SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors

Lei Jin and Sangyeun Cho
Department of Computer Science
University of Pittsburgh
Pittsburgh, USA
Email: {jinlei,cho}@cs.pitt.edu

***Abstract*—This paper proposes a new software-oriented approach for managing the distributed shared L2 caches of a chip multiprocessor (CMP) for latency-oriented multithreaded applications. The conventional shared cache scheme loses performance due to the blind distribution of data predominantly accessed by a single thread. SOS, our software-oriented distributed shared cache management approach, infers a program's data affinity hints through a novel machine learning based analysis of its L2 cache access behavior. The OS utilizes the hints to guide proper data placement in the L2 cache with page coloring. The derived hints are independent of the program input and can be used for multiple runs. By off-loading the cache management task onto software, SOS deviates substantially from previously proposed hardware-based strategies and opens up a new opportunity for the CMP cache optimization. Our experimental results demonstrate that SOS is very effective in reducing the number of remote cache accesses. By using the hints for guiding page coloring alone, SOS achieves an average speedup of 10% and up to 23% over the shared cache scheme. When hints are used to direct data replication, SOS secures an additional performance gain of 9%, performing 19% better than the shared cache scheme on average.**

***Keywords*-CMP; NUCA; OS; Page Coloring; Performance;**

I. Introduction

Chip multiprocessor (CMP) architectures integrate multiple relatively simple processors on a single chip to exploit thread level parallelism, unlike previous single-core architectures that depend on complex hardware to extract instruction-level parallelism [3], [23], [29]. CMP architectures have advantages over single-core architectures in terms of design complexity, performance scalability and power efficiency. However, integrating multiple processors on a single chip dramatically increases the pressure on the memory subsystem [13], and designing an effective on-chip memory hierarchy remains a challenge. Researchers have paid considerable attention to the last-level cache design, which tends to have large capacity and is broken into smaller slices (or banks) distributed across the chip. Increasing wire delays cause the Non-Uniform Cache Architecture (NUCA) to become an inevitable choice [17]. With the variable cache access latency introduced by NUCA and very slowly improving memory access time, efficient management of the on-chip cache hierarchy is critical to the CMP performance.

Conventional distributed cache organizations are the shared cache [14], [18], [22], [26] and the private cache [1], [11], [30]. The shared cache provides a logically shared cache view out of physically distributed cache slices by interleaving consecutive cache blocks among them. While the shared use of all cache slices maximizes the cache capacity utilization, the blind data distribution of the shared cache makes its performance sensitive to the program location on the chip and the program's cache access pattern. Without a significant reduction of wire delays, the shared cache is obviously not scalable to large-scale CMPs. The private cache avoids excessive remote accesses by always keeping copies of the accessed data blocks in the processor's local cache slice. Each processor becomes an autonomous unit, making the CMP easier to scale. However, uncontrolled data replication and strict capacity partitioning lead to significant under-utilization of the total cache capacity; the increased miss rate can easily offset the benefit of low-latency local accesses.

Many recent proposals try to combine the advantages of these two basic schemes [2], [5]–[7], [34]. While the previous proposals improve on the baseline shared and private cache organizations, they have common drawbacks: performance benefits come with the introduction of complex and potentially expensive hardware structures. Furthermore, hardware-based schemes are typically optimized for a specific cache access patterns and may not perform well for the programs lacking such patterns. Finally, centralized hardware structures in some proposals may create performance bottleneck for large-scale CMP architectures.

In this work, we propose SOS, a novel software-oriented approach for managing the NUCA L2 cache of a CMP to improve the performance of latency-oriented multithreaded applications. By off-loading the cache management task onto software, SOS can exploit a program's cache access behaviors that are hard to capture and exploit with hardware mechanisms. SOS employs an one-time profiling of a program (at compile time) to characterize effective data access patterns using the K-means clustering algorithm. The gen-

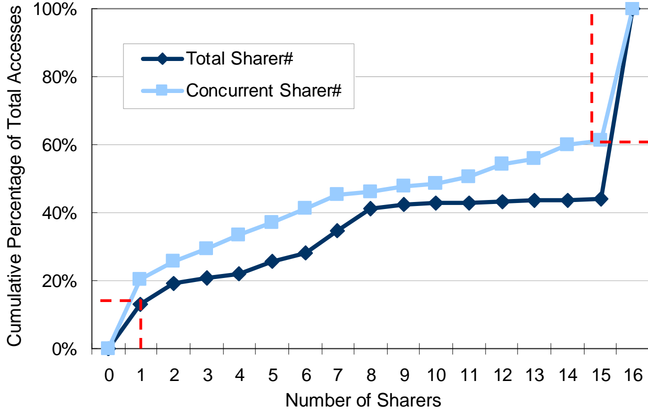

Figure 1. Data sharing behavior of cholesky at page granularity. The two curves capture the number of memory references to a page shared by a different number of threads. For the dark curve ("total sharer #"), the number of sharers is determined over the entire program run. For the light curve ("concurrent sharer #"), the number of sharers is determined at any instant when an access is made.

erated data affinity hints, independent of the program input, are opportunistically used to guide the OS page coloring for improved data locality. To the best of our knowledge, our work is the first to study a software-oriented NUCA cache management strategy for latency-oriented multithreaded applications. Moreover, SOS is orthogonal to other hardware schemes and can be used in combination with synergy. Our experimental results demonstrate that by using hints to guide page coloring alone SOS performs 10% better than the shared cache on average and up to 23% for the benchmark programs studied. When data replication is enabled and guided by SOS, it brings an additional performance gain of 9%, performing 19% better than the shared cache on average.

This work has been motivated by our observation that much of the shared data in a program are accessed either predominantly by a single thread or by almost all the processors; moreover, such behavior for a given data object does not change significantly with the program input. Figure 1, as an example, presents the data sharing behavior of cholesky from the SPLASH-2 benchmark suite [27]. Nearly 17% of the total references are shown to go to pages predominantly accessed by a single thread. At the top right of the plot, we also identify around 40% of the total references accessing pages that are shared by all threads. We find this "zigzag" pattern common in the SPLASH-2 programs. Indeed, this is intuitive as parallel programs often involve partitioned data processing (private data) and global synchronization and data exchange (highly shared data). It is desirable to place private data in the requester's local cache bank early to avoid remote accesses. On the other hand, it is beneficial to replicate highly shared data or fetch them from neighbors nearby. This observation uncovers an excellent opportunity

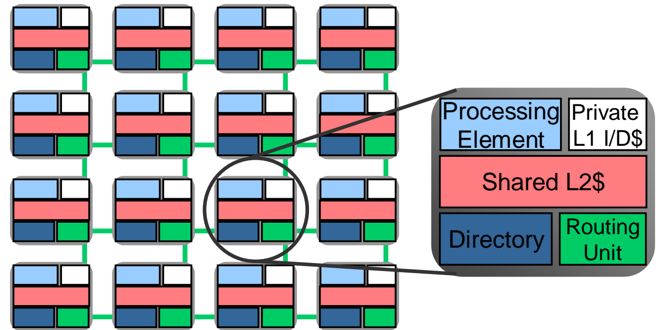

Figure 2. A tiled CMP architecture with a 2D mesh network.

for an offline analysis to derive data affinity hints and guide run-time data placement and data replication.

In what follows, we first summarize related work in Section II. We present SOS at length in Section III, with a particular focus on how to recognize and exploit the common memory access patterns in multithreaded applications, followed by detailed evaluation results in Section IV. Finally, Section V concludes. Without a special note, we assume a tile-based CMP architecture organized in a 4×4 2D mesh network throughout this paper, as shown in Figure 2.

II. Related Work

Zhang and Asanović [34] proposed the "victim replication" scheme based on a shared L2 cache organization, where each L2 cache slice can replicate remote cache lines replaced from its local L1 cache. Essentially, L2 cache slices provide a large victim cache space for the cache blocks whose home are remote. However, excessive replication can increase conflict misses considerably. Beckmann et al. [2] proposed a controlled victim replication design called "ASR," which tries to reduce cache pollution caused by excessive replication. In order to measure the best replication level, their design employs a set of tables within each processor to keep track of the performance gain and loss according to increasing or decreasing replication. These tables consume considerable chip area. The ASR scheme also relies on a ring network and a broadcast-based coherence protocol. This limits it from scaling up to a large CMP. Chishti et al. [6] proposed a cache design called "CMP-NuRAPID" having a hybrid of private per-processor tag arrays and shared data arrays. Based on the hardware organization, they studied a series of optimizations, such as controlled replication, in-situ communication, and capacity stealing. Compared to the shared cache, however, CMP-NuRAPID requires much more complex cache management hardware. Chang and Sohi [5] proposed a "cooperative caching" framework based on a private cache design with a centralized directory. They studied optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement, and global replacement of inactive data. However, the complex central directory

limits its scalability. Finally, Cho and Jin [7] proposed an on-chip cache management framework where memory data can be dynamically placed into any cache slice. By increasing the data mapping granularity from memory block to memory page, they showed that the OS memory management module can be conveniently extended to handle the task of on-chip L2 cache management. Their work, however, does not specifically study how to achieve high program performance when such a flexible data mapping mechanism is provided.

The problem of tackling non-uniformity in L2 cache latencies bears similarity to the problem of attacking disparate memory latencies in distributed shared memory multiprocessors such as the Non-Uniform Memory Architecture (NUMA) or Cache-Only Memory Architecture (COMA) machines [12]. Because the ratio between local memory accesses and remote memory accesses will largely determine the application performance in such a machine, it is of utmost importance to improve the data locality at the level of distributed main memory by carefully placing, migrating, and replicating pages [8]–[10], [19], [21], [31]–[33]. Our work is most similar to the hardware profile-guided page placement scheme proposed by Marathe and Mueller [21]. In their work, a truncated version of the program code is profiled before program execution. The sampled memory access trace is then used to decide the affinity for each touched page. This method has several limitations that we overcome in this work. First, it requires program profiling before each execution. The quality of the truncated code is crucial to the accuracy of the affinity information. Automatically generating the representative code for the whole program is shown to be difficult. Furthermore, they assume that dynamic memory allocation returns the same address in both the profiling execution and the normal program execution. This assumption is not always true and could hurt the program performance significantly when the assumption does not hold. SOS only profiles and analyzes the trace once at compile time. The generated hints are independent of the program input and can be used for multiple runs. Moreover, the run-time system can decide (for any reason) to skip optimization and simply fall back to the baseline shared cache scheme.

III. SOS Scheme

Figure 3 depicts the two major stages of the proposed SOS scheme: the one-time profiling and data analysis phase and the hint exploitation phase. In the first phase, SOS profiles a given program's L2 cache accesses with a test input. Based on the collected traces, access histograms are constructed for each page (buckets in each histogram count accesses from different processors). Then K-means clustering algorithm is applied on those per-page access histograms to derive page clusters. Given the page cluster information, SOS finally determines the patterns for dynamic and static data areas and attach those hints to the binary. At run-time stage, the OS

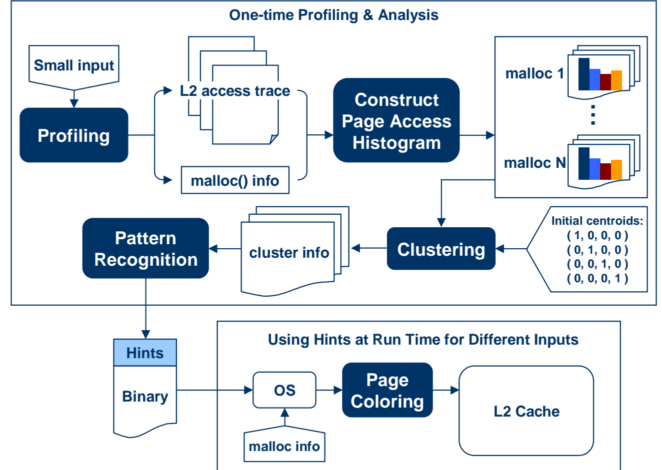

Figure 3. The one-time profiling and data access pattern analysis flow (upper box) and the hint exploitation flow (lower box) of SOS.

peels off the hints from the binary and uses the information whenever a new memory page mapping event occurs.

In this section, we first examine the common data access patterns found in multithreaded applications. We will then discuss in detail the data access pattern recognition algorithm of SOS and how the resultant data affinity hints are exploited to guide the OS cache management decisions. Finally, we will discuss the architectural support for SOS.

A. Access pattern classification

The memory footprint touched by a program can be generalized into *static data region* and *dynamic data region*. Global variables and data structures are those used to track program-wide information, to synchronize and to exchange data among threads. They are often assigned statically. The locations and sizes of these data are known prior to the program execution. This determinism makes offline affinity analysis for static data straightforward. Data inside a dynamic data region plays an important role in large-scale parallel programs. Often it is necessary to allocate memory regions dynamically since the varying input sets prohibit programs from claiming the memory space statically. *Malloc* is a typical library function to dynamically allocate memory regions in the C language. We use *malloc* in this paper as the indication of dynamic allocation in general. Dynamic data are usually the target of computation and tend to be accessed more frequently than static data. Accordingly, precise distribution of dynamic data can have a large impact on the overall program performance. For this reason, we focus our discussion of data access patterns on the dynamic data region. The programs we examined are selected from SPLASH-2 and PARSEC benchmark suites.

In this work, we classify a program's data access patterns into: *Even Partition*, *Scattered*, *Dominant Owner*, *Small-Entity*, and *Shared*, and examine each in the following.

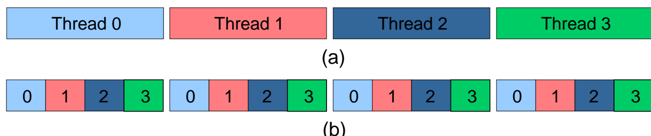

Figure 4. An example of a dynamically allocated memory area shared by four threads evenly.

Even Partition: In many scientific programs, a large one-dimensional data array is allocated at the beginning of the execution. The data array can be easily partitioned among threads due to its regular structure. The array index is commonly a function of thread ID and some loop indices. For example, the following code illustrates how the array is allocated by the main thread and how each thread accesses its partition with its thread ID (*ProcNo*).

```
Main thread:
Array = malloc(sizeof(int) * NumProc * N);

Thread [ProcNo]:
for(i = 0; i < N; i++)
    Array[ProcNo * N + i] = i;
```

This access pattern leads to an even partitioning of the whole data array among threads, which presents a good opportunity to distribute data in the L2 cache as shown in Figure 4(a). Sometimes data array accesses can be interleaved in a finer granularity, such as:

```
Thread [ProcNo]:
for(i = 0; i < N; i++)
    Array[i * NumProc + ProcNo] = i;
```

The corresponding data distribution pattern is shown in Figure 4(b) which is essentially the same as the previous one. The compiler can always rearrange the data layout and transform it back to the pattern in Figure 4(a). Benchmarks fft, lu and blackscholes have this kind of data access pattern.

Scattered: At times, the workload assigned to each thread is not balanced or the programmer chooses a separate memory area to allocate for each thread. Hence, unlike the "even" pattern, the whole data set is scattered into multiple memory regions. A common way to declare these regions is to allocate memory area in the thread body. For example:

```
Thread [ProcNo]:
Array = malloc(sizeof(int) * N);

for(i = 0; i < N; i++)
    Array[i] = i;
```

Since the allocated memory area is exclusively accessed by the owner, the data distribution is straightforward. The allocation can also be done in the main thread, especially when workload distribution is unbalanced or a multi-dimensional array is used. The following code depicts the scenario:

```
Main thread:
ArrayPtr = malloc(sizeof(int) * NumProc);
for(i = 0; i < NumProc; i++)
    ArrayPtr[i] = malloc(sizeof(int) * Size[i]);

Thread [ProcNo]:
for(i = 0; i < Size[i]; i++)
    ArrayPtr[ProcNo][i] = i;
```

Note how *ProcNo* plays a role in addressing array elements. Data placement of this pattern is simple as those in-order allocated areas are exclusively accessed by each thread using *ProcNo*. This is the most common access pattern in the multithreaded programs we examined. The representative benchmarks that more or less exhibit this pattern are barnes, cholesky, fmm, ocean, and radix.

Dominant Owner: There are occasions when shared data areas are mostly accessed by only one thread. These memory areas are commonly allocated for auxiliary structures in the main thread. They help record temporary information while initialization progresses. We regard these areas as private because it is logical to place these data in a tile, where they are accessed the most. Benchmark radiosity is an example, which has a global data structure accessed a lot by the main thread.

Small-Entity: When the program data are organized by a linked list, tree or graph, it usually involves intermittent allocation and freeing of nodes. A small trunk of memory area can be repeatedly allocated and reclaimed by multiple *malloc* and *free* instances. This poses extreme difficulty for tracking memory usage and managing data at coarse granularity. Benchmarks cholesky, raytrace and swaption show this behavior. Another representative case is a data stack. On function calls and returns, the data stack expands and shrinks accordingly. However, unlike the previous case, the ownership of stack data is explicit. This is because data items in the stack are used as function parameters and local variables, which are almost always private to threads.

Shared: The last category contains all data areas that could not be classified into one of the previous types. These areas are highly shared by multiple threads. No particular affinity pattern can be found in these areas, or the pattern changes under different inputs. However, they can be further separated into read-only sharing or read-write sharing. Because the input change seldom affects the read/write behavior of the data, it is safe to mark regions of read-only sharing as replication candidates.

This work does not attempt to recognize all possible data access patterns that may prove useful. By focusing on the most frequently observed access patterns we presented in this section, we aim to motivate SOS. We note that our study revealed other more complex memory access patterns in the programs we studied. We leave the strategies to efficiently discover and exploit such patterns to our future work.

B. Access pattern recognition

Our goal of this step is to derive data affinity hints for the pages in dynamic regions, that can be used across different input sets and architecture configurations, such as different cache sizes. In addition, choosing a flexible way to represent

these hints is important for this method to become effective. It is relatively straightforward to provide hints for the static data regions that are determined at compile time. However, the location and size of a dynamic data region are unknown until the *malloc* returns. Our strategy is to associate one dynamic hint with each *malloc* instance, which can be identified uniquely by the file name, the line number in source code, and the number of times it has been called. Every dynamic hint only expresses which pattern the area allocated by this *malloc* would exhibit, instead of giving specific mapping details. Only at run time, when the address and size of a dynamic area are determined, the actual page to cache mappings are generated. These hints can be embedded in the program binary and loaded into the system before being utilized by the OS. When a page fault occurs, hints are consulted to derive the page location among the L2 cache slices.

The overall flow of the proposed SOS approach is illustrated in Figure 3. In order to analyze the access patterns of dynamically allocated memory regions, we profile the program with a small, reasonably representative input set once. During profiling, we collect the L2 cache read trace from each tile and the range of each *malloc* instance. Then we process each reference from the trace by checking it against all *malloc* ranges. The reference is a dynamic access if it falls into one of the *malloc* ranges. Each page within the *malloc* range is associated with a counter vector. When the page receives a reference from a tile, the counter corresponding to that tile is incremented. After the trace has been processed, the vector of each dynamic page represents the access histogram for all tiles. Each vector is normalized by the maximum counter value within it. *K-means clustering* is then performed on these vectors for each *malloc* range. An example of the initial centroids for a 4-tile CMP is:

```
C0 (1, 0, 0, 0)
C1 (0, 1, 0, 0)
C2 (0, 0, 1, 0)
C3 (0, 0, 0, 1)
C4 (1, 1, 1, 1)
```

The K-means clustering algorithm works as follows:

```
do (
  1. Assign each vector to the nearest
     cluster centroid based on Euclidean distance.
  2. Calculate distance between the vector and the
     centroid as the error.
  3. Accumulate the error for this iteration.
  4. Update the new cluster centroids by averaging
     vectors within each cluster.
  5. Calculate the error difference between
     the current and the last while iteration.
} while(error difference > threshold)
```

After the clustering procedure finishes, each cluster contains many vectors corresponding to pages in the dynamic memory area. All pages in cluster 0 (*C0*) are accessed mostly by tile 0. All Pages in cluster 1 (*C1*) are accessed mostly by tile 1. This applies to all clusters, except the last one (*C4*), where pages are accessed almost equally by all tiles.

Figure 5. (a) An example of the even partition pattern. (b) Page clusters after K-means clustering of the access histograms.

Next, we want to recognize the patterns discussed in the previous subsection based on the clustering result. The even partition pattern is checked by counting the number of “right pages” in each cluster. For instance, suppose a dynamic range has 16 pages and the profiling is done on a 4-tile CMP. Figure 5(a) shows the desired even partition pattern, where each partition receives 4 pages. The clustering result is given in Figure 5(b). As illustrated, the partitions may not be perfectly even in reality. To recognize the pattern, we classify a cluster as a *fitting cluster* if half of the pages from the desired partition range reside in the cluster. The *malloc* is said to have the “Even Partition Pattern” if more than 75% of all clusters are fitting clusters.

Recognizing the “Scattered Pattern” is similar, except that now we need to perform the clustering algorithm for all *malloc* instances. As shown in Figure 6, the *malloc* is called four times as it is defined in a *for* loop. The aggregated space of these four *malloc* ranges has 16 pages. The area allocated by the first *malloc* is mostly accessed by tile 0. The area allocated by the second *malloc* is used mainly by tile 1 and so on. If half of the pages in the *malloc* range are assigned to the same cluster, the cluster number is checked against the number of times this *malloc* is called. If more than half of the instances of the same *malloc* have the matching cluster number, this *malloc* is defined to have the “Ordered Scattered Pattern.” Otherwise, the *malloc* instances might be called by the parallel threads. In such a case, the *malloc* is defined as “Private Scattered Pattern.” One special case for the example in Figure 6 is that *malloc* is only called once and the majority of the pages in the range are assigned to the same cluster. This is a clear sign that the *malloc* has the “Dominant Owner Pattern.” Lastly, if most of the pages are clustered into the last group (*C4*), the *malloc* is not associated with a recognizable pattern. It has the “Shared Pattern.”

Finally, the above algorithm can be applied to the static pages in a program. In fact, the access pattern for each static page is deterministic and a simple page number to tile mapping can be used as a hint. For instance, when the number of tiles is fixed, we can count the number of accesses from different tiles to determine which tile accessed a given page the most. If the access count from a particular tile is more than 50% of the total accesses, the page is assigned

Figure 6. An example of the clustering results of four instances of the same *malloc*.

to that tile. Otherwise the page is marked as "Shared."

C. Hint exploitation

The recognized access patterns are encoded and embedded into the program binary so that they can be exploited during later program executions. Since hints for dynamically allocated data and static data are different, they are presented separately. Static hints give the target tile for each static page explicitly, thus can be used directly at run time. Dynamic hints only tell the type of each *malloc* instance in the source code. They need to be translated into actual mappings when the corresponding *malloc* returns the dynamic area range.

Even Partition Pattern: After the *malloc* returns the range of the dynamic area, the space is divided into equal-sized pieces and then assigned to tiles in sequential order.

Ordered Scattered Pattern: A counter with initial value of 0 is associated with each hint of this type. Every call to the corresponding *malloc* triggers the assignment of the dynamic area to the tile indexed by the counter value. Then the counter is incremented.

Private Scattered Pattern: If the hint indicates that a *malloc* has this pattern, the returned area is assigned directly to the tile, who calls this *malloc* instance.

Dominant Owner Pattern: For this pattern, the target cache slice number comes with the hints. The allocated dynamic area is assigned to the target tile.

Shared Pattern: By default, no effort is made to optimize for dynamic data regions in this pattern. All data are distributed at cache line granularity instead of page size to balance the L2 cache pressure. However, this can be a good hint for data replication to improve temporal data affinity. As we correctly place private data, the amount of data replication and the resultant conflict misses can be reduced significantly.

D. Architectural support

SOS requires minimal hardware modification. The pattern analysis and hint generation are done in software before run time (i.e., compile time). A compiler can easily intercept dynamic memory allocation information by inserting a wrapper function around each *malloc*. Hints are encoded into the program binary and loaded before program execution. Strictly being "hints," our data affinity information causes no harm if the OS and the hardware do not support any cache-level data affinity optimization. SOS simply falls back to the plain shared cache scheme. We note that partitioning the data at page granularity has limited negative impact on the performance as pages usually have uniform access behavior.

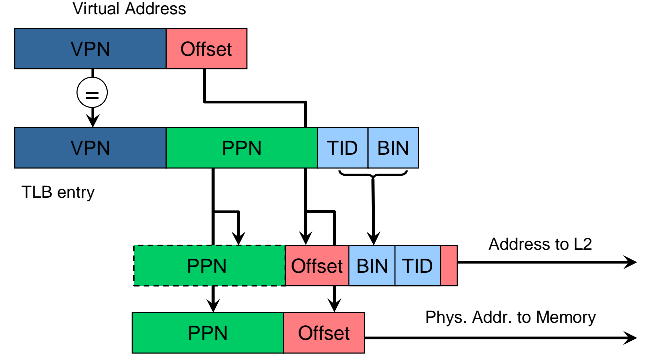

Figure 7. Each TLB entry is augmented with tile ID (TID) and cache bin (BIN) fields. These two fields together with the higher bits from the page offset are used to index the L2 cache. The L2 cache tag field is extended to accommodate the full-length PPN. Page locations in the L2 cache and in the memory are decoupled. Similar mechanisms have been previously used [16], [25].

However, a slight change to the page table and the TLB entries is required, as shown in Figure 7. In order to control page placement in the L2 cache flexibly, the extra tile ID (*TID*) and the cache bin index (*BIN*) fields are attached to the page table and the TLB entries. Values for these two fields are determined at the time of page faults by the OS using hints. If a page does not have a matching hint or the hint indicates it is a highly shared page, the lower bits of the cache block address are used to generate *TID* and *BIN*. Adopting the lower bits from cache block address essentially distributes the whole page across tiles at cache line granularity. On the other hand, if the *TID* and *BIN* values are given in the hint (or *BIN* can be generated randomly at run time to remove hot-spots), they are used to fill the corresponding fields in the page table and the TLB entry. Virtual address to physical address translation for memory access is the same as before. However, the physical address sent to the L2 cache needs special handling. Values in the *TID* and *BIN* fields together with bits from page offset form the cache index address as shown in Figure 7. The whole *PPN* is used as cache tag. This is necessary to guarantee that a given physical address uniquely maps to a cache block.

The presented extra fields incur a small storage overhead, while offering the required flexibility for SOS. Assuming a 16-tile CMP with 128KB 8-way associative L2 cache slices, *TID* and *BIN* are 4 bits and 1 bit respectively. This incurs around 8% increase in the page table size (64-bit address). The extra tag bits introduce less than 1% the cache area overhead for a 64-byte cache line. The *TID* and *BIN* values are assigned at the page mapping time. They are persistent until the page is replaced out of main memory. Therefore there is no consistency issue, and no TLB flush is required.

Component	*Parameter*
Processor Model	in-order
Issue Width	2
L1 I/D Cache	
Cache Line Size	64 B
Cache Size / Associativity	8 KB / direct-mapped
Load-to-Use Latency	2 cycles
L2 Cache	
Cache Line Size	64 B
Cache Size / Associativity	128 KB / 8-way
Tag Latency	2 cycles
Data Latency	6 cycles
Replacement Policy	Random
Network on Chip	
Topology	4×4 2D mesh
Hop Latency	3 cycles
Main Memory Latency	300 cycles

Table I
BASELINE ARCHITECTURE CONFIGURATION.

Program	*Small*	*Median*	*Large*
barnes (particle)	16K	32K	64K
cholesky	tk15.O	tk16.O	tk29.O
fft (point)	256K	1M	4M
fmm (particle)	16K	32K	64K
lu (matrix)	512	1024	2048
ocean (grid)	258	514	1026
radiosity	test	room	largeroom
radix (key)	4M	8M	32M keys
raytrace	teapot	car	balls4
volrend	scaled4	scaled2	head
water-ns (molecule)	512	1000	2744
water-sp (molecule)	512	1000	4096
blackscholes (option)	64K	128K	256K
swaption (swaption)	4K	8K	16K

Table II
BENCHMARKS WITH SMALL, MEDIAN AND LARGE INPUTS.

IV. EVALUATION

A. Experiment setup

To evaluate SOS, we constructed a detailed CMP memory system simulator by extending the Simics [28] timing interface. It models a 16-tile CMP with a 4×4 2D mesh on-chip network as shown in Figure 2. Each tile has a two-issue in-order processor and private L1 instruction and data caches. The distributed L2 cache slices are shared across the chip. Cache coherence is enforced by a distributed directory-based coherence protocol with MESI states [20]. Network contention is modeled within routers. Table I describes the baseline architecture configuration. Our architecture parameters have been derived from recent multicore processors with "light" processor cores [4], [15], [18] as we explore a relatively large-scale processor with 16 tiles. The extra level of page address translation as described in Figure 7 is done in the simulator to avoid the OS modifications. The simulator is responsible for maintaining the translation table and looking up the target tile ID when a virtual page number is given. Likewise, data affinity hints are fed into the simulator directly. The overhead of making affinity decision in the OS is a small one-time cost and is ignored in our evaluation.

We experiment with 12 programs from the SPLASH-2 benchmark suite [27] and 2 programs from the PARSEC benchmark suite [24]. They are listed in Table II with associated inputs. There are a number of reasons for picking up these programs. First, since the experiment involves page placement actions, most of which are done at the very initial stage of program execution, it is necessary to simulate the programs from the beginning to the end. That is, "fast forwarding" is not an option. Given the slow speed of a detailed CMP simulator, we were not able to simulate very large applications. Second, in this work we manually replace dynamic memory allocation function calls in the source code by wrapper functions to capture dynamic memory allocations instead of implementing a full-blown compiler. This method works well for C programs. Third, our main focus in this work is on multithreaded applications. The SPLASH-2 and the PARSEC benchmark suites are the most commonly and widely used programs in the research community today. In order to evaluate the generality of the proposed approach, we pick three different input sets for experiments. We use the small input set to collect traces, and then use the median and large input sets to report results.

We evaluate and compare the shared cache (L2S), the private cache (L2P), the victim replication scheme (L2VR) and the page coloring scheme using hints provided by SOS (L2H). Finally, they are compared with our SOS scheme (SOS). SOS not only provides hints about which data are private and where they should be placed, does it also suggest which shared pages are beneficial to have data replicated at run time and which should not. A data replication cost analysis can be employed to derive the information. For instance, both high data temporal locality and read-to-write ratio can offer good indication for triggering/suppressing data replication. In this study, however, we simply consider all shared data as the replication candidate.

B. Results

1) Hint accuracy: The first set of results demonstrate the efficacy of the proposed data access pattern recognition algorithm. Figure 8 shows the breakdown of the L2 cache accesses of the studied benchmark. We do not show the part for the Dominant Owner pattern as it is almost negligible. In case of cholesky, the algorithm recognizes that around 15% of the total accesses exhibit the Private Scatter pattern. Nearly 80% of the total accesses come from the shared data areas. These numbers are consistent with the profile result presented in Figure 1 (shown in Section I). Other benchmarks such as raytrace, volrend, water-nsquared and water-spatial also have a very large number of accesses to shared data. This is aligned to the data structures of these programs—they naturally have a lot of data sharing.

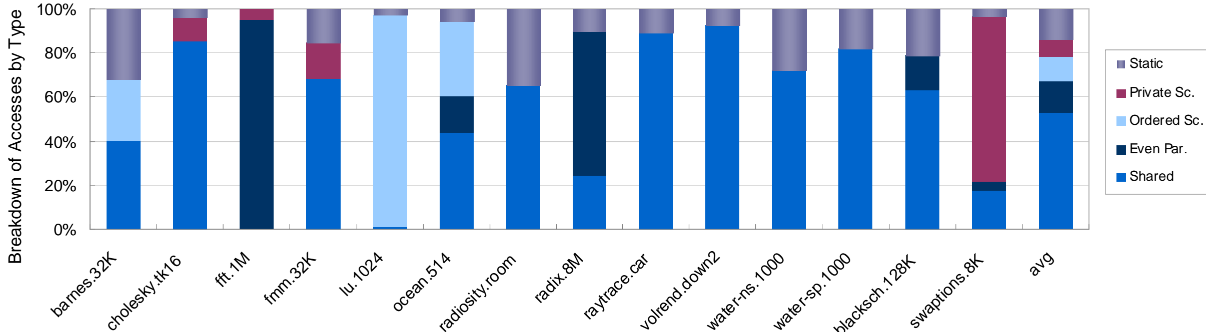

Figure 8. Breakdown of L2 cache accesses based on the classified pattern types (median input set).

Program	*Small*		*Median*	
	Accuracy	*Coverage*	*Accuracy*	*Coverage*
barnes	82.1%	47.8%	84.6%	43.3%
cholesky	82.9%	7.3%	85.9%	9.0%
fft	96.1%	53.7%	99.0%	69.4%
fmm	88.2%	28.1%	90.2%	28.5%
lu	96.7%	77.1%	98.3%	87.4%
ocean	99.0%	48.9%	98.7%	52.5%
radiosity	97.7%	26.8%	96.6%	33.5%
radix	90.4%	69.0%	66.1%	54.3%
raytrace	68.4%	7.9%	31.7%	3.9%
volrend	80.4%	9.7%	79.6%	8.0%
water-ns	45.0%	25.2%	45.0%	25.7%
water-sp	67.2%	16.6%	67.2%	17.2%
blackscholes	83.7%	34.2%	60.6%	29.8%
swaption	60.8%	44.8%	61.7%	47.3%

Table III
PATTERN RECOGNITION ACCURACY AND COVERAGE.

In contrast, fft, lu, ocean, radix and swaptions exhibit abundant private data accesses that are captured by our algorithm. Overall, the recognized patterns represent more than 50% of the total L2 accesses.

To further examine the effectiveness of the hints derived by SOS, we collect the access histogram of each page and its page location suggested by the hints during the simulation. The data affinity hint is considered “accurate” and a page is considered “accurately colored” if the predicted local processor accesses the page the most as indicated by the hint. We define the *hint accuracy* as the ratio of the number of accurately colored pages to the total number of colored pages. Note that we do not count the shared pages in this calculation. The accuracy metric measures how good our algorithm is at identifying and representing the access patterns. We define another metric called *coverage*, which is the ratio of the number of accesses to the “accurately colored” pages and the total L2 cache accesses. This metric helps us to understand how much impact our affinity hints would have on the overall L2 cache accesses. Table III shows the results with the small and median input sets. The hints derived by SOS achieve a high accuracy of over 80% in most cases. The exceptions are raytrace, water-ns, and water-sp. These programs use complex data structures such as trees and 3D matrices during computation. Data structures are updated constantly, leading to many *malloc*s and *free*s, which poses a challenge for our current offline pattern analysis framework.

The coverage varies from program to program. Some programs such as lu, ocean, and radix have good data affinity and well recognized data partitions, thus achieving a high coverage ratio. Others such as cholesky, raytrace, and volrend have low coverage. This means the cache access patterns of major data regions are not recognized since they are widely shared by threads. Interestingly, the accuracy and coverage of median input set and those of small input set are very close. This proves that the hints provided by SOS are stable across different input sets. In some cases, the accuracy and coverage are even higher for the median input set even though the hints are derived from the small input set. This can happen because the target data area becomes larger with the median input set, capturing relatively more accesses. We do not show the result for the large input set because the result is nearly identical to that of the median input set.

2) Performance improvement: Let us turn our attention to the program performance and behavior on the studied machine architecture. Figure 9 shows the normalized execution times of the five cache management schemes for the small, median and large input sizes as given in Table II. Execution time is normalized to L2S, the baseline design. Since we derive the hints from small input, Figure 9(a) provides a measure of how our hints perform under ideal situation. Comparing the results in Figure 9(a), (b) and (c), it is obvious that the relative performance of these schemes change little with different input sizes. One major reason is that many of these examined benchmarks are well optimized and have relatively small working set sizes that do not scale with the benchmark input. This explains why L2P performs better than L2S most of the time. The results also demonstrate that our offline pattern recognition algorithm performs robustly for different input sizes as L2H consistently provides around 10% performance improvement over L2S. This proves that our proposed algorithm captures existing program access patterns effectively. The patterns are stable and help color memory pages correctly even with changed input. In the following discussions, we will use

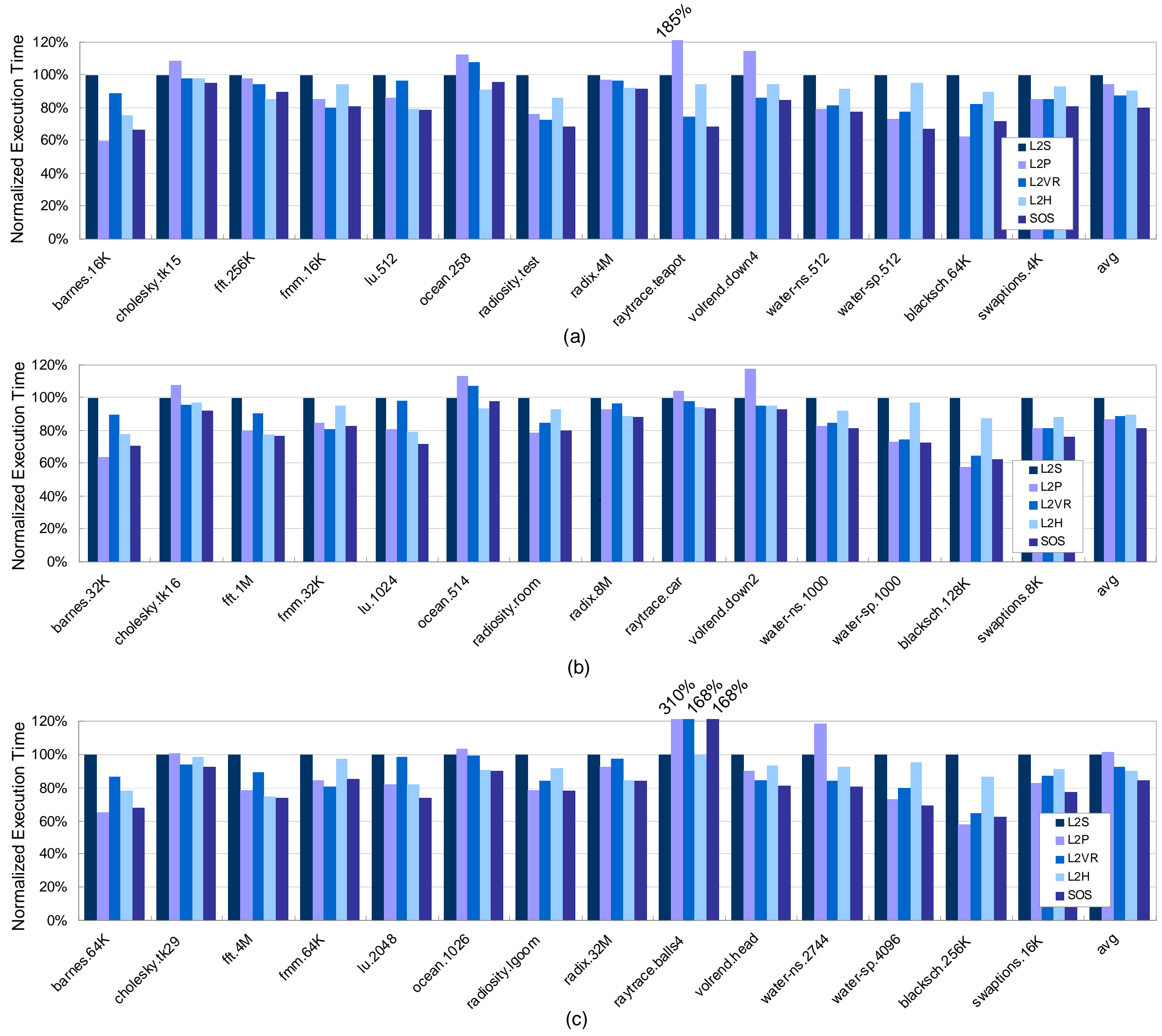

Figure 9. Execution time with (a) median and (b) large input set. Results are normalized to those of the shared cache scheme (L2S).

results with the median input set unless otherwise noted.

Let us take a look at Figure 9(b) in more detail. First, L2P performs considerably better than L2S due to the small program working set size. The exceptions are cholesky, ocean, raytrace and volrend, where data sharing is relatively high. For the same reason, L2VR is also very effective, achieving around 11% execution time improvement over L2S and similar to L2P on average. L2VR brings the L2 cache access latency close to L2P through replication. L2H approaches the performance of L2P and L2VR very well, achieving 10% execution time improvement over L2S. The improvement of L2H comes from the optimized data affinity for those data used mostly by one thread. No effort is made by L2H to tackle highly shared data. When hints are used to direct data replication, SOS improves performance by nearly 9% over L2VR and L2P. It marks 19% performance improvement over L2S. These results demonstrate that L2H is complementary to other hardware-based optimization techniques. When hints are used to control both private data placement and shared data replication, SOS removes unnecessary replications by allocating private data locally in L2 cache. This helps reduce the cache pressure significantly, resulting in fewer off-chip accesses. On the other hand, L2VR cuts the remote access latency of L2H by duplicating shared data in local L2 cache. SOS entertains the advantages of both private cache and shared cache schemes.

3) Miss rate comparison: Figure 10 provides more insights about the performance difference in Figure 9(b). It shows the ratio of total number of off-chip reads to the total number of L2 accesses. In general, L2P incurs more

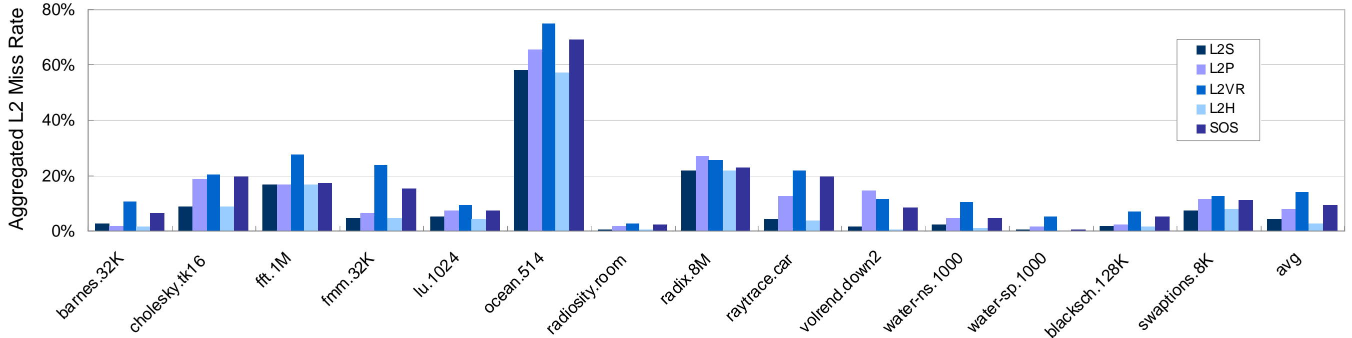

Figure 10. Aggregated L2 cache miss rates with the median input set.

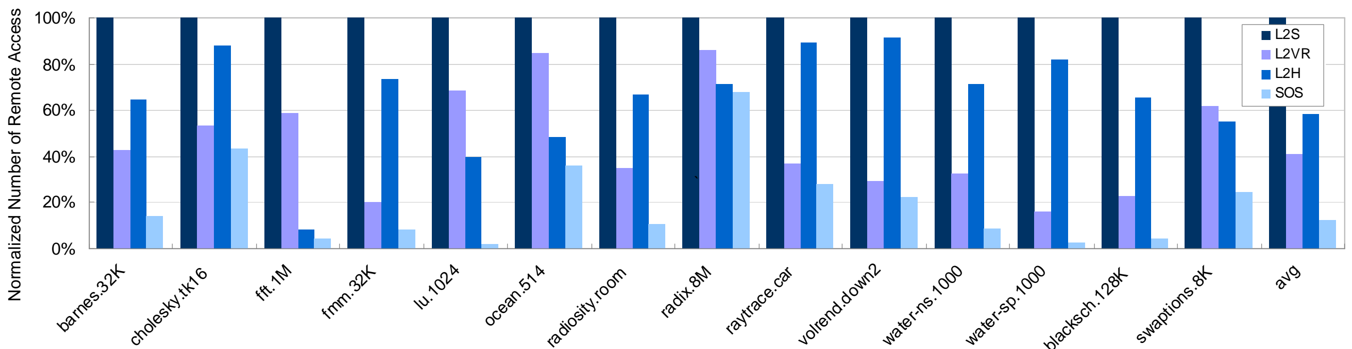

Figure 11. Number of remote L2 data requests of the three shared cache variants: L2VR, L2H, and SOS, normalized to that of L2S.

off-chip accesses than L2S due to conflict misses caused by smaller effective L2 cache size. In case of cholesky, ocean, raytrace and volrend, L2S performs better than L2P because the gain of more on-chip accesses in L2S is large enough to compensate for the loss in longer remote access latency. However, the margin is very small for the rest of the benchmarks. L2VR increases the cache pressure even more than L2P as it has to duplicate private data that are blindly distributed in L2S. This explains why L2VR has worse performance than L2P in Figure 9(b). L2H has a comparable off-chip access rate as L2S since it is basically the shared cache with improved data locality. SOS mitigates the cache pressure of L2VR by correctly placing the private data. It effectively improves the L2 cache miss rate of SOS to closely match that of L2S. However, unlike L2P, SOS writes back the modified cache line directly to its home node. This eliminates the need for expensive three-way cache-to-cache transfer of modified data when requested later by other threads.

4) Remote access reduction: The other determinate factor of the distributed shared cache performance is the L2 cache access latency which is affected by the number of remote accesses. Figure 11 shows the number of remote data requests for L2S and its variants L2VR, L2H and SOS, normalized to that of L2S. These four schemes have a similar number of L1 misses. Therefore, the more remote L2 data requests there are, the longer the average L2 access latency. Figure 11 shows that L2VR effectively removes nearly 60% of the remote accesses of L2S by duplicating clean L1 victims in the local L2 cache slice. However, the remote access reduction comes at the sacrifice of the decreased L2 cache hit rate as shown in Figure 10. L2H has on average 40% less remote accesses than L2S without any compromise in the L2 cache hit rate, since it essentially rearranges the data distribution of L2S. Because L2H only optimizes the private data while L2VR replicates any clean data blocks, it has less remote access reduction than L2VR. Surprisingly, SOS eliminates over 85% the remote accesses of L2S on average. There are two reasons for this result. First, SOS reduces the number of remote accesses by correctly distributing data in the local L2 cache in the first place. This also eliminates the need for replicating those data as in L2VR. Second, a smaller amount of victim replication leads to lower cache pressure, which in turn preserves more replications of shared data in the local L2 cache.

V. Conclusions

This paper proposed and studied SOS, a new software-oriented shared cache management approach for CMP architectures. We make the following contributions in this paper:

- We carried out a classification of the memory access patterns for latency-oriented multithreaded applications. Based on that, we proposed an efficient software-oriented shared cache management method, which is

substantially different from existing hardware-based schemes. Our approach is orthogonal to the hardware schemes, and can work together with them for even higher performance.

- We proposed a novel memory access pattern recognition algorithm based on the K-means clustering method. Our results show that the algorithm works well in recognizing those commonly seen access patterns for dynamic memory regions. The recognized patterns are independent across program inputs and can be used for multiple runs. This makes our scheme very flexible as the offline analysis needs be done only once at compile or profile time.
- We evaluated the proposed scheme and compared it with the shared cache, the private cache, and their variants. We showed that by applying the hints to guide page coloring and data replication on the shared L2 cache, it performs significantly better than both the shared cache and the private cache.

Our future work includes (1) exploring the access patterns for throughput-oriented workloads; and (2) exploiting more sophisticated access patterns to fully uncover the potential of our software-oriented cache management approach.

VI. Acknowledgement

This work was supported in part by NSF grant CCF-0702236. The authors thank Tomas Singliar (currently with Boeing) for his input on earlier drafts of this work and the anonymous reviewers for their constructive comments.

References

[1] AMD Dual-Core Processors. http://www.amd.com.

[2] B. M. Beckmann et al. "ASR: Adaptive Selective Replication for CMP Caches" *Proc. MICRO*, pp. 443–454, Dec. 2006.

[3] S. Borkar et al. "Platform 2015: Intel Processor and Platform Evolution for the Next Decade," *Tech.@Intel Mag.*, Mar. 2005.

[4] S. Bell et al. "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," *Proc. ISSCC*, pp. 88–89, 598, Feb. 2008.

[5] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. ISCA*, June 2006.

[6] Z. Chishti et al. "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Proc. ISCA*, pp. 357–368, June 2005.

[7] S. Cho and L. Jin. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," *Proc. MICRO*, pp. 455–465, Dec. 2006.

[8] A. L. Cox and R. J. Fowler. "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proc. SOSP*, pp. 32–44, Dec. 1989.

[9] M. Gupta and P. Banerjee. "Automatic Data Partitioning on Distributed Memory Multiprocessors," *Proc. DMCC*, pp. 43–50, 1991.

[10] M. Gupta and P. Banerjee. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE TPDS*, pp. 179–193, Vol.3 1992.

[11] J. M. Hart et al. "Implementation of a Fourth-Generation 1.8-GHz Dual-Core SPARC V9 Microprocessor," *IEEE JSSC*, 41(1), 2006.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*, 3rd Ed., Elsevier, 2003.

[13] J. Huh, D. Burger, and S. W. Keckler. "Exploring the Design Space of Future CMPs," *Proc. PACT*, pp. 199–210, Sep. 2001.

[14] Intel. "A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors," *Tech.@Intel Mag.*, May 2005.

[15] Intel. *Intel Atom Processor*, http://www.intel.com/technology/atom.

[16] L. Jin and S. Cho "Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches," *Proc. ICPP*, Sep. 2008.

[17] C. Kim et al. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proc. ASPLOS*, Oct. 2002.

[18] P. Kongetira et al. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, 25(2): 21–29, Mar.-Apr. 2005.

[19] R. P. LaRowe and C. S. Ellis. "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM TOCS*, 9(4):319–363, Nov. 1991.

[20] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. ISCA*, June 1997.

[21] J. Marathe and F. Mueller. "Hardware profile-guided automatic page placement for ccNUMA systems," *Proc. PPoPP*, March 2006.

[22] S. Naffziger et al. "The Implementation of a 2-core Multi-Threaded Itanium-Family Processor," *Proc. ISSCC*, Feb. 2005.

[23] S. Palacharla, N. P. Jouppi and J. E. Smith. "Complexity-effective Superscalar Processors," *Proc. ISCA*, pp. 206–218, May 1997.

[24] C. Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. PACT*, Oct. 2008.

[25] T. Sherwood, B. Calder, and J. Emer. "Reducing Cache Misses Using Hardware and Software Page Placement," *Proc. ICS*, June 1999.

[26] B. Sinharoy et al. "POWER5 system microarchitecture," *IBM J. Res. & Dev.*, 49(4/5):505–521, July/Sep. 2005.

[27] S. C. Woo et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. (ISCA)*, June 1995.

[28] Virtutech AB. Simics Full System Simulator. http://www.simics.com/.

[29] J. E. Smith and G. S. Sohi. "The Microarchitecture of Superscalar Processors," *Proc. IEEE* Dec. 1995.

[30] T. Takayanagi et al. "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE JSSC*, 40(1), Jan. 2005.

[31] S. Tandri and T. Abdelrahman. "Computation and Data Partitioning on Scalable Shared Memory Multiprocessors," *Proc. PDPTA*, 40(1), Nov. 1995.

[32] B. Verghese et al. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," *Proc. ASPLOS*, Oct. 1996.

[33] K. M. Wilson and B. B. Aglietti. "Dynamic Page Placement to Improve Locality in CC-NUMA Multiprocessors for TPC-C," *Supercomputing (SC)*, pp. 258–265, Nov. 2001.

[34] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proc. ISCA*, pp. 336–345, June 2005.