performance interference between colocated applications. These techniques, however, manage multicore resource contention without addressing the data sharing issues in parallel applications.

Blagodurov et al. [3] examined data placement in non-uniform memory partitions but did not address data sharing-induced coherence traffic both within and across sockets. Tam et al. [19] utilized address sampling (available on Power processors) to identify task groups with strong data sharing. Address sampling is a relatively expensive mechanism to identify data sharing (compared to our performance counter-based approach) and is not available in many processors in production today.

Calandrino and Anderson [4] proposed cache-aware scheduling for real-time schedulers. The premise of their work is that working sets that do not fit in the shared cache will cause thrashing. The working set size of an application was approximated to the number of misses incurred at the shared cache. Their scheduling is aided with job start times and execution time estimates, which non-real time systems often do not have. Knauerhase et al. [9] analyze the run queue of all processors of a system and schedule them so as to minimize cache interference. Their scheduler is both fair and less cache contentious. They do not, however, consider parallel workloads and the effect of data sharing on the last level cache.

Tang et al. [20] demonstrate the impact of task placement on latency-critical datacenter applications. They show that whether applications share data and/or have high memory demand can dramatically affect performance. They suggest the use of information on the number of accesses to "shared" cache lines to identify intra-application data sharing. While this metric is a useful indicator of interference between tasks, it measures only one aspect of data sharing—the cache footprint, but misses another important aspect of data sharing—off-chip traffic in the case of active read-write sharing. Additionally, their approach requires input statistics from an offline stand-alone execution of each application.

Previous work has also pursued fair uses of shared multicore resources between simultaneously executing tasks. Ebrahimi et al. [6] proposed a new hardware design to track contention at different cache/memory levels and throttle tasks with unfair resource usage or disproportionate progress. At the software level, fair resource use can be accomplished through scheduling quantum adjustment [7] or duty cycle modulation-enabled speed balancing [21]. These techniques are complementary and orthogonal to our placement strategies, and can be used in conjunction with our proposed approach for improved fairness and quality of service.

6 Conclusions

In this paper, we have designed and implemented a performance monitoring and sharing-aware adaptive mapping system. Our system works in conjunction with Linux's default scheduler to simultaneously reduce costly communications and improve resource utilization efficiency. The performance monitor uses commonly available hardware counter information to identify and separate data sharing from DRAM memory access. The adaptive mapper uses a cost-sensitive approach based on the performance monitor's behavior identification to relocate tasks in an effort to improve both parallel and mixed workload application efficiency in a general-purpose machine. We show that performance counter information allows us to develop effective and low-cost mapping techniques without the need for heavy-weight access traces. For stand-alone parallel applications, we observe performance improvements as high as 72% without requiring user awareness of machine configuration or load, or performing compiler profiling. For multiprogrammed workloads consisting of a mix of parallel and sequential applications, we achieve up to 36% performance improvement while reducing performance disparity across applications in each mix. Our approach is effective even in the presence of workload variability.

Acknowledgments This work was supported in part by the U.S. National Science Foundation grants CNS-1217372, CCF-1217920, CNS-1239423, CCF-1255729, CNS-1319353, CNS-1319417, and CCF-137224, and by the Semiconductor Research Corporation Contract No. 2013-HJ-2405.

References

[1] SPECCPU2006 benchmark. www.spec.org.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention man-

agement on multicore systems. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[4] J. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st EUROMICRO Conf. on Real-Time Systems (ECRTS)*, Dublin, Ireland, July 2009.

[5] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 455–468, Orlando, FL, Dec. 2006.

[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, Pittsburgh, PA, Mar. 2010.

[7] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 25–38, Brasov, Romania, Sept. 2007.

[8] G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade - Second Edition*, pages 599–619. 2012.

[9] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[10] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 426–434, Las Vegas, NV, 2008.

[11] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, Aug. 2009.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 31–46, Hollywood, CA, 2012.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, CA, July 2010.

[14] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *44th Int'l Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, Dec. 2011.

[15] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 63–74, Beijing, China, June 2008.

[16] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using Markov Chain Monte Carlo. In *25th Int'l Conf. on Machine Learning (ICML)*, pages 880–887, Helsinki, Finland, 2008.

[17] K. Shen. Request behavior variations. In *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 103–116, Pittsburg, PA, Mar. 2010.

[18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, Apr. 2004.

[19] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Second EuroSys Conf.*, pages 47–58, Lisbon, Portugal, Mar. 2007.

[20] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 283–294, San Jose, CA, June 2011.

[21] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2009.

[22] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *4th EuroSys Conf.*, pages 89–102, Nuremberg, Germany, Apr. 2009.

[23] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proc. 4th Intl Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008.