Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors

David Tam   Reza Azimi   Michael Stumm

Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada M5S 3G4
{tamda, azimi, stumm}@eecg.toronto.edu

ABSTRACT

The major chip manufacturers have all introduced chip multiprocessing (CMP) and simultaneous multithreading (SMT) technology into their processing units. As a result, even low-end computing systems and game consoles have become shared memory multiprocessors with L1 and L2 cache sharing within a chip. Mid- and large-scale systems will have multiple processing chips and hence consist of an SMP-CMP-SMT configuration with non-uniform data sharing overheads. Current operating system schedulers are not aware of these new cache organizations, and as a result, distribute threads across processors in a way that causes many unnecessary, long-latency cross-chip cache accesses.

In this paper we describe the design and implementation of a scheme to schedule threads based on sharing patterns detected online using features of standard performance monitoring units (PMUs) available in today's processing units. The primary advantage of using the PMU infrastructure is that it is fine-grained (down to the cache line) and has relatively low overhead. We have implemented our scheme in Linux running on an *8-way* Power5 SMP-CMP-SMT multiprocessor. For commercial multithreaded server workloads (VolanoMark, SPECjbb, and RUBiS), we are able to demonstrate reductions in cross-chip cache accesses of up to 70%. These reductions lead to application-reported performance improvements of up to 7%.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*concurrency, scheduling, threads*; D.4.8 [**Operating Systems**]: Performance—*measurements, monitors*; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; I.5.3 [**Pattern Recognition**]: Clustering—*algorithms, similarity measures*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors*; C.5.5 [**Computer System Implementation**]: Servers; C.5.1 [**Computer System Implementation**]: Large and Medium ("Mainframe") Computers; B.3.2 [**Memory Structures**]: Design Styles—*cache memories, shared memory, virtual memory*; H.2.4 [**Database Management**]: Systems—*concurrency, parallel databases*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Algorithms, Management, Measurement, Performance, Design, Experimentation

Keywords

Affinity scheduling, cache behavior, cache locality, CMP, detecting sharing, hardware performance counters, hardware performance monitors, multithreading, performance monitoring unit, resource allocation, shared caches, sharing, simultaneous multithreading, single-chip multiprocessors, SMP, SMT, thread migration, thread placement, thread scheduling

1. INTRODUCTION

With diminishing potential improvements in clock speeds, processor chip manufacturers have turned towards increasing parallelism to obtain further performance gains. The major chip manufacturers have all introduced chip multiprocessing (CMP) and simultaneous multithreading (SMT) technology over the last several years for their laptop, desktop, and server processing units. As a result, even low cost computing systems and game consoles have become shared memory multiprocessors. Small- to medium-sized systems will contain a small number of processing chips (e.g., 1 to 4), while the number of cores and hardware threads in each core will likely increase over the next few years. For example, the Sun Niagara chip currently has 32 hardware contexts.

A key difference between the more traditional small-scale shared memory multiprocessors (SMPs) and these newer systems is that the latter have non-uniform data sharing overheads; i.e., the overhead of data sharing between two processing components differs depending on their physical location. For the processing units that reside on the same CPU core (i.e., hardware virtual contexts), communication
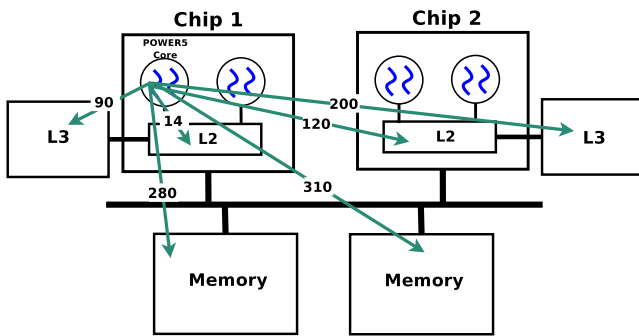
Figure 1: The IBM OpenPower 720 architecture. The numbers on the arrows indicate the access latency from a thread to different levels of the memory hierarchy. Any cross-chip sharing takes at least 120 CPU cycles.
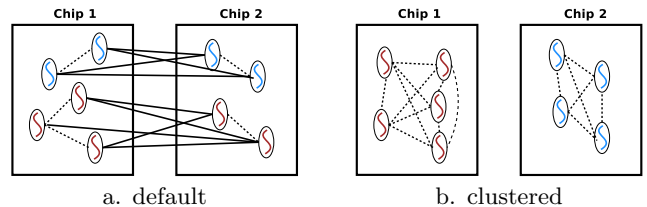

Figure 2: Default versus clustered scheduling. The solid lines represent high-latency cross-chip communications, the dashed lines are low-latency intra-chip communications (when sharing occurs within the on-chip L1 and L2 caches).

typically occurs through a shared L1 cache, with a latency of 1 to 2 cycles. For processing units that do not reside on the same CPU core but reside on the same chip, communication typically occurs through a shared L2 cache, with a latency of 10 to 20 cycles. Processing units that reside on separate chips communicate either by sharing memory or through a cache-coherence protocol both with an average latency of hundreds of cycles. As an example, consider the IBM OpenPower 720 latencies depicted in Figure 1.

Although operating systems have become increasingly cache-aware, their CPU schedulers today do not take the non-uniform sharing overheads into account. As a result, threads that heavily share data will not typically be co-located on the same chip. Figure 2 shows an example of a scenario where two clusters of threads are distributed across the processing units of two chips. The distribution is usually done as a result of some dynamic load-balancing scheme. If the volume of intra-cluster sharing is high, a default OS scheduling algorithm (as shown on the left) may result in many high-latency inter-chip communications (solid lines). If the OS can detect the thread sharing pattern and schedule the threads accordingly, then threads that communicate heavily could be scheduled to run on the same chip and, as a result, most of the communication (dashed lines) would take place in the form of on-chip L1 or L2 cache sharing.

A benefit of locating sharing threads onto the same chip is that they incidentally perform prefetching of shared regions for each other. That is, they help to obtain and maintain frequently used shared regions in the local cache.

Finally, non-communicating threads with high memory footprints may be better placed onto different chips, helping to reduce potential cache capacity problems.

Detecting sharing patterns of threads automatically has been a challenge. One approach used in the past for implementing software distributed shared memory (DSM) exploited page protection mechanisms to identify active sharing among threads [1]. This approach has two serious drawbacks: (i) the page-level granularity of detecting sharing is relatively coarse with a high degree of false sharing, and (ii) the over-

head of protecting pages results in high overhead with an attendant increase in page-table traversals and translation look-aside buffer (TLB) flushing operations.

In this paper we describe the design and implementation of a scheme to schedule threads based on detecting sharing patterns online with low overhead by using the data sampling features of the performance monitoring unit (PMU) available in today's processing units. The primary advantage of using the PMU infrastructure over page-level mechanisms is that the former is fine-grained (down to individual L2 cache lines) and has far lower overheads since most of the monitoring is offloaded to the hardware.

We have implemented this scheme in the Linux kernel running on an *8-way* IBM Power5 SMP-CMP-SMT multiprocessor. For commercial multithreaded server workloads (VolanoMark, SPECjbb, and RUBiS), we are able to demonstrate significant reductions in cross-chip cache accesses of up to 70%. These reductions lead to performance improvements of up to 7%.

The specific workloads we target in our experiments are multithreaded commercial server applications, such as databases, application servers, instant messaging servers, game servers, and mail servers. The programming model of these workloads is that there are multiple threads of execution, each handling a client request to completion. These threads exhibit some degree of memory sharing, and thus make use of the shared memory programming paradigm, as opposed to message passing. The scheme we propose automatically detects clustered sharing patterns among these threads and groups these threads accordingly onto the processing chips.

In theory, thread clustering may be done by the application programmer. However, it is fairly challenging for a programmer to determine the number of shared memory regions and intensity of sharing between them statically at development time. Another problem with manual, application programmer-written thread clustering is the extra effort of re-inventing the wheel for every application. Additional complexities may arise when application code is composed from multiple sources, such as shared libraries especially if the source code is not available. The dynamic nature of multiprogrammed computing environments is also difficult to account for during program development. Our scheme is capable of detecting sharing patterns that the application programmer may have been unaware of. In addition,

our scheme can handle phase changes and automatically re-cluster threads accordingly.

As a motivational example, our scheme can be applied to the Java platform without requiring modifications to the application or virtual machine run-time system. A Java application developer may write her multithreaded J2EE servlet as usual and the underlying OS would detect sharing among threads and cluster them accordingly.

2. RELATED WORK

The work most closely related to ours was by Bellosa and Steckermeier [4]. They first suggested using hardware performance counters to detect sharing among threads and to co-locate them onto the same processor. Due to the high costs of accessing performance counters at the time, ten years ago on a Convex SPP 1000, they did not obtain publishable results for their implementation. The larger scope of their research focused on performance scalability of NUMA multiprocessors, stressing the importance of using locality information in thread scheduling.

Weissman [23] proposed hardware performance counters to detect cache misses and reduce conflict and capacity misses. Their system required user-level code annotations to manually and explicitly identify shared regions among threads in order to deal with sharing misses. In our work, we demonstrate a technique that can automatically detect the shared regions.

Thread clustering algorithms were examined by Thekkath and Eggers [22]. This research dealt with finding the best way to group threads that share memory regions together onto the same processor so as to maximize cache sharing and reuse. Unfortunately, they were not able to achieve performance improvements for the scientific workloads they used in their experiments. The two main factors cited were (1) the global sharing of many data structures, and (2) the fact that data sharing in these hand-optimized parallel programs often occurred in a sequential manner, one thread after another. In contrast, our chosen workloads (1) exhibit non-global, clustered sharing patterns and (2) are not hand-optimized multithreaded programs but are written as client-server applications that exhibit unstructured, intimate sharing of data regions. Their work focused on the clustering algorithm, assuming that the shared-region information is known a priori, and was evaluated in a simulator. In contrast, our work focuses on the missing link and demonstrates a technique to detect these shared regions in an online, low-overhead manner on real hardware running a real operating system. Since our focus is not on the clustering algorithm itself, we used a relatively simple, low-overhead algorithm.

Sridharan et al. examined a technique to detect user-space lock sharing among multithreaded applications by annotating user-level synchronization libraries [19]. Using this information, threads sharing the same highly-contended lock are migrated onto the same processor. Our work adopts the same spirit but at a more general level that is applicable to any kind of memory region sharing. Locks could be considered a specific form of memory region sharing, where the region holds the lock mechanism. Consequently, our technique implicitly accounts for lock sharing among threads.

Bellosa proposed using TLB information to reduce cache misses across context switches and maximized cache reuse by identifying threads that share the same data regions [3]. Threads that share regions are scheduled sequentially, one after each other so as to maximize the chance of cache reuse. Koka and Lipasti had the same goals and provided further cache miss details [11]. The work of these two research groups was in the context of a uniprocessor system, in an attempt to maximize cache reuse of a single L2 cache, whereas our work targets multiple shared caches in a multiprocessor system, in an attempt to maximize cache reuse.

Philbin et al. attempted to increase cache sharing reuse of a single-threaded sequential program by performing automatic parallelization, creating fine-grained threads that made maximum cache reuse [16]. Larus and Parkes attempted to reduce cache misses between context switches by exploring a technique called cohort scheduling [12]. In the realm of databases, Harizopoulos and Ailamaki explored a method to transparently, without application source code modifications, increase instruction cache sharing re-use by performing more frequent but intelligently chosen thread context switches [9]. Selecting threads belonging to the same stage may improve instruction cache reuse. The general staged-event driven architecture is described and explored by Welsh et al. [24].

The remaining related work mostly concentrates on determining the best tasks to co-schedule in order to minimize capacity and conflict misses. Our work is targeted specifically at exploiting the shared aspect of shared caches in a multi-chip setting. Our work may be complementary to these past efforts in minimizing capacity and conflict misses of shared caches.

Many researchers have investigated minimizing cache conflict and capacity problems of shared L2 cache processors. Snavely and Tullsen did seminal work in the area of co-scheduling, demonstrating the problem of conventional scheduling and the potential performance benefits of symbiotic thread co-scheduling on a simulator platform [18]. With the arrival of Intel HT multiprocessor systems, Nakajima and Pallipadi explored the impact of co-scheduling on these real systems [14]. Parekh et al. made use of hardware performance counters that provided cache miss information to perform smart co-scheduling [15]. Others, such as McGregor et al. and El-Moursy et al., have found that on multiprocessors consisting of multiple SMT chips, cache interference alone was insufficient in determining the best co-schedules because SMT processors intimately share many micro-architectural resources in addition to the L1 and L2 caches [6, 13]. McGregor et al. found that per-thread memory bandwidth utilization, bus transaction rate, and processor stall cycle rate were important factors. El-Moursy et al. found that the number of ready instructions and the number of in-flight instructions were important. Suh et al. described the general approach of memory-aware scheduling, where jobs were selected to run based on cache space consumption [20, 21]. For example, a low cache consumption job was run in parallel with a high cache consumption job. Fedorova et al. examined the issue of operating system scheduler redesign and explored co-scheduling to reduce cache conflict and capacity misses based on a model of cache miss ratios [7, 8]. Bulpin

and Pratt also made use of hardware performance counters to derive a model for estimating symbiotic co-scheduling on an SMT processor [5]. Settle et al. proposed adding hardware activity vectors per cache line, creating a framework for exploring cache optimizations [17]. Their goal, within a single SMT chip context, was to minimize capacity and conflict misses.

3. PERFORMANCE MONITORING UNIT

Most modern microprocessors today have performance monitoring units (PMUs) with integrated hardware performance counters (HPCs) that can be used to monitor and analyze performance in real time. HPCs allow the counting of detailed micro-architectural events in the processor, such as branch mispredictions and cache misses. They can be programmed to interrupt the processor when a certain number of specified events occur. Moreover, PMUs make various registers available for inspection, such as addresses that cause cache misses or the corresponding offending instructions.

However, HPCs in practice are difficult to use because of (i) their limited number, (ii) the various constraints imposed on them, and (iii) the lack of documentation describing them in detail. For example, they do not provide enough counters to simultaneously monitor the many different types of events needed to form an overall understanding of performance. Moreover, HPCs primarily count low-level micro-architectural events from which it is difficult to extract high-level insight required for identifying causes of performance problems.

We use fine-grained HPC multiplexing that is introduced by previous work [2] to make a larger set of logical HPCs available. The PMU infrastructure is also able to speculatively associate CPU stalls to different causes [2]. Figure 3 shows an example of stall breakdown for the VolanoMark application. The average cycles-per-instruction (CPI) of the application is divided into *completion cycles* and *stall cycles*. A completion cycle is a cycle in which at least one instruction is retired. A stall cycle is a cycle in which no instruction is completed, which can be due to a variety of reasons. Stalls are broken down based on their causes. By using the hardware features, stalls that are due to data cache misses are further broken down according to the source from where the cache miss was satisfied. While it is possible to have a detailed breakdown of data cache misses according their sources, for the purpose of this paper, we are only interested in knowing whether the source was *local* or *remote*, where local means a cache on the same chip as the target thread, and remote means a cache on any other chip[1].

4. DESIGN

4.1 Overview of Thread Clustering Scheme

Our thread clustering approach consists of four phases.

1. **Monitoring Stall Breakdown:** Using HPCs, CPU stall cycles are broken down and charged to different microprocessor components to determine whether

---

[1]Although the L3 cache is often off-chip, we consider the L3 cache that is directly connected to a chip to be a local cache.
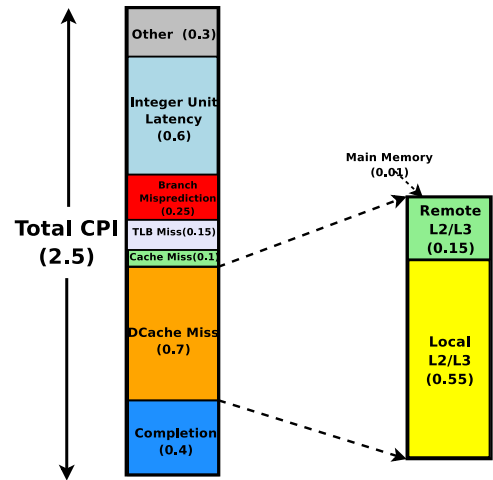

Figure 3: The stall breakdown for VolanoMark. The stalls due to data cache misses are further broken down according to the source from where the cache miss is eventually satisfied.

    cross-chip communication is performance limiting. If this is the case, then the second phase is entered.

2. **Detecting Sharing Patterns:** The sharing pattern between threads is tracked by using the data sampling features of the hardware PMU. For each thread, a summary vector, called *shMap*, is created that provides a signature of data regions accessed by the thread that resulted in cross-chip communication.

3. **Thread Clustering:** Once sufficient data samples are collected, the shMaps are analyzed. If threads have a high degree of data sharing then they will have similar shMaps and as a result, they will be placed into the same cluster.

4. **Thread Migration:** The OS scheduler attempts to migrate threads so that threads of the same cluster are as close together as possible.

We apply these phases in an iterative process. That is, after the thread migration phase, the system returns to the stall breakdown phase to monitor the effect of remote cache accesses on system performance and may re-cluster threads if there is still a substantial number of remote accesses. Additionally, application phase changes are automatically accounted for by this iterative process.

In the following subsections, we present the details of each phase.

4.2 Monitoring Stall Breakdown

Before starting to analyze thread sharing patterns, we determine whether there is a high degree of cross-chip communication with significant impact on application performance. Thread clustering will be activated only if the share of remote cache accesses in the stall breakdown is higher than a certain threshold. Otherwise, the system continues to monitor the stall breakdown. We used an activation threshold of

20% per billion cycles. That is, for every one billion cycles, if 20% of the cycles are spent accessing remote caches, then sharing detection phase is entered. Note that the overhead of monitoring stall breakdown is negligible since it is mostly done by the hardware PMU. As a result, we can afford to continuously monitor stall breakdown with no visible effect on system performance.

4.3 Detecting Sharing Patterns

In this phase, we monitor the addresses of the cache lines that are invalidated due to remote cache-coherence activities and construct a summary data structure for each thread, called *shMap*. Each shMap shows which data items each thread is fetching from caches on remote chips. We later compare the shMaps with each other to identify threads that are actively sharing data and cluster them accordingly.

4.3.1 Constructing shMaps

Each shMap is essentially a vector of 8-bit wide saturating counters. We believe that this size is adequate for our purposes because we are using sampling and are only looking for a rough approximation of sharing intensity. Each vector is given only 256 of these counters so as to limit overall space overhead. Each counter corresponds to a *region* in the virtual address space. Larger region sizes result in larger application address space coverage by the shMaps, but less precision and more sharing incidents will be falsely reported as a result. The largest region size with which no false-positives can occur is the size of an L2 cache line, which is the unit of data sharing for most cache-coherence protocols. Consequently, we used a region size of 128 bytes, which is the cache line size of our system.

With shMaps, we have effectively partitioned the application address space into regions of fixed size. Since 256 entries at 128 byte region granularity is inadequate to cover an entire virtual address space, we made use of hashing. We used a simple hash function to map these regions to corresponding entries in the shMap. A shMap entry is incremented only when the corresponding thread incurs a remote cache access on the region. Note that threads that share data but happen to be located on the same chip will not cause their shMaps to be updated as they do not incur any remote cache accesses.

We rely on hardware support to provide us with the addresses of remote cache accesses. While this feature is not directly available in most hardware PMUs, we use an indirect method to capture the address of remote cache accesses with reasonable accuracy. In Section 5.2.1 we provide details of how we implemented this method on the Power5 processor.

Constructing shMaps involves two challenges. First, to record and process every single remote cache access is prohibitively expensive, and secondly, with a small shMap the potential rate of hash collisions may become too high. We use sampling to deal with both challenges. To cope with the high volume of data we use *temporal sampling*, and to reduce the collision rate (and eliminate aliasing problems altogether) we use *spatial sampling*. Using temporal and spatial sampling of remote cache accesses instead of capturing them precisely is sufficient for our purposes because we only need an indication of the thread sharing pattern. If a data item is highly
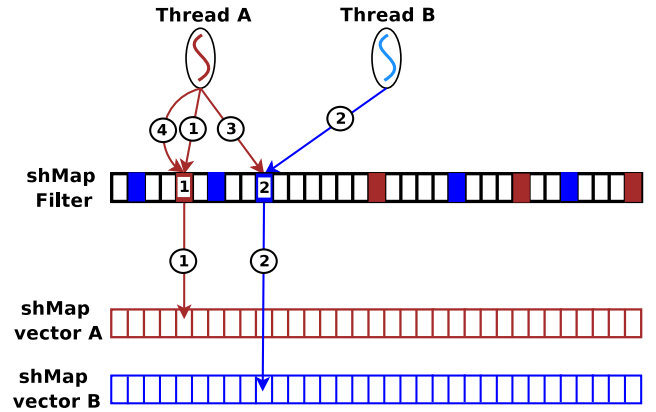

Figure 4: Constructing shMaps: each remote cache access by a thread will be indexed into the shMap filter. Only those remote cache accesses that pass the filter are marked in the corresponding entry in the shMap filter.

shared, i.e., remote cache accesses occur very frequently, it will likely be captured by the sampling.

Temporal Sampling. We record and process only one in N occurrences of remote cache access events. In order to avoid undesired repeated patterns, we constantly readjust N by a small random value. Moreover, the value of N is further adjusted by taking two factors into account: (i) the frequency of remote cache accesses (which is measured by the PMU), and (ii) the runtime overhead. A high rate of remote cache accesses allow us to increase N, since we will obtain a representative sample of addresses even with large values of N.

Spatial Sampling. Rather than monitor the entire virtual address space, we select a fairly small set of regions to be monitored for remote cache accesses. The regions are selected somewhat randomly, but there must be at least one remote cache access on a region to make it eligible to be selected. The hypothesis is that once a high level of sharing is detected on a subset of cache lines, it is a clear indication that the actual intensity of sharing is high enough to justify clustering.

We implement spatial sampling by using a filter to select remote cache access addresses after applying the hashing function. This *shMap filter* is essentially a vector of addresses with the same number of entries as a shMap. All threads of a process use the same shMap filter. A sampled remote cache access address is considered further (i.e., is allowed to pass the filter) only if its corresponding entry in the shMap filter has the same address value. Otherwise, the remote cache access is discarded and not used in the analysis. Each shMap filter entry is initialized (in an immutable fashion) by the first remote cache access that is mapped to the entry. Threads compete for entries in the shMap filter. This policy eliminates the problem of aliasing due to hash collisions. Figure 4 shows the function of the shMap filter.

In an unlikely pathological case, it is possible that some threads starve out others by grabbing the majority of the shMap filter entries, thus preventing remote cache accesses of other threads from being recorded. We place a limit on the number of entries allowed by a thread to partially address this problem. Additionally, we envision the thread clustering process to be iterative, thereby automatically handling insufficient thread clustering in subsequent iterations. That is, after detecting sharing among some threads and clustering them, if there is still a high rate of remote cache accesses, thread clustering is activated again, and the previously victimized threads will obtain another chance.

4.4 Thread Clustering

4.4.1 Similarity Metric

We define the similarity of two shMap vectors, corresponding to two threads, as their dot products:

$$similarity(T_1, T_2) = \sum_{i=0}^{N} T_1[i] * T_2[i]$$

where i is the ith entry of the vector $T_x[\ ]$

The rationale behind choosing this metric for similarity is two fold. First, it automatically takes into account only those entries where both vectors have non-zero values. Note that T_1 and T_2 have non-zero values in the same location only if they have had remote cache accesses on the same cache line (i.e., the cache line is being shared actively). We consider very small values (e.g., less than 3) to be zero as they may be incidental or due to cold sharing and may not reflect a real sharing pattern.

Second, it takes into account the intensity of sharing by multiplying the number of remote cache accesses each of the participating threads incurred on the target cache line. That is, if two vectors have a large number of remote cache accesses on a small number of cache lines, the similarity value will be large, correctly identifying that the two threads are actively sharing data. Other similarity metrics could be used, but we found this metric to work quite well for the purpose of thread clustering.

In our experiments, we used a similarity threshold value of approximately 40000. For two candidate vectors, this similarity threshold could be achieved under various simple scenarios, such as: (1) a single corresponding entry in each vector has values greater than 200; or (2) two corresponding entries in each vector have values greater than 145.

4.4.2 Forming Clusters

One way to cluster threads based on shMap vectors is to use standard machine learning algorithms, such as hierarchical clustering or K-means [10]. Unfortunately, such algorithms are too computationally expensive to be used online in systems with potentially hundreds or thousands of active threads, or they require the maximum number of clusters to be known in advance, which is not realistic in our environment.

To avoid high overhead, we use a simple heuristic for clustering threads based on two assumptions that are simplifying but fairly realistic. First, we assume data is naturally partitioned according to the application logic and threads that work on two separate partitions do not share much except for data that is globally (i.e., process-wide) shared among all threads. In order to remove the effects of globally shared data on clustering, we build a histogram for shMap vectors in which each entry shows how many shMap vectors have a non-zero value in that particular entry. We consider a cache line to be globally shared if more than half of the total number of threads have incurred a remote cache access on it. We ignore information on globally shared cache line when composing clusters.

The second assumption is that when a subset of threads share data, the sharing is reasonably symmetric. That is it is likely that *all* of them incur remote cache accesses on similar cache lines, no matter how they are partitioned.

As a result, the clustering algorithm can be simplified as follows. Based on the first assumption, if the similarity between shMap vectors is greater than a certain threshold, we consider them to belong to the same cluster. Also, according to the second assumption, any shMap vector can be considered as a cluster representative since all elements of a cluster share common data equally strongly.

The clustering algorithm scans through all threads in one pass and compares the similarity of each thread with the representatives of known clusters. If a thread is not similar to any of the known cluster representatives, a new cluster will be created, and the thread that is currently being examined will be designated as the representative of the newly created cluster. The set of known clusters is empty at the beginning. The computational complexity of this algorithm is $O(T * c)$ where T is the number of threads that are suffering from remote cache accesses, and c is the total number of clusters, which is usually much smaller than T.

4.5 Thread Migration

Once thread clusters are formed, each cluster is assigned to a chip with the global goal of maintaining load-balance. That is, in the end, there should be an equal number of threads on each chip. Our cluster-to-chip assignment strategy is as follows. First, we sort the clusters from the largest size to the smallest size so that we can easily select the next largest available cluster to migrate. Second, we assign the current largest cluster to the chip with the lowest number of threads. If such an assignment causes an imbalance among chips, then we instead evenly assign the cluster's threads to each chip. This strategy attempts to maintain good load-balancing at every step, and if the current cluster is problematic, then it is neutralized by distributing its threads evenly among the chips. The above steps are repeated for every thread cluster. Finally, the remaining non-clustered threads are placed onto the chips to balance out any remaining differences. We recognize that this is a best-effort, practical, online strategy that provides no guarantee of optimality.

Load balance within each chip is addressed by uniformly and randomly assigning threads to the cores and the different hardware contexts on the core. To minimize cache

Table 1: IBM OpenPower 720 specification.

Item	Specification
# of Chips	2
# of Cores	2 per chip
CPU Cores	IBM Power5, 1.5GHz, 2-way SMT
L1 DCache	64KB, 4-way associative, per core
L1 ICache	64KB, 4-way associative, per core
L2 Cache	2MB, 10-way associative, per chip
L3 Cache	36MB,12-way associative, per chip, off-chip
RAM	8GB (2 banks x 4GB)

capacity and conflict problems within a single chip, a variety of intra-chip scheduling techniques described in Section 2 could be applied, such as the CMT-aware scheduler (Chip MultiThreading) of Fedorova et al. [7] and the SMT-aware scheduler of Bulpin and Pratt [5].

In balancing threads among chips, cores, and hardware contexts, we make the simplifying assumption that threads are fairly homogeneous in their usage of assigned scheduling quantum. Although we have not done so in this paper, we plan to enable default Linux load-balancing within each chip, as opposed to loading-balancing across chips, so that balancing can take place among the cores and hardware contexts within a chip. This feature would help in reducing the severity of any subsequent load imbalance.

5. EXPERIMENTAL SETUP
5.1 Platform
The multiprocessor used in our experiments is an IBM Open-Power 720 computer. It is an *8-way* Power5 machine consisting of a 2x2x2 SMPxCMPxSMT configuration, as shown in Figure 1. Table 1 describes the hardware specifications.

While our evaluation platform is fairly modest, we believe it is suitable to explore much of the sharing behavior we discussed in this paper. However, for fully realizing the potential of our approach, we plan to evaluate it on machines with a larger number of processors.

We used Linux 2.6.15 as the operating system. Linux was modified in order to add the features needed for hardware performance monitoring, including the stall breakdown and remote cache access address sampling. We also changed the CPU scheduling code to migrate threads according to the thread clustering scheme proposed in this paper.

5.2 Platform Specific Implementation Issues
5.2.1 Capturing Remote Cache Accesses on Power5
The Power5 PMU provides a mechanism called *continuous sampling* that captures the address of the last L1 data cache miss or TLB miss (or both) in a continuous fashion regardless of the instruction that caused the data cache miss or TLB miss. The sampled address is recorded in a register which is updated on the next data cache miss or TLB miss. It is not possible to directly determine whether the sampled local L1 data cache miss was satisfied by a remote or local cache access. As a result, by just taking data cache misses regardless of their source, an unacceptable level of noise is added to the monitoring scheme. Fortunately, we have been

able to develop a technique to remove much of this noise from our samples as follows.

In the Power5 processor, it is possible to *count* the occurrences of local L1 data cache misses that are satisfied by a remote L2 or remote L3 cache access. As a result, it is possible to set the PMU overflow exception to be raised when a certain number of remote cache accesses has been reached. Once an overflow exception is raised, the "last" local data cache miss is very likely to have required a remote cache access that caused one of the HPCs to overflow. Therefore, by reading the sample data register only when the remote cache access counter overflows, we ensure that most of the samples read are actually remote cache accesses. Our experiments with various microbenchmarks verify the effectiveness of this method as almost all of the local L1 data cache misses recorded in our trace are indeed satisfied by remote cache accesses.

5.3 Workloads
For our experiments, we used a synthetic microbenchmark and three commercial server workloads, VolanoMark which is an Internet chat server, SPECjbb2000, which is a Java-based application server workload, and RUBiS, which is an online transaction processing (OLTP) database workload. For VolanoMark and SPECjbb, we used the IBM J2SE 5.0 Java virtual machine (JVM). For RUBiS, we used MySQL 5.0.22 as our database server. These server applications are written in a multithreaded, client-server programming style, where there is a thread to handle each client connection for the life time of the connection. We present details of each benchmark below.

5.3.1 Synthetic Microbenchmark
The synthetic microbenchmark is a simple multithreaded program in which each worker thread reads and modifies a scoreboard. Each scoreboard is shared by several threads, and there are several scoreboards. Each thread has a private chunk of data to work on which is fairly large so that accessing it often causes data cache misses. This is to verify that our technique is able to distinguish remote cache accesses that are caused by accessing the shared scoreboards from local cache accesses that are caused by accessing the private data. All scoreboards are accessed by a fixed number of threads. A clustering algorithm is supposed to cluster threads that share a scoreboard and consider them as the unit of thread migration.

5.3.2 VolanoMark
VolanoMark is an instant messaging chat server workload. It consists of a Java-based chat server and a Java-based client driver. The number of rooms, number connections per room, and client think times are configurable parameters. This server is written using the traditional, multithreaded, client-server programming model, where each connection is handled completely by a designated thread for the life-time of the connection. In actuality, VolanoMark uses two designated threads per connection. Given the nature of the computational task, threads belonging to the same room should experience more intense data sharing that threads belonging to different rooms.

In our experiments, we used 2 rooms with 8 clients per room and 0 think time as our test case. In this setting, the hand-optimized placement of threads would be for the threads of each room to be located on separate chips. In the worst case scenario, the threads are placed randomly or in a round-robin fashion.

5.3.3 SPECjbb2000

SPECjbb2000 is a self-contained Java-based benchmark that consists of multiple threads accessing designated *warehouses*. Each warehouse is approximately 25 MB in size and stored internally as a B-tree variant. Each thread accesses a fixed warehouse for the life-time of the experiment. Given the nature of the computational task, threads belonging to the same warehouse should experience more intense data sharing than threads belonging to different warehouses.

In our experiments, we modified the default configuration of SPECjbb so that multiple threads can access a warehouse. Thus, in our configuration, we ran the experiments using 2 warehouses and 8 threads per warehouse.

5.3.4 RUBiS

RUBiS is an OLTP server workload that represents an online auction site workload in a multi-tiered environment. The client driver is a Java-based web client that accesses an on-line auction web server. The front-end web server uses PHP to connect to a back-end database. In our experiments, we ran MySQL 5.0.22 as our back-end database. We focus on the performance of the database server. We made a minor modification to the PHP client module so that it uses persistent connections to the database, allowing for multiple MySQL requests to be made within a connection. While this modification improves performance by reducing the rate of TCP/IP connection creation and corresponding thread creation on the database server, it also enables our algorithm to monitor the sharing pattern of individual threads over the long term.

In our workload configuration, we used two separate *database instances* within a single MySQL process. We used 16 clients per database instance with no client think time. This configuration may represent, for instance, two separate auction sites run by a single large media company. We expect that threads belonging to the same database instance will experience more intense sharing with each other than with other threads in the MySQL process.

5.4 Thread Placement

We evaluated four thread placement strategies: default Linux, round-robin, hand-optimized, and automatic thread clustering. The default Linux thread placement strategy attempts to find the least loaded processor in which to place the thread. In addition, Linux performs two types of dynamic load-balancing: *reactive* and *pro-active*. In reactive load-balancing, once a processor becomes idle, a thread from a remote processor is found and migrated to the idle processor. Pro-active load-balancing attempts to balance the CPU time each threads gets by automatically balancing the length of the processor run queues. The default Linux scheduler does not take data sharing into account when migrating and scheduling the threads.

For round-robin scheduling, we modified Linux to disable dynamic load balancing. Threads of our targeted workload are placed in a round-robin fashion among processors. This thread placement strategy is unaware of data sharing patterns among threads. The round-robin scheduling is implemented in order to be able to exhibit worst case scenarios where sharing threads are scattered onto different chips.

With hand-optimized scheduling, threads are placed by considering natural data partitioning according to the application logic[2]. For VolanoMark, threads belonging to one room are placed on one chip while threads belonging to the other room are placed on the other chip. Within each chip, threads of the room are placed in a round-robin fashion to achieve load-balance within the chip. Similarly for SPECjbb, threads of one warehouse are placed onto the same chip. The same pattern applies for RUBiS: the threads of one database instance are placed onto one chip while threads of the second database instance are placed onto the other chip. For hand-optimized scheduling, the Linux scheduler is modified to disable both reactive and pro-active load-balancing.

6. RESULTS
6.1 Thread Clustering

Figure 5 shows a visual representation of shMap vectors and the way they are clustered for the four applications. Each application is represented by a gray scale picture in which each row represents a shMap vector of a thread. The darker a point is, the more often remote cache accesses are sampled for the corresponding shMap entry. Therefore, a continuous vertical dark line represents thread sharing among correctly clustered threads. To simplify the picture, the globally (process-wide) shared data have been removed[3]. From Figure 5 it is clear that the shMaps are effective in detecting sharing and clustering threads for three applications out of four (microbenchmark, SPECjbb, and RUBiS). In all three cases the automatically detected clusters conforming to a manual clustering that can be done with specific knowledge about the application logic (i.e., a cluster for each scoreboard for the microbenchmark, for each warehouse in SPECjbb, and for each database instance in MySQL). JVM garbage collector threads in SPECjbb and VolanoMark did not affect cluster formation since they are run infrequently and do not have the opportunity to exhibit much sharing.

For VolanoMark however, the detected clusters do not conform with the logical data partitioning of the application logic (i.e., one partition per chat room). However, as we will show later, the automatic clustering approach still improves performance by co-locating threads that share data.

6.2 Performance Results

Figure 6 shows the impact of the different thread scheduling schemes on processor stalls caused by accessing high-latency remote caches. In general, it is clear that it is possible to

---

[2]We do not claim that the hand-optimized thread placements are the optimal placements, but are merely significantly improved placements based on application domain knowledge.

[3]For illustration purposes, SPECjbb was run with 4 warehouses. In subsequent experiments, 2 warehouses are used.

a. Microbenchmark

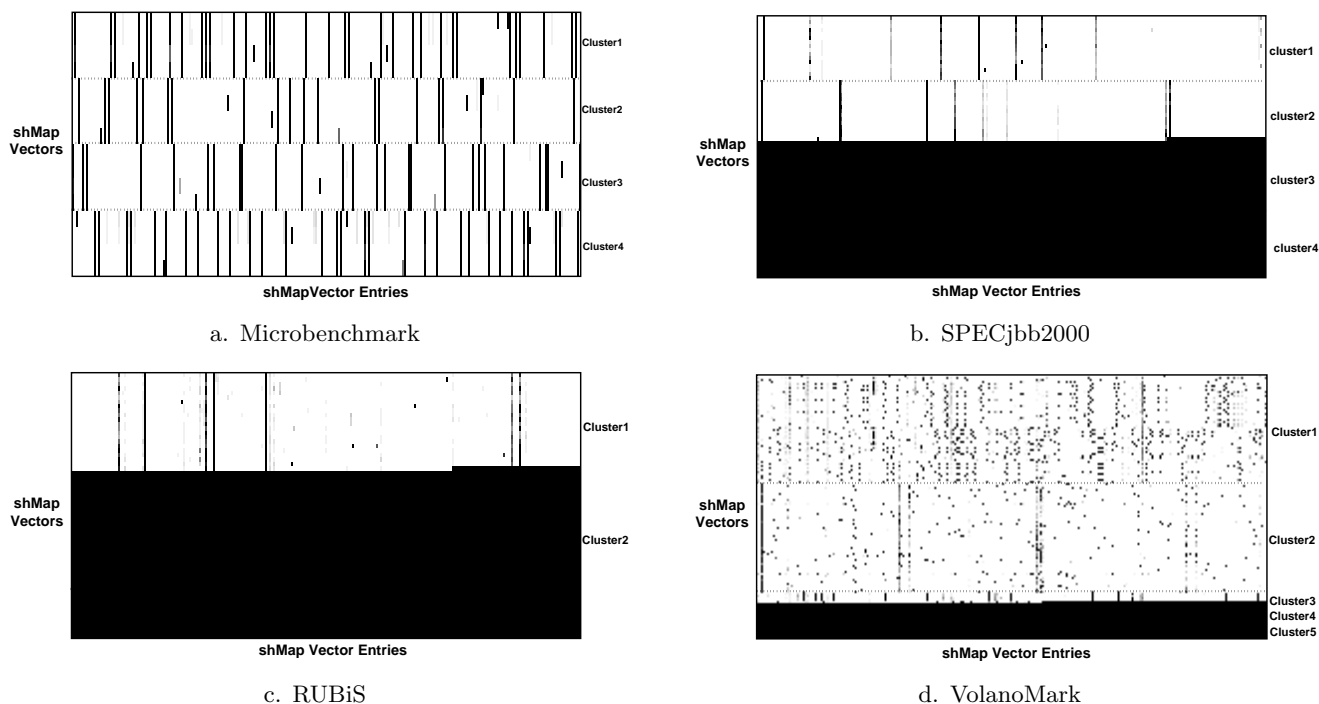

b. SPECjbb2000

c. RUBiS

d. VolanoMark

Figure 5: Visual representation of shMap vectors. Each labeled cluster consists of several rows of shMap vectors. Each row represents a thread's shMap vector. Each shMap entry is represented with a gray scale point. More frequently accessed entries appear darker.
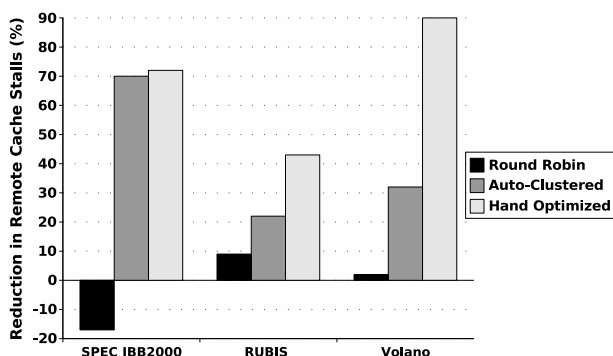

Figure 6: The impact of the scheduling schemes on reducing stalls caused by remote cache accesses. The baseline is Linux default scheduling.

Figure 7: The performance impact of scheduling schemes on application performance. The baseline is Linux default scheduling.

remove a significant portion of remote access stalls either by hand-optimizing the thread placement, or through automatic clustering. For SPECjbb, the automatic clustering approach performs nearly as good as the hand-optimized method. For the other two applications there is still further room for improvement.

Figure 7 shows the impact of the different thread scheduling schemes on application performance. Again, both the hand-optimized and the automatic clustering schemes manage to improve performance by a reasonable amount, but there is still room for improving the automatic clustering scheme. The magnitudes of performance gain appear rea-
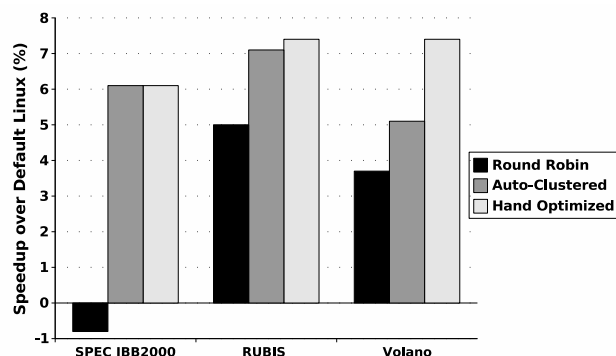
sonable because they approximately match the reduction in processor stalls due to remote cache accesses. For example, in Figure 3, 6% of stalls in VolanoMark were due to remote cache accesses and thread clustering was able to improve performance by 5% by removing some of these stalls.

6.3 Runtime Overhead & Temporal Sampling Sensitivity

The average runtime overhead for identifying stall break-down is negligible. Therefore, the main runtime overhead of the system is due to detecting the sharing patterns. Figure 8 shows the runtime overhead of this phase as a func-
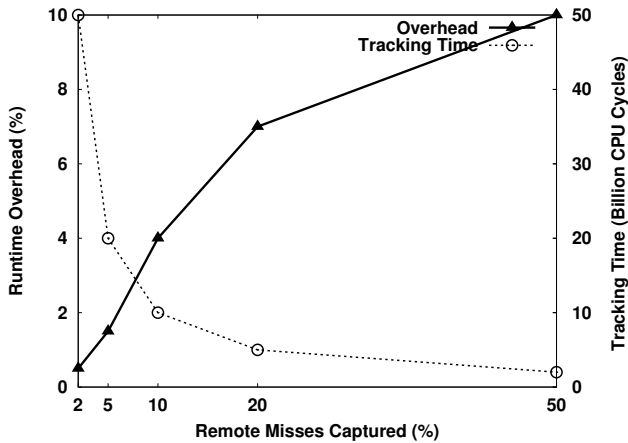
Figure 8: Runtime overhead of the sharing detection phase and the time (in billion CPU cycles) that is required to collect a million remote cache access samples. The x-axis is the temporal sampling rate, i.e., the percentage of the remote cache accesses that are sampled.

tion of temporal sampling rate in terms of the percentage of the remote cache accesses that are actually examined for SPECjbb. As a higher percentage of the remote cache accesses are captured, the overhead increases. However, the length of this phase is fairly limited and only goes until we collect a sufficient number of samples to be able to cluster the threads. In our experiments, we have found we need roughly a million samples to accurately cluster the threads. Therefore, on the right y-axis of Figure 8 we show how long (in billion CPU cycles) we need to stay in the detection phase to collect a million samples. Hence, the higher the sampling rate, the higher the run-time overhead will be, but the shorter the detection phase will last. According to Figure 8 it seems a sampling rate of 10 (capturing one in every 10 remote cache accesses) is a good balance point in this trade-off.

6.4 Spatial Sampling Sensitivity
Although not shown, we have tried varying the number of entries in the shMap vectors for our workloads and found the cluster identification to be largely invariant. For example, we ran experiments using shMap sizes of 128 entries and 512 entries. The impact of using 128 entries as opposed to 256 entries on SPECjbb can be roughly visualized by covering the left half of the Figure 5b. Clustering would still identify the same groups of threads as sharing.

7. DISCUSSION
7.1 Local Cache Contention
Clustering too many threads onto the same chip could create local cache contention problems. The local caches may not have sufficient capacity to contain the aggregate working set of the threads. In addition, because these local caches are not fully associative but are set-associative, cache conflict problems may be magnified. Fortunately in our system, local L2 cache contention is mitigated by a large local L3 cache (36 MB). However, local cache contention was not significant in our workloads.

7.2 Migration Costs
Thread migration incurs the costs of cache context reloading into the local caches and TLB reloading. Compared to typical process migration that is performed by operating systems, such as default Linux, thread migration has lower costs since threads in a single application address space typically exhibit more cache context and TLB sharing. Any reloading costs are expected to be amortized over the long thread execution time at the new location, where threads enjoy the benefits of reduced remote caches accesses. Our results have shown that these benefits outweigh the costs.

7.3 PMU Requirements
Ideally, we would like the ability to specifically configure the PMU to continuously record the data address of remote cache accesses. Unfortunately, this direct capability is not available on the Power5 processor and so it was composed using basic PMU capabilities as described in Section 5.2.1. Currently, as far as we are aware, no other commercially available processors provide the direct capability or suitably composable basic capabilities.

It is interesting to note that although hardware designers initially added PMU functionality primarily to collect information for their own purposes, namely for designing the next generation of processor architectures, PMUs have become surprisingly useful for purposes other than for which they were envisioned. Consequently, they are now adding more and more capabilities requested by software designers. We hope that this paper provides compelling evidence of the usefulness of PMU sharing detection capabilities so that more processor manufacturers would seriously consider adding them to future processors.

7.4 Important Hardware Properties
Our thread clustering approach is viable because there exists a large disparity between local and remote cache latencies. On larger multiprocessor systems, where this disparity is even greater, we expect higher performance gains. In actuality, running on a *32-way* Power5 multiprocessor consisting of 8 chips, we saw a greater performance impact from thread clustering. Our preliminary results indicate a 14% throughput improvement in SPECjbb when comparing handcrafted placement to the default Linux configuration. We are currently working on obtaining the throughput results of automatic thread clustering.

8. CONCLUDING REMARKS
We have described the design and implementation of a scheme to schedule threads based on sharing patterns detected online using features of standard performance monitoring units (PMUs) available in modern processing units. Experimental results indicate that our scheme is reasonably effective: running commercial multithreaded server Linux workloads on an *8-way* Power5 SMP-CMP-SMT multiprocessor, our scheme was able to reduce remote cache access stalls by up to 70% and improve application performance by up to 7%. Our work in this area is admittedly still at a relatively early stage. Although we have briefly examined the impact of temporal and spatial sampling, we have not yet examined the sensitivity of other parameters, such as the similarity metric and the clustering algorithm. Comparing

the detection accuracy of our light-weight clustering algorithm against full-blown clustering algorithms is a subject of future work. Moreover, the platform used for experimentation is modest; we plan to run experiments on larger-scale systems. Nevertheless, we find the results obtained so far to be promising and we are currently considering additional workloads.

This work, we believe, represents the first time hardware PMUs have been used to detect sharing patterns in a fairly successful fashion. More specifically, we have found our method of identifying sharing patterns using shMap signatures to be surprisingly effective considering (i) their relatively small size of only 256 entries, and (ii) the liberal application of sampling along several dimensions (temporal and spatial).

Finally, we believe that it would be straight-forward to extend our scheme to provide scheduling support for traditional NUMA multiprocessors. For this work, we filtered out all PMU cache miss events except for misses that are satisfied by remote L2 and remote L3 cache accesses. This could easily be changed to filter out all cache misses that are satisfied from remote L3 caches and remote memory.

9. ACKNOWLEDGEMENTS

10. REFERENCES

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[2] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Intl. Conf. on Supercomputing*, 2005.

[3] F. Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. In *Symp. on Operating Systems Principles - Work in Progress Session*, 1997.

[4] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *J. of Parallel and Distributed Computing*, 37(1):113–121, Aug 1996.

[5] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Usenix Annual Technical Conf.*, 2005.

[6] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Intl. Parallel and Distributed Processing Symp.*, 2006.

[7] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Usenix Annual Technical Conf.*, 2005.

[8] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *SIGOPS European Workshop*, 2004.

[9] S. Harizopoulos and A. Ailamaki. STEPS towards cache-resident transaction processing. In *Conf. on Very Large Data Bases*, 2004.

[10] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[11] P. Koka and M. H. Lipasti. Opportunities for cache friendly process scheduling. In *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[12] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *Usenix Annual Technical Conf.*, 2002.

[13] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Intl. Parallel and Distributed Processing Symp.*, 2005.

[14] J. Nakajima and V. Pallipadi. Enhancements for Hyper-Threading technology in the operating system – seeking the optimal micro-architectural scheduling. In *Workshop on Industrial Experiences with Systems Software*, 2002.

[15] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. Technical report, Dept. of Computer Science & Engineering, Univ. of Washington, 2000.

[16] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.

[17] A. Settle, J. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced SMT job scheduling. In *Symp. on Parallel Architectures and Compilation Techniques*, 2004.

[18] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[19] S. Sridharan, B. Keck, R. Murphy, S. Chandra, and P. Kogge. Thread migration to improve synchronization performance. In *Workshop on Operating System Interference in High Performance Applications*, 2006.

[20] E. G. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 116–132, Cambridge, MA, Jun 16 2001. Springer-Verlag.

[21] E. G. Suh, L. Rudolph, and S. Devadas. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Symp. on*

High-Performance Computer Architecture, 2002.

[22] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Intl. Symp. on Computer Architecture*, 1994.

[23] B. Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[24] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symp. on Operating Systems Principles*, 2001.