Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing

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

Abstract

The efficiency of modern multiprogrammed multicore machines is heavily impacted by traffic due to data sharing and contention due to competition for shared resources. In this paper, we demonstrate the importance of identifying latency tolerance coupled with instruction-level parallelism on the benefits of colocating threads on the same socket or physical core for parallel efficiency. By adding hardware counted CPU stall cycles due to cache misses to the measured statistics, we show that it is possible to infer latency tolerance at low cost. We develop and evaluate SAM-MPH, a multicore CPU scheduler that combines information on sources of traffic with tolerance for latency and need for computational resources. We also show the benefits of using a history of past intervals to introduce hysteresis when making mapping decisions, thereby avoiding oscillatory mappings and transient migrations that would impact performance. Experiments with a broad range of multiprogrammed parallel, graph processing, and data management workloads on 40-CPU and 80-CPU machines show that SAM-MPH obtains ideal performance for standalone applications and improves performance by up to 61% over the default Linux scheduler for mixed workloads.

1 Introduction

Modern multi-socket multicore machines present a complex challenge in terms of performance portability and isolation, especially for parallel applications in multiprogrammed workloads. Performance is heavily impacted by traffic due to data sharing and contention due to competition for shared resources, including physical cores, caches, memory, and the interconnect.

Significant effort has been devoted to mitigating the impact on performance of competition for shared resources [7, 15, 17, 18, 23, 26] for applications that do not share data. Our own past work, Sharing-Aware Mapper (SAM) [22], has shown that inter-socket coherence activity among threads can be used as a criterion for same-socket thread colocation for improved system efficiency and parallel application performance. Using existing x86 performance counters, SAM is able to separate inter-socket traffic due to coherence from that due to memory access, favoring colocation for threads experiencing high coherence traffic, and distribution for threads with high cache and/or memory bandwidth demand. Although SAM is able to infer inter-socket coherence traffic, it cannot determine the impact of the coherence activity on application performance. This inability to relate traffic to performance limits its effectiveness to prioritize tasks in multiprogrammed workloads.

In this paper, we develop and evaluate a multicore CPU scheduler that combines information on sources of traffic with tolerance for latency and need for computational resources. First, we demonstrate the importance of identifying latency tolerance in determining the benefits of colocating threads on the same socket or physical core on parallel efficiency in multiprogrammed workloads. High inter-socket coherence activity does not translate to proportional benefit from thread colocation for different applications or threads within an application. The higher the latency hiding ability of the thread, the lower the impact of inter-socket coherence activity on performance. We infer the benefits of colocation using commonly available hardware performance counters, in particular, CPU stall cycles on cache misses. A low value for CPU stall cycles is an indication of latency tolerance, making stall cycles an appropriate metric for prioritizing threads for colocation.

Second, we focus on the computational needs of individual threads. Hyperthreading [13], where hardware computational contexts share a physical core, present complex tradeoffs for applications that share data. Colocating threads on the same physical core allows direct access to a shared cache thereby resulting in low latency data communication when fine-grain sharing (sharing of cache-resident data) is exhibited across threads. At the same time, competition for functional units can reduce the instructions per cycle (IPC) for the individual threads relative to running on independent physical cores. The benefits of colocation are therefore a function of granularity of sharing (whether the reads and writes by different threads occur while the data is still cache resident) as well as the instruction-level parallelism available within

each thread. We find that a combination of IPC and coherence activity thresholds are sufficient to inform this tradeoff.

Finally, we show that utilizing interval history in phase classification can avoid the oscillatory and transient task migrations that may result from merely *reacting* to immediate past behavior [8]. In particular, we keep track of the level of coherence activity that was incurred by a thread in prior intervals, as well as its tolerance for latency, and use this information to introduce hysteresis when identifying a phase transition.

The combination of these three optimizations enable us to obtain ideal performance for standalone applications and improve performance by up to 61% over linux for multiprogrammed workloads. Performance of multiprogrammed workloads improve on average by 27% and 43% over standard Linux on 40- and 80-CPU machines respectively. When compared with SAM [22], our approach yields an average improvement of 9% and 21% on the two machines with a peak improvement of 24% and 27%. We also reduce performance disparity across the applications in each workload. These performance benefits are achieved with very little increase in monitoring overhead.

2 Background: Separating Traffic due to Sharing and Memory Access

This work builds on our prior effort of SAM [22], a Sharing-Aware-Mapper that monitors individual task [1] behavior using hardware performance counters. SAM identifies and combines information from commonly available hardware performance counter events to separate traffic due to data sharing from that due to memory access. Further, the non-uniformity in traffic is captured by separately characterizing intra- and inter-socket coherence activity, and local versus remote memory access.

Following an iterative, interval-based approach, SAM uses the information about individual task traffic patterns to retain colocation for tasks with high intra-socket coherence activity, and consolidate tasks with high inter-socket coherence activity. At the same time, SAM distributes tasks with high memory bandwidth needs, colocating them with the memory they access. These decisions reduce communication and contention for resources by localizing communication whenever possible.

While SAM is able to separate and identify coherence traffic from memory bandwidth needs, it does not currently determine the impact of the traffic on performance; in other words, its ability to tolerate the latency of communication, which would allow task prioritization for constrained resources. Additionally, SAM currently does not differentiate between logical hardware contexts and physical cores. Furthermore, while SAM's successive iterations are able to capture changes in application behavior and workload mixes to effect task placement changes, it merely *reacts* to the current state of task placement to effect a more efficient task placement. Thus, it misses opportunities to learn from past placement decisions as well as to adapt to periodicity in application behavior. Our goal in this paper is to address these shortcomings in SAM and realize multiprogrammed performance much closer to standalone static best placement.

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

3 Identifying Latency Tolerance and Computational Needs

In this section, we demonstrate the importance of identifying latency tolerance and computational needs in multicore task placement, and show how this information may be inferred from widely available performance counter events.

3.1 Tolerance for Coherence Activity Latency

Data sharing across tasks can result in varying communication latencies primarily dictated by task placement. The closer the tasks sharing the data, the lower the latency. For example, when tasks that share data are placed across sockets, the need to move data across sockets results in substantially increased latency. Hence, the natural choice would be to prioritize tasks with high coherence activity for colocation on the same socket.

However, the performance impact of coherence activities depends in reality on the latency tolerance of the application. We focus here on identifying this latency tolerance in addition to being able to measure and identify data sharing.

We introduce two additional metrics to augment inter-socket coherence activity as a measure of sharing behavior: IPC (Instructions per Cycle) and SPC (Stalls per inter-socket Coherence event). The Intel platforms provide access to a counter that tracks cycles stalled on all last-level cache misses. While these stalls include those due to coherence activity, they also include stalls on other forms of misses. When coherence activity is high, stalls are dominated by coherence activity, and thus stalls on cache misses can be used as an approximation of stalls due to coherence misses. SPC is thus approximated to be stalls on all last-level cache misses divided by the number of coherence events in the specific time interval.

For low to moderate coherence activity levels, the above approximation for SPC can no longer be justified. In such cases, we use IPC as an indicator of latency tolerance. Higher IPC is generally achieved with high

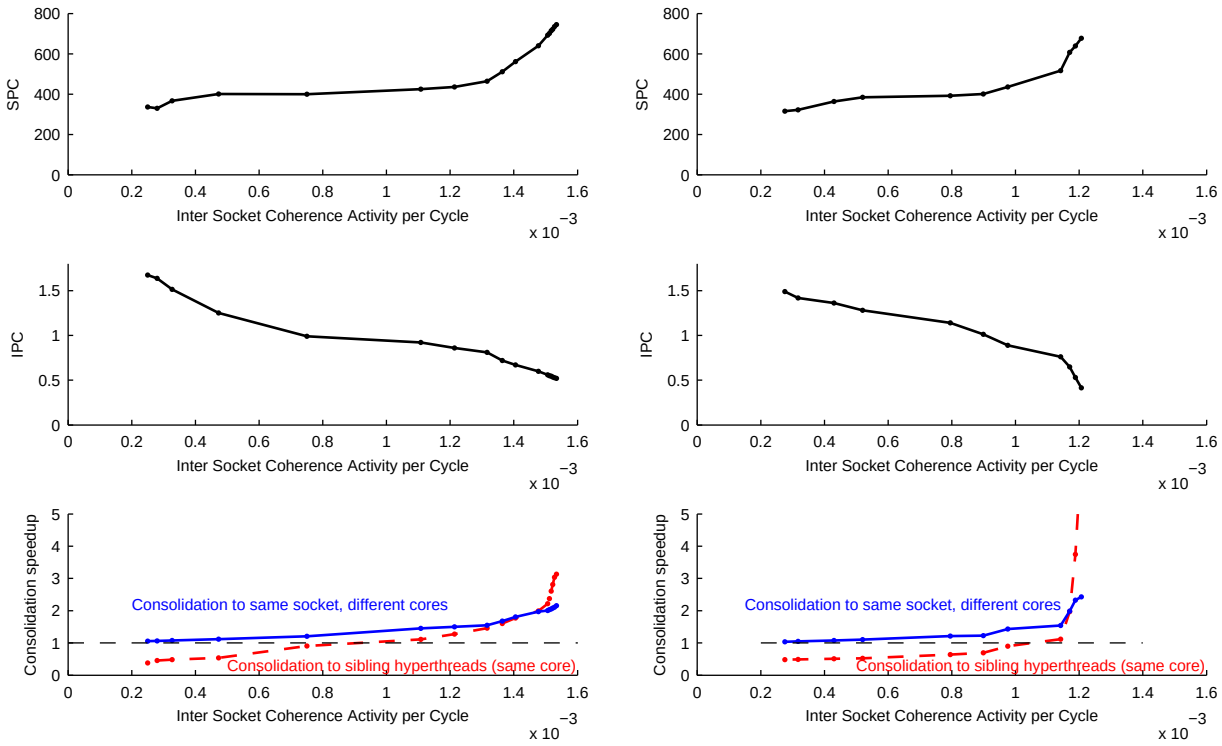

Figure 1: SPC (stalls per coherence event), IPC (instructions per cycle), and speedup (relative to running on separate sockets) when consolidating threads onto the same socket on different physical cores (blue curve) and same physical core (red dashed curve), as a function of inter-socket coherence activity controlled by a microbenchmark with a (left) higher instruction-level parallelism (ILP) and (right) lower ILP computational loop.

instruction-level parallelism, which provides the ability to hide access latency.

In general, higher SPC implies lower latency hiding potential; higher IPC implies more instruction-level parallelism that can be used to hide latency. We use a combination of IPC and SPC in order to be able to predict a task's latency hiding potential. In order to demonstrate the importance of SPC and IPC in making placement decisions, we designed a microbenchmark that is parameterized to control coherence activity. Coherence activity is generated by a *coherence activity* loop that alternates increments by two threads to a set of shared counters that are guaranteed to fit in the level of cache closest to the processor. The frequency of coherence activity is controlled by adding to the *coherence activity* loop a computational loop consisting of (a) additions on registers or (b) additions/multiplications on independent registers thereby increasing instruction-level parallelism, and varying the number of iterations of this loop. Figure 1 presents SPC, IPC, and speedup obtained by consolidating threads on the same socket (or physical core) relative to running on different sockets as the ratio of computation to coherence activity in each outer loop is varied using the two different computational loops. As coherence activity (inter-socket coherence events per cycle) increases, speedup from consolidation mirrors SPC, but the performance gains and inflection points depend on the application. For example, at roughly the same coherence activity of 1.2×10^{-3} coherence events/cycle, consolidation on different physical cores in the same socket results in a speedup of 1.5 for the microbenchmark on the left and 2.4 for the one on the right. The corresponding SPC is 436 and 677, respectively, allowing clear prioritization of the microbenchmark on the right for consolidation.

3.2 Placement on Hyperthreads

Modern processors are often equipped with hyperthreads or other equivalent logical hardware contexts that share the processor pipeline and private caches. Hyperthreads improve processor occupancy and efficiency by providing multiple instruction streams/hardware contexts to keep functional units busy. At the same time, since the hardware contexts share pipeline resources and private caches, contention for these resources can slow the performance of the individual contexts, presenting performance isolation challenges.

Placing tasks that share data on hyperthreads that share

a physical core allows shared data to be accessed directly from the private caches (without traffic on the intra- and inter-socket interconnects), with the potential for a significant performance boost. This advantage is conditional on the data being retained in the cache until the time of access by the sharing task, requiring temporal proximity of the accesses to the shared data.

Parameter Thresholds: Figure 1 shows the impact on performance for different task placement strategies. When coherence activity is moderate ($< 0.78 \times 10^{-3}$) and IPC is sufficiently high (> 0.9), colocating tasks on different physical cores in the same socket results in the best performance. The resource contention introduced by hyperthreading results in slowdowns that are not overcome by the potential for direct access to shared data (the shared counter) from the private cache shared by the tasks. Hence, if placement on the same socket requires sharing physical cores, distributing tasks across sockets works better than colocating them on hyperthreads. However, when coherence activity increases further and IPC is less than 0.9, the benefits of colocating tasks that share data exceed the cost of contention. At very high levels of data sharing, indicated by high coherence activity ($> 0.78 \times 10^{-3}$) and SPC values in excess of 550, the benefits of hyperthreading greatly exceed that of using different physical cores on the same socket. In such cases, consolidation on hyperthreads in the same physical core can provide both performance and energy savings using Intel's powerstepping (the latter has not been explored in this paper).

To summarize, when coherence activity is high ($> 0.78 \times 10^{-3}$), (higher) SPC is used to prioritize applications for consolidation and (lower) IPC is used to determine whether hyperthreading is beneficial. When coherence activity is moderate or low ($< 0.78 \times 10^{-3}$), (higher) IPC is used to prioritize distribution over consolidation.

3.3 Setting up Performance Counters

We use two machines to evaluate our performance: a dual-socket IvyBridge and a quad-socket Haswell. Each processor from the two microarchitectures contain 10 physical cores with 2 hardware contexts/hyperthreads each. Each physical core has a private L1 and L2 cache and share a last level L3 cache with other cores in the processor. We use SAM's infrastructure [22] to access the hardware performance counters provided by Intel's PMU (Performance Monitoring Unit). Each hardware context has only four programmable counters in addition to the fixed counters: instructions, cycles, and unhalted cycles. We use time multiplexing across two time periods to sample eight performance counters.

We use the same counters as SAM to monitor intra-socket coherence activity, inter-socket coherence activity, remote DRAM accesses, and overall bandwidth consumption. Additionally, we also monitor the CYCLE_ACTIVITY: STALL_CYCLES_L2_PENDING event to count the stalls on cache accesses. The counts are normalized using unhalted cycles for the interval of their collection and accumulated in a data structure maintained in the task control block.

4 Latency Tolerance- and Sharing-Aware Mapping

4.1 Implementation Mechanisms

Our mapper is implemented as a kernel module that is executed by a daemon process with root privilege. Hardware performance counters are read at every system timer interrupt (tick) and the information gathered is stored in the task control block of the currently running task. Our mapper is invoked every 100ms, at which time data from the currently executing tasks is consolidated into per-application and per-socket data structures. The daemon maintains a per application data structure for currently active (executing) applications in order to allow grouping of tasks belonging to the same application (based on address space). This data structure keeps track of application history, including current application classification. The per-socket data structure used for determining memory bandwidth requirements and remote memory accesses is identical to that used in SAM [22].

Once the mapper decides on task to core mapping, task migration is effected by updating processor affinity. We use the `sched_setaffinity` kernel call to update the processor affinity of a task. Note that task migration takes place at the next Linux scheduling operation and is required only if tasks are not already colocated. Our decision making logic thereby co-exists with other load balancing operations in Linux.

4.2 Data Consolidation

The hardware performance counter values collected and stored in task control blocks are used to infer inter- and intra-socket coherence activity, memory bandwidth utilization, remote memory accesses, and latency tolerance. The inferred values are then used to categorize tasks as having: 1) low, medium, or high coherence activity; 2) low or high memory bandwidth demand; and 3) low or high IPC, based on thresholds as discussed in Section 3.

The per-task information is aggregated to obtain per-socket information on memory bandwidth and inter-socket coherence activity. The per-task information is also used to categorize the parent application as incur-

ring 1) low, medium, or high coherence activity; and 2) low or high IPC, based on the task with the most coherence activity.

4.3 Mode-based Phase Identification

SAM's original adaptation strategy is *reactive* in that changes in application behavior (phase identification) in one interval trigger a potential re-mapping in the next interval. Reactive adaptation works well when application behavior is relatively stable with few transitions. However, frequent phase transitions can potentially lead to oscillating placement decisions with a resulting reduction rather than improvement in performance.

In this paper, we explore the use of history over multiple past intervals [8]. For each application, a history of interval classification — whether the interval was classified as incurring medium or high coherence activity and high compute intensity (IPC) levels — is maintained for the last n intervals. This history is maintained in three 64 bit integers using shift operations (to shift in a 1 when a particular interval exhibits the behavior). Bit masking and counting are used to determine the occurrence count of each of the bottlenecks. If the occurrence count for a bottleneck exceeds a threshold, we identify the application as suffering from the bottleneck in the next interval. The recent inclusion of the `popcnt` instruction in the Intel ISA results in very fast bit counting operations, allowing low overhead examination. The occurrence threshold builds hysteresis into this feedback control loop, thereby preventing oscillatory behavior. In our implementation, we set n to 10 and the occurrence threshold to 6. We analyze the sensitivity of this threshold and its impact on performance in Section 5.4.

If the number of intervals with high coherence activity exceeds the predefined threshold, the current interval is classified as incurring high coherence activity even if the performance counters for the current interval reflect low coherence activity. This strategy for classification helps avoid task migrations due to transient application behavior as well as avoids oscillatory mappings due to frequent phase transitions.

A cumulative count of the stalls due to inter-socket coherence activity, instructions executed, and cycles elapsed since the last phase transition is maintained in order to calculate SPC and IPC. In an interval with low inter-socket coherence activity and high intra-socket coherence activity, accumulation is suppressed in order to retain SPC and IPC information from the interval where the phase transition was detected. The goal is to retain SPC and IPC information gathered during an inter-socket placement (prior to colocation) for the purposes of prioritization. Based on thresholds, if the classification changes, the cumulative counters are reset to the values for the current interval.

4.4 Hyperthread and Latency Tolerance-Aware Mapping Policy

Presuming that all hardware contexts are busy, the mapping task consists of placing m tasks on m hardware contexts so that there is a $1:1$ correspondence between tasks and hardware contexts. Applications in the highest coherence activity phase are prioritized and mapped first. These applications are sorted by their SPC values and scheduled in order. Applications whose SPC values are not known are placed at the end of the list, but are still scheduled ahead of applications with low data sharing. For each application, tasks that share data with each other are selected for colocation by updating their core affinities. If tasks do need colocation, we look for a socket that has not been assigned any task during the current round of mapping. If a sufficient number of cores in a single socket cannot be found, we colocate the threads on the least number of sockets possible.

We then look at applications with moderate levels of activity. They are sorted in order of IPC and applications with the smallest IPC are prioritized for colocation. When we encounter threads with IPC values greater than the IPC threshold (0.9), we alter the mapping to prohibit the threads from sharing the same physical core. If such a situation is unavoidable inside the same socket, we look to other sockets to determine if performance loss can be avoided. Alternately, if the SPC value of high data sharing applications is more than 550, tasks of that application are preferentially placed on hyperthreads to derive benefits from very high data sharing.

5 Evaluation

Our evaluation was conducted on two machines. The first is our development platform and is a dual-socket machine, equipped with 2.2 GHz Intel Xeon E5-2660 v2 processors from the Ivy Bridge architecture. Each socket can accommodate up to 20 hardware contexts on 10 physical cores, sharing a last-level cache of 25MB. Each socket is directly connected to 8GB local DRAM memory, resulting in non-uniform access to a total of 16GB DRAM memory. This machine is labeled *40-CPU IvyBridge*.

The second machine contains four sockets, equipped with 1.9 GHz Intel Xeon E7-4820 v3 processors from the Haswell architecture. Each processor accommodates up to 20 hardware contexts and has up to 64 GB of local DRAM per socket for a total of 256GB DRAM. This machine is labeled *80-CPU Haswell*.

The operating system we use is Fedora 19 and the kernel was compiled using GCC 4.8.2. Linux kernel (ver-

sion 3.14.8) was modified to accommodate the changes needed for our techniques.

We compare the performance of our sharing-aware mapper, SAM-MPH, with that of SAM [22] and default Linux. In order to attribute the performance improvements in SAM-MPH, we also show incremental performance gained due to identifying and prioritizing task placement based on latency tolerance (SAM-M), hyperthreading aware mapping (SAM-H), and using history across multiple intervals to indentify phase changes (SAM-P).

5.1 Benchmarks

We use a combination of microbenchmarks, SPEC-CPU [1], PARSEC [3], and several parallel and graph-based benchmarks in order to evaluate SAM-MPH.

Similar to SAM, we use microbenchmarks to stress the coherence protocol and memory bandwidth. HuBench and LuBench contain pairs of threads sharing data with each other to generate coherence traffic. HuBench generate high coherence activity and is very sensitive to data sharing latencies while LuBench has enough thread-private computation to hide its data sharing latency. MemBench is a memory intensive microbenchmark that uses multiple threads to access thread-private memory in a streaming fashion. These threads saturate memory bandwidth on one socket, therefore benefiting from distribution across sockets to maximize resource utilization.

GraphLab [16] and *GraphChi* [14] are recent application development tools specially suited for graph-based parallel applications. Unlike the SPECCPU and most PARSEC applications, graph-based processing involves considerable data sharing across several workers. These application are also much more dynamic, with tasks actively being created and deleted, and going through phases of computation that are vastly different in characteristics, depending on the type of problem and the active parallelism available in the input data. The unpredictable and dynamic nature of these applications make them good candidates for evaluating the effectiveness of our mapper.

We use a variety of machine learning, data mining, and data filtering applications for our evaluation—TunkRank (Twitter influence ranking), Alternating Least Squares (ALS) [28], Stochastic gradient descent (SGD) [12], Singular Value Decomposition (SVD) [11], Restricted Bolzman Machines (RBM) [10], Probabilistic Matrix Factorization (PMF) [21], Biased SGD [11], and Lossy SDG [11].

In addition to the above workloads, we also evaluate our implementation on MongoDB, a very widely used data management server. The load for MongoDB is generated by YCSB (Yahoo! Cloud Serving Benchmark).

5.2 Standalone Application Evaluation

Figure 2 shows that for most cases, standalone application performance on SAM-MPH is as good and sometimes better than a static optimum schedule. SAM-MPH and the other variants significantly outperform Linux in almost all cases. Linux generally distributes load across sockets and cores while SAM can detect and respond to data sharing and resource contention but not at the hyperthread level. For these standalone applications, SAM is already able to identify and isolate data sharing to achieve close to the best static schedule. SAM-M and SAM-MP thus add little extra benefit. SAM-MPH is able to identify all the bottlenecks exposed by SAM and outperforms it in 5 cases, demonstrating the importance of considering resource contention at the hyperthread level and of eliminating migrations due to transient application behavior.

With LuBench, PMF, RBM, SVD, and ALS, SAM-MPH performs better than SAM. SAM underperforms Linux in the case of LuBench. LuBench with 20 threads incurs non-trivial data sharing, which prompts SAM to colocate the threads if possible. However, when LuBench executes on two hardware contexts on the same physical core, single thread performance is significantly affected due to contention for pipeline resources. Since Linux by default spreads load out across sockets, it avoids the resource contention on hardware contexts. SAM-MPH identifies both data sharing and pipeline resource contention in LuBench and prioritizes pipeline resource contention as the bigger bottleneck in this case.

For applications PMF, SVD, RBM, and ALS, both Linux and SAM perform very close to the static optimum schedule with SAM being slightly faster. However, SAM-MPH outperforms the best static schedule by a significant margin for PMF and a slight margin for the rest. The best static schedule, as the name suggests, does not adapt to dynamic phases in the application. These four applications exhibit phases that share data and phases that contend for pipeline resources when colocated on hyperthreads. Since SAM-MPH is able to identify these different phases of computation and adapt accordingly to the bigger bottleneck, it is able to perform better than the best static schedule.

Figure 3 shows the intra- and inter-socket coherence activity for the standalone applications on the 40-CPU IvyBridge. In general, SAM is able to suppress inter-socket coherence activity slightly better than SAM-MPH. This slight reduction is attributed to SAM's decision making based on a single interval at a time. SAM-MPH relies on past history (consisting of several intervals) to detect application characteristics, resulting in higher hysteresis. The hysteresis has negligible impact on performance. For the five applications discussed

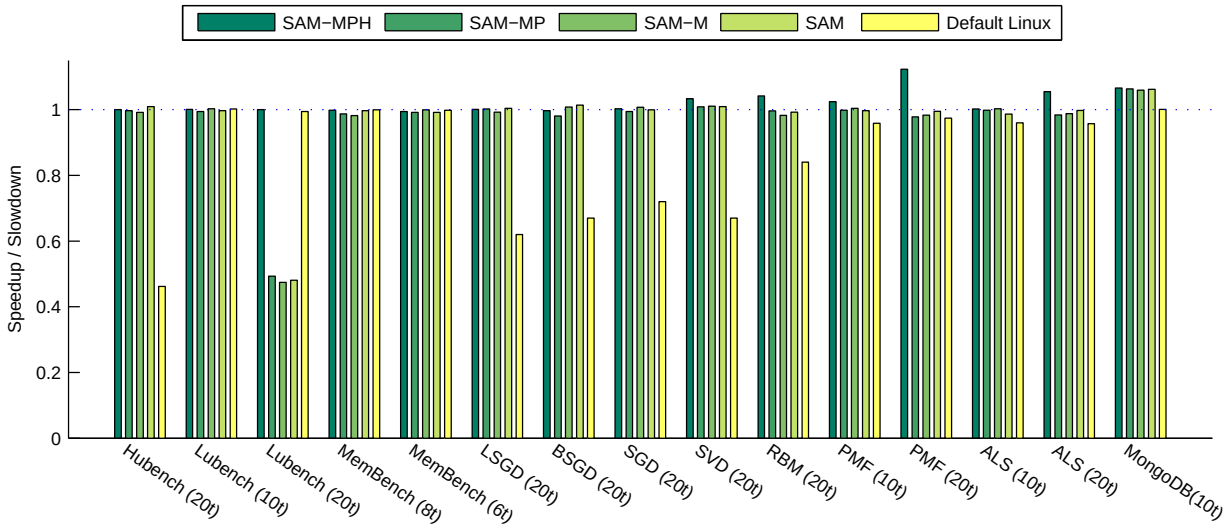

Figure 2: Performance of standalone applications using SAM-MPH, SAM-MP, SAM-M, SAM, and default Linux on the 40-CPU IvyBridge. The performance metric is the execution time speedup (the higher the better) compared to that of the best static task→CPU mapping determined through offline testing.

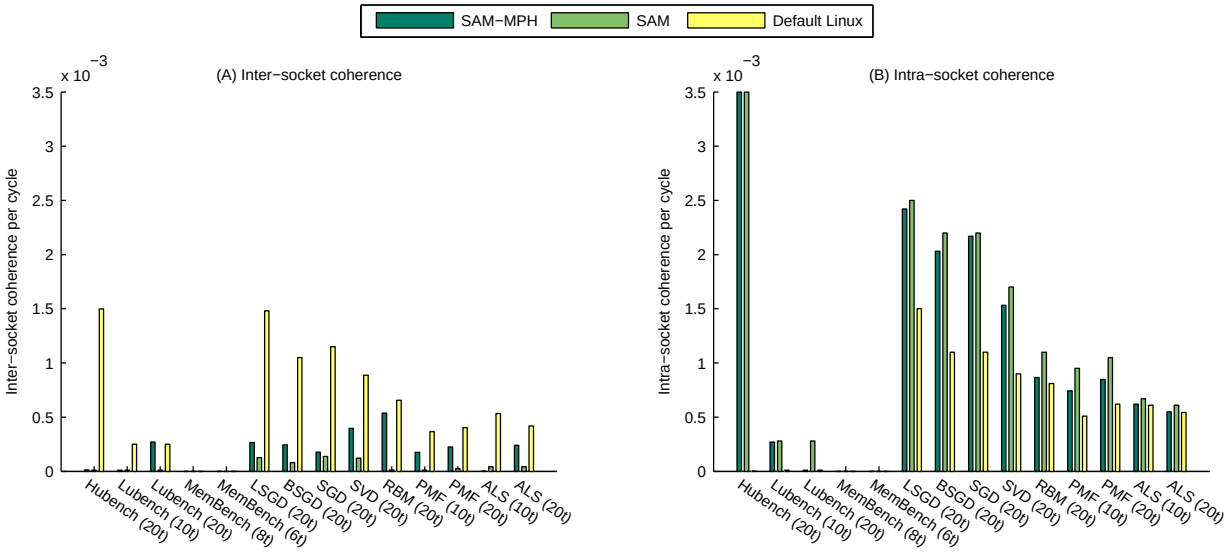

Figure 3: Intra- and Inter-socket traffic for standalone applications. Fig. (A): per-thread inter-socket coherence activity; Fig. (B): per-thread intra-socket coherence activity; All values are normalized to unhalted CPU cycles.

above, SAM-MPH has less intra-socket coherence activity than SAM, since it sometimes avoids colocating threads onto hyperthreads to reduce resource contention on the hyperthreads.

Table 1 outlines information about each application, with major and minor factors that influence its performance. It also shows the Stalls incurred per inter-socket coherence event (SPC). The SPC value reported here is averaged across all high communication phases of the application. We can see for SGD, LSGD, SVD, and BSGD, higher SPC translates to higher performance improvement on colocation.

RBM also exhibits high SPC values but its performance improvement doesn't directly correlate to SPC. This is attributed to the fact that RBM also has compute heavy phases which do not get significant speedup on colocation. Additionally, it would have to be placed on separate physical cores rather than on hyperthreads. Due to these factors, the overall speedup gained during the high coherence phase does not fully translate to very high performance gain.

PMF and ALS have moderate levels of coherence ac-

Application	SPC	Major Bottleneck	Minor Bottleneck
HuBench (20t)	745	DS	None
LuBench (10t)	367	IPC	DS
LuBench (20t)	367	IPC	DS
MemBench (10t)	-	Memory	None
MemBench (20t)	-	Memory	None
SGD (20t)	398	DS	None
BSGD (20t)	421	DS	None
LSGD (20t)	455	DS	None
RBM (20t)	403	DS	IPC
SVD (20t)	442	DS	IPC
PMF (10t)	-	None	IPC and DS
PMF (20t)	-	None	IPC and DS
ALS (10t)	-	None	IPC and DS
ALS (20t)	-	None	IPC and DS

Table 1: Application characteristics and SPC values. DS: Data Sharing: high coherence activity; IPC: Instructions Per Cycle: instruction-level parallelism with high CPU demand; Memory: Memory bound: high memory bandwidth demand.

tivity during which IPC is used for colocation decisions rather than SPC, since SPC cannot be obtained reliably at these levels as explained in Section 3. Hence the SPC value is not reported. When IPC is > 0.9, which is frequently the case for these applications, threads are preferentially placed on separate physical cores in the same socket, with placement across sockets preferred over placement on hyperthreads.

In addition to parallel data sharing workloads, we evaluate SAM-MPH on MongoDB, generating load with YCSB threads, both running on the same machine. For this workload, SAM-MPH and SAM perform very similarly. We observe an improvement of about 3.67% and 6.6% on the larger and smaller evaluation platforms respectively. The marginal improvement is a result of a small amount of data being shared by the threads of the application.

Our experiments with the PARSEC and SPECCPU benchmarks show very similar results to SAM and thus we do not discuss them further in this paper.

Figure 4 shows the results of SAM-MPH, SAM, and the default Linux scheduler on the 80-CPU Haswell. Overall, these results are very similar to results on the 40-CPU IvyBridge: SAM-MPH is able to match and sometimes exceed the performance of the best static schedule. The 80-CPU Haswell, with twice the number of sockets and cores as the 40-CPU IvyBridge, shows the performance gap between SAM-MPH and Linux widening further. SAM-MPH halves the execution time of applications compared to the default Linux scheduler. On average, for standalone workloads, SAM-MPH is 57% faster than Linux. It also matches or exceeds the performance of the best static schedule.

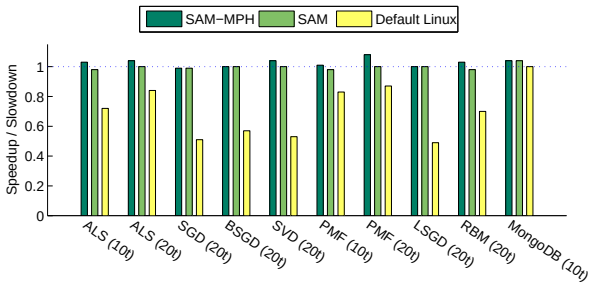

Figure 4: Performance of standalone applications using SAM-MPH, SAM, and default Linux on the 80-CPU Haswell. The performance metric is the execution time speedup (the higher the better) compared to that of the best static task→CPU mapping determined through offline testing.

5.3 Multiprogrammed Workload Evaluation

Applications in multiprogrammed workloads interfere with each other in different ways, depending on the characteristics of applications in the mix and the phase of their execution. This interference may result in slowdown of some or all of the applications.

Table 2 shows various application mixes that are used in the evaluation of SAM-MPH. The workload mixes cover a wide range of application characteristics. Individual applications can be affected due to contention for the processor pipeline, cache space contention, contention for memory bandwidth, communication due to data sharing, and non-uniform communication latencies. We expect SAM-MPH to be able to identify each of these bottlenecks and perform task to core mapping in such a way that would minimize the negative impact on performance due to resource contention and non-uniformity in communication.

Figure 5 shows the performance of SAM-MPH for the multiprogrammed workloads on the 40-CPU IvyBridge. Our performance metric for application mixes is the geometric mean of the individual application speedups, calculated for each application in a workload mix by comparing its runtime to that of its best standalone static runtime.

On average, SAM-MPH is about 27% faster than stock Linux and 9% faster than SAM. More importantly, applications managed by SAM-MPH show very little degradation in performance when compared with the best standalone static schedule. It must be noted that for many workload mixes, it is not possible to get numbers matching the standalone static schedule due to resource contention. The average speedup for SAM-MPH is 0.976, proving that applications seldom show signs of slowdown. The minimum speedup with SAM-MPH is 0.93.

SAM-MPH is able to improve SAM's performance

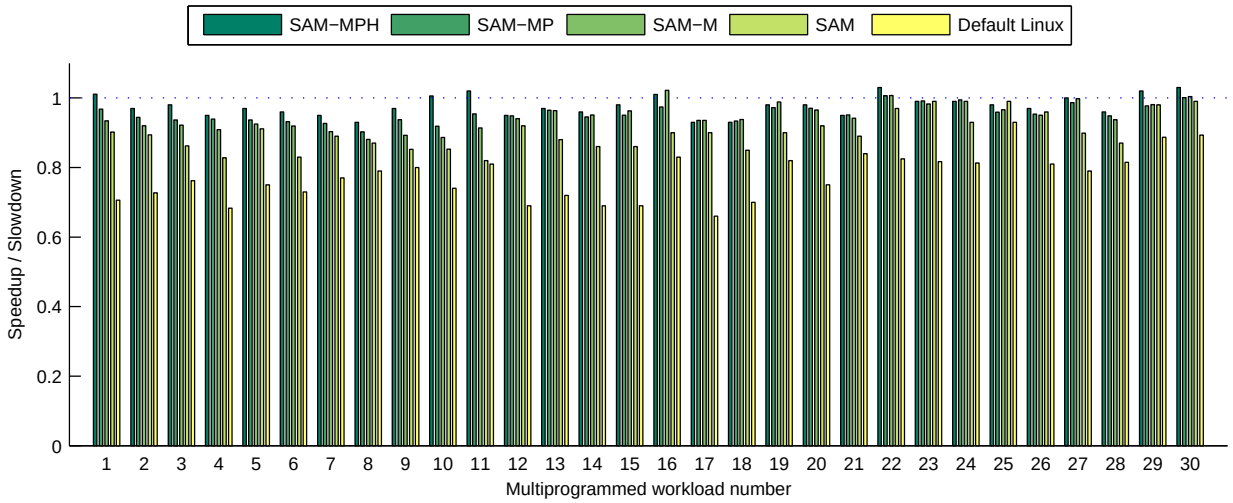

Figure 5: Performance of multiprogrammed workloads using SAM-MPH, SAM-MP, SAM-M, SAM, and default Linux on the 40-CPU IvyBridge. The performance metric is the geometric mean of the individual application speedup (higher is better) compared to execution time obtained for a standalone run using the best static task→CPU mapping determined through offline testing.

primarily due to 2 factors. First, SAM-MPH can prioritize applications that are more sensitive to bottlenecks base on latency tolerance, while SAM cannot distinguish tasks that are more sensitive to communication from others that are capable of absorbing the latency. Using application phase detection and accurate metrics that are deduced in these phases, SAM-MPH is able to understand the performance impact of communication due to data sharing.

Second, SAM-MPH identifies potential slowdown when putting two tasks on logical threads on the same physical core. Using this knowledge, it attempts to pair up applications such that they benefit from being placed on logical threads. If that is not possible, it attempts to schedule tasks so that they do not contend highly for the processor pipeline.

SAM-MPH's ability to identify data sharing and its impact on performance is the primary reason for the observed speedups in workloads 1–11. In each of these workloads, all applications exhibit data sharing. It is not possible to schedule all tasks such that tasks of an application remain inside the socket. SAM-MPH is able to prioritize applications that are more sensitive to the latency of communication due to data sharing.

In these workloads, ALS and PMF are the applications that are given the least priority and hence spread out across sockets. As discussed previously, these applications exhibit high ILP that is able to absorb the data communication latency. In addition to leveraging the tasks' ability to hide latency, SAM-MPH also identifies that ALS and PMF contend for the processor pipeline and avoids pairing their tasks together on the same physical core. Instead, SAM-MPH pairs each of their tasks with a task from the other applications to minimize resource contention. It is the combination of the these optimizations that yield consistent increase in performance of over 26% for these workloads.

SAM-M, being able to prioritize applications, can perform better than SAM. For workloads 1–11, SAM-M improves performance over Linux and SAM by 21% and 5% respectively. However, SAM-MP is faster than Linux and SAM by 25% and 8% respectively, demonstrating that SAM-MP's robustness adds value. SAM-MPH additionally mitigates contention on hyperthreads (due to ALS, RBM, and PMF), and is able to improve over SAM-MP to achieve performance closest to the standalone static schedule. SAM-MPH outperforms Linux and SAM by 30% and 13%.

For workloads 12–18, SAM-MPH identifies two applications with data sharing. The ideal decision for these workloads is to pin one application on each socket to localize all communication within a socket, which SAM-MPH and its variants correctly arrive at. SAM performs significantly slower since it does not have the notion of task groups and therefore is not able to separate the tasks. SAM relies on iteratively moving tasks onto the same socket since successful migrations will not cause additional inter-socket communication. Though this method works well in comparison with Linux, it does not achieve runtimes close to the optimal static runtime.

Workloads 19–21 contain applications with data sharing running simultaneously with other memory and CPU

Multiprog. workload #	Application mixes
1	12 ALS, 14 SGD, 14 LSGD
2	12 ALS, 14 SGD, 14 BSGD
3	12 ALS, 14 BSGD, 14 LSGD
4	12 ALS, 14 SVD, 14 BSGD
5	12 ALS, 14 SVD, 14 LSGD
6	12 ALS, 14 SVD, 14 SGD
7	12 ALS, 14 SVD, 14 RBM
8	12 ALS, 14 SGD, 14 RBM
9	12 PMF, 14 SGD, 14 RBM
10	12 PMF, 14 SGD, 14 BSGD
11	12 PMF, 14 SGD, 14 LSGD
12	20 SGD, 20 BSGD
13	20 SGD, 20 LSGD
14	20 SGD, 20 SVD
15	20 BSGD, 20 LSGD
16	20 LSGD, 20 ALS
17	20 LSGD, 20 SVD
18	20 BSGD, 20 SVD
19	6 SGD, 6 BSGD, 4 Mem, 4 CPU
20	6 BSGD, 6 LSGD, 4 Mem, 4 CPU
21	6 SGD, 6 LSGD, 4 Mem, 4 CPU
22	10 SGD, 10 BSGD
23	10 SGD, 10 LSGD
24	10 LSGD, 10 BSGD
25	10 LSGD, 10 ALS
26	10 SVD, 10 SGD
27	10 SVD, 10 BSGD
28	10 SVD, 10 LSGD
29	8 SVD, 8 LSGD
30	6 SVD, 6 LSGD

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

bound tasks. These cases demonstrate the capability to balance load and resource utilization alongside reducing latency due to communication. In these cases, SAM-MPH and its variants achieve close to the standalone performance. SAM underperforms due to its inability to form groups between the two data sharing applications. It does, however, balance load and resource utilization. Workloads 23–30 also exhibit data sharing characteristics but use only 20 out of the 40 hardware contexts that are available. In these cases, SAM attempts to colocate all threads onto the same socket. This eliminates all inter-socket coherence traffic but increases pressure and contention for the last-level cache. Since SAM-MPH and its variants identify task grouping, they separate the two groups on the two available sockets, further reducing contention and eliminating communication due to data sharing.

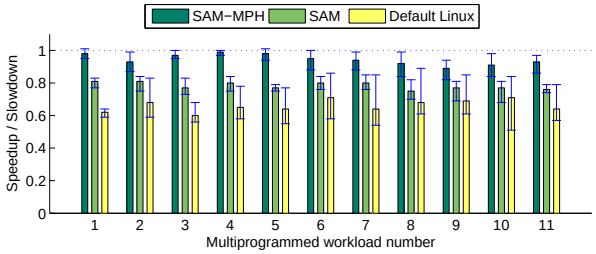

Figure 6: Performance of multiprogrammed workloads using SAM-MPH, SAM, and default Linux on the 80-CPU Haswell. The performance metric is the geometric mean of the individual application speedup (higher is better) compared to execution time obtained for a standalone run using the best static task→CPU mapping determined through offline testing. Whiskers represent the max-min speedup range for the individual applications within each workload.

Figure 6 shows the results obtained for the workload mixes listed in Table 3 on the four-socket 80-CPU Haswell. The workload mixes test SAM-MPH's ability to identify phases in applications that are most sensitive to data sharing. It also examines how SAM-MPH scales to a bigger machine with twice the number of processors and cores. We can see that SAM-MPH is able to achieve significantly better performance for all the workload mixes.

On average, we observe a 21% improvement over SAM and 43% improvement over stock Linux. While a reduction in performance compared to standalone execution is inevitable due to resource contention in multiprogrammed workloads, SAM-MPH is able to reduce the penalty. Performance improvement over Linux can be as high as 61% for our multiprogrammed workloads, while the minimum improvement was at 29% for workload 10.

Equally important, as the whisker plots in Figure 6 showing the minimum and maximum speedups for the individual applications in each workload show for the 4-socket 80-CPU Haswell machine, SAM-MPH reduces performance disparity (a measure of fairness) across applications in a workload in comparison to default Linux. The geometric mean of the minimum application speedup across all workload mixes is 0.889, 0.734, and 0.571 for SAM-MPH, SAM, and default Linux respectively. The corresponding values for the maximum speedup are 0.989, 0.822, and 0.795. On the 2-socket 40-CPU IvyBridge machine, the geometric mean of the minimum speedup is 0.953, 0.860, and 0.710, and that of the maximum is 1.0003, 0.932, and 0.839 respectively. Both SAM and SAM-MPH show a compressed spread.

5.4 Sensitivity Analysis

SAM-MPH relies on parameter thresholds to identify bottlenecks. In addition to the thresholds used in SAM,

Multiprog. workload #	Tasks per app	Application mixes
1	20	SGD, BSGD, SVD
2	20	SGD, BSGD, SVD, ALS
3	20	SGD, BSGD, SVD, LSGD
4	20	SGD, BSGD, RBM, LSGD
5	20	SGD, BSGD, RBM, SVD
6	20	SGD, BSGD, RBM, ALS
7	16	SGD, SVD, ALS, LSGD, BSGD
8	16	SGD, SVD, PMF, BGSD, LSGD
9	16	RBM, LSGD, SVD, PMF, BSGD
10	16	RBM, LSGD, SVD, PMF, ALS
11	16	SVD, SGD, RBM, BSGD, LSGD

Table 3: Multiprogrammed application mixes for experiments on the 80-CPU Haswell.

we use SPC and IPC values to prioritize task colocation based on latency tolerance and contention for pipeline resources. In this section, we look at sensitivity of SAM-MPH's behavior to these parameter thresholds. We increase/decrease the thresholds in steps of 5% to analyze the sensitivity of performance to these thresholds.

IPC is used to decide if tasks can be colocated on the same physical core. If the IPC threshold is too high, it is possible to map two compute intensive tasks on the same physical core, thereby slowing both down. A threshold increase of 20% (new IPC threshold of 1.08) can result in a performance reduction of about 7% on PMF. If the IPC threshold is too low, the mapper can miss a potential window to improve performance by colocating tasks that share data on the same physical core, thereby improving their performance and reducing contention by eliminating traffic on intra- and inter-socket interconnects. In our experiments, lowering the IPC threshold by 30% (new IPC threshold of 0.63) results in a loss in performance of 18% for the SGD application with 20 threads. The reduction in threshold created a false need to spread tasks across sockets in order to avoid colocating them on the same physical core, resulting in the slowdown.

SPC is used to prioritize applications when they are observed to have high coherence activity. Since SPC is approximated by attributing stalls due to all cache misses to coherence activity, SPC is reliable only at higher coherence activity levels. The coherence activity threshold used to identify when SPC is reliable is important to performance. We find a performance reduction of over 15% for mixed workload 8 when the threshold was increased by 30% (from 0.78×10^{-3} to 1.01×10^{-3}) due to missed opportunities. When the threshold is reduced by 50% (from 0.78×10^{-3} to 0.39×10^{-3}), we lose over 30% performance for workload 1 due to improper prioritization of applications.

Overall, we observe that while the value of the parameters is important to performance, SAM-MPH shows stable behavior over a reasonably broad range of values for these important parameters. In fact, we used the same thresholds, scaled for frequency, on the two platforms.

5.5 Overhead Assessment

SAM-MPH functionality can be divided into three distinct parts. The implementation complexity of each of these dictate the overall overhead of SAM-MPH. First, performance counters are read every 1 mSec. Second, every 100 mSecs, performance counter data is consolidated: application and socket-level bottlenecks are identified to be used to map tasks to cores. Finally, task mapping decisions are taken in order to improve performance. In order to measure the overhead of SAM-MPH, we perform a piecewise estimation since the implementation overhead is well within measurement error.

Reading performance counters are done at intervals of 1 mSec and consume 8.89 μSecs per call. This overhead is constant and does not vary with the number of processors/active tasks. Data consolidation, performed every 100 mSecs can consume a varying amount of time, primarily depending on the number of active applications. Worst case time consumption per SAM-MPH mapping call, including decision making and thread migration is 230 uSecs. Worst case behavior can be observed when each active task is its own application. The same overhead for when all cores are utilized but by only one application is about 14 uSecs. The additional overhead of over 200 uSecs is added by code that groups tasks into applications using address space information. The more distinct applications, the more the time spent on attributing tasks to applications. In most practical situations however, the number of applications will be significantly fewer than the number of cores. Overall, SAM-MPH adds a worst case overhead of just over 1%, which is far outweighed by its benefits.

SAM-MPH's data consolidation and decision making is implemented in a centralized fashion using a daemon process. On our machines, with 40 and 80 hardware threads, this implementation methodology works well. In the future, if SAM-MPH's overheads become a limitation, a distributed implementation may be warranted.

6 Related Work

Multicore resource contention and interference (particularly the shared last-level cache, off-chip bandwidth, and memory) has been well studied in previous work. Suh et al. [23] focus on minimizing cache misses using hardware counter-assisted marginal gain analysis. Page coloring [7, 26] has been used in the operating system memory allocator to effectively partition cache space without the need for specialized hardware features. Inter-task interference at the DRAM memory level has been

mitigated using parallelism-aware batch scheduling [18]. In an offline approach, Mars et al. [17] designed co-running microbenchmarks to control pressure on shared resources, and thereby predict the performance interference between colocated applications. While these techniques help manage resource contention, they do not address the impact of non-uniform topology on traffic due to data sharing.

ESTIMA [6] uses stall cycles to learn and predict application scalability on larger core counts using an offline approach. In contrast, our focus is on online multiprogrammed workload scheduling. Rao et al. [20] discuss using processor uncore pressure to minimize NUMA induced bottlenecks when scheduling virtual machines. While their approach of minimizing overall uncore pressure works to mitigate resource contention, it is not effective in eliminating resource pressure caused by data sharing.

Several efforts have also been made to automatically determine sharing among tasks. Tam et al. [24] utilized address sampling (available on Power processors) to identify task groups with strong data sharing. Tang et al. [25] relied on the number of accesses to "shared" cache lines to identify intra-application data sharing. Our past work [22] monitored and separated inter-CPU coherence activity from memory traffic to determine the benefits of consolidating tasks on the same socket versus distributing tasks across CPU sockets. Our work in this paper makes two new contributions. First, we identify latency tolerance in some workloads where inter-CPU coherence activity does not necessarily lead to CPU stalls and performance degradation. Second, we identify when the benefits of consolidating tasks on the same physical core due to data sharing outweigh performance loss due to contention for functional units and cache space.

Scheduling for simultaneous hardware multithreading, e.g., Intel's hyperthreads [13], has not been ignored in the past. Early work by Nakajima and Pallipadi [19] proposed two simple scheduling heuristics—1) task cache affinity to one hyperthread infers affinity to its sibling hyperthread; 2) scheduler should prefer a CPU whose sibling hyperthread is idle. Bulpin and Pratt [5] calibrated a blackbox linear model that predicts hyperthreading performance impact on a range of processor metrics. Their blackbox model provides no semantics on the hypothetical linear relationship and it is unclear how it applies broadly to other processors. Work in this paper monitors the cache coherence traffic, resulting stalls, and instruction retirement rates to understand the inter-CPU data sharing and potential latency tolerance, and thereby inform hyperthread colocation decisions.

The scalability of multicore and hardware multithreading has also been an emphasis in software system designs. For instance, Zhang et al. [27] presented user-space techniques (in the OpenMP runtime) to optimize inter-hyperthread data locality, instruction mix, and load balance. Multicore operating systems like Corey [4] and Multikernel [2] are designed to minimize cross-CPU sharing and synchronization for enhanced scalability. More recently, Callisto [9] is an OpenMP runtime system to handle synchronization and balance load on multicores. These efforts to improve software scalability are complementary to our CPU scheduling work—e.g., reduced data sharing traffic in some software tasks presents more flexibility to the scheduler that must consider resource contention, data sharing, and load balancing issues among all system and application tasks.

7 Conclusions

This paper presents new advances in resolving the tension between data sharing and resource contention in multicore task to core mapping. We make three specific contributions. First, we demonstrate the importance of identifying application latency tolerance, in addition to capturing data sharing traffic [22, 24, 25], in determining the true benefits of application and thread colocation. Second, we recognize that core-level sharing must pay attention to resource contention between hardware threads [5, 19] and show that a combination of IPC and coherence activity thresholds can inform the performance tradeoffs of core sharing. Third, we build an adaptive CPU socket and core sharing scheduler, called SAM-MPH, that uses history to avoid ineffective migrations due to oscillatory or transient behavior.

We perform experiments with a broad range of applications including SPEC CPU2000 [1], the PARSEC parallel benchmark suite [3], and GraphLab [16] / GraphChi [14] graph processing applications. Evaluation on a dual-socket, 40-CPU IvyBridge machine shows that SAM-MPH is 25% faster than Linux for standalone applications. On a larger 80-CPU Haswell machine with 4 sockets, SAM-MPH can halve the runtime of standalone workloads and can improve performance over Linux by up to 61% for multiprogrammed workloads. While SAM-MPH relies on thresholds to identify resource bottlenecks, our results show that performance is not sensitive to precise threshold values. Finally, SAM-MPH's runtime overhead in performance counter collection, analysis, and decision making is $\sim$1%, making it suitable for production use.

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. We also thank the anonymous USENIX ATC reviewers and our shepherd Andy Tucker for comments that helped improve this paper.

References

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

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

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

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[5] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conf.*, pages 399–402, Anaheim, CA, Apr. 2005.

[6] G. Chatzopoulos, A. Dragojević, and R. Guerraoui. Estima: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 27:1–27:11, New York, NY, USA, 2016. ACM.

[7] 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.

[8] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[9] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *9th EuroSys Conf.*, Amsterdam, Netherlands, Apr. 2014.

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

[11] 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.

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

[13] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, Apr. 2003.

[14] 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.

[15] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *USENIX Annual Technical Conf.*, Santa Clara, CA, July 2015.

[16] 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.

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

[18] 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.

[19] J. Nakajima and V. Pallipadi. Enhancements for hyper-threading technology in the operating system — seeking the optimal scheduling. In *Second Workshop on Industrial Experiences With Systems Software*, pages 25–38, Boston, MA, Dec. 2002.

[20] J. Rao, K. Wang, X. Zhou, and C. Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 306–317, Feb 2013.

[21] 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.

[22] S. Srikanthan, S. Dwarkadas, and K. Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In

USENIX Annual Technical Conf., Santa Clara, CA, July 2015.

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

[24] 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.

[25] 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.

[26] 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.

[27] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In *17th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 256–263, San Francisco, CA, Sept. 2004.

[28] 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.