Utilizing Shared Data in Chip Multiprocessors with the Nahalal Architecture

Zvika Guz[1], Idit Keidar[2], Avinoam Kolodny[2], Uri C. Weiser[2]

Department of Electrical Engineering

Technion - Israel Institute of Technology, Haifa 32000, Israel

[1]zguz@tx.technion.ac.il [2]{idish, kolodny, uri.weiser}@ee.technion.ac.il

ABSTRACT

This paper addresses a new cache organization in a Chip Multiprocessors (CMP) environment. We introduce Nahalal, an architecture whose novel floorplan topology partitions cached data according to its usage (shared versus private data), and thus enables fast access to shared data for all processors while preserving the vicinity of private data to each processor. The Nahalal architecture combines the best of both shared caches and private caches, enabling fast accesses to data as in private caches while eliminating the need for inter-cache coherence transactions. Detailed simulations in Simics demonstrate that Nahalal decreases cache access latency by up to 41.1% compared to traditional CMP designs, yielding performance gains of up to 12.65% in run time.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles – *Cache memories.*
C.1.2 [**Processor Architectures**]: Multiprocessors.

General Terms

Management, Performance, Design.

Keywords

Chip Multiprocessors, Cache memories.

1. Introduction

The design of a memory hierarchy for an on-chip multi-processing environment is one of the key challenges introduced by the shift towards Chip Multiprocessors (CMPs). In particular, cache data access is often a principal bottleneck in such systems, as multiple threads compete for limited on-die memory resources and accessibility. CMP environment necessitates a fresh look at cache design, and cannot directly inherit the traditional principles and know-how's of uniprocessor architectures.

A major factor that impacts the performance and energy consumption of cache access in modern CMP designs is wire delays: as global wire delays become a dominant factor in VLSI design [1][2][3][4], on-chip cache access times and power dissipation increasingly depend on the distance between the processor and the data. Hence, modern cache structures are no longer monolithic, but instead are comprised of multiple *banks* that can be accessed simultaneously by different processing elements. Access times to all banks are not the same, but instead depend on distances in the on-die layout. This is called a *Non Uniform Cache Architecture (NUCA)* [5]. (See further details in Section 2.)

Current L2 cache designs are typically based either exclusively on *private caches* or exclusively on *shared caches*. A private cache is a cache associated with a single core, while a shared cache is a cache shared among multiple cores. The Intel Core™2 Duo processor family uses a shared cache [6], whereas AMD's Athlon™ 64 X2 Dual-Core processors use a private L2 for each processor [7]. AMD's next processors family, Barcelona, uses a private L2 for each core, and all the cores share an L3 cache [8].

There are tradeoffs between these two choices. An important advantage of private caches over shared ones is the proximity of data to the processor that uses it, which yields fast access times and low energy consumption. On the other hand, private caches entail a static partitioning of the total capacity among the cores, which may lead to inefficient use of the cache capacity when the working sets of the different cores vary in size. Moreover, *shared data*, which is accessed by more than one core, needs to reside in multiple copies in private caches, further reducing the effective capacity. In order to manage such replicated data, a cache coherence protocol needs to be implemented. This complicates cache management and burdens the system with coherence transactions.

In contrast to private caches, shared caches may choose to store only a single copy of each data line, and thus eliminate the need for maintaining coherence among different copies of the same data. Storing single copies also increases the overall effective cache capacity. Since the gap between external memory latencies and on-chip access times continues to grow [9], such higher cache utilization will be of particular importance in future architectures. Despite all of these advantages, shared caches have one critical shortcoming that renders them inefficient, namely, costly access to shared data [10][11][12]. In shared NUCA solutions, shared cache lines inevitably reside far from some of their client processors, resulting slow and expensive access. (See further details in Section 4.)

In order to understand the gravity of this problem, we conducted an extensive study of memory access patterns in a broad range of multi-threaded applications. (Our findings are reported in [12]; since the focus of this paper is on architecture rather than application study, we forgo them here, but for

SPAA'08, June 14–16, 2008, Munich, Germany.

completeness, include them in the Appendix). Our findings show that in many multithread applications, a substantial fraction of the memory accesses involved such shared cache lines [10][12][13]. Furthermore, in commercial and transactional memory workloads, a significant fraction of memory accesses involve modified-shared data, which cannot be replicated without a performance penalty for ensuring cache coherence. These findings illustrate the importance of designing adequate solutions for shared data when designing CMP architectures. This is where current state-of-the-art solutions (based on either shared or private Last Level caches) fall short. Our finding further show that although shared lines consume a substantial fraction of the total accesses to memory, these lines comprise only a small fraction of the total working set. Moreover, most of the shared data is shared by all processors. We dub this phenomenon the *shared hot-lines* effect.

In this paper, we leverage the shared hot lines effect in the design of a novel cache architecture. We propose Nahalal, a novel CMP cache architecture that treats shared data as a first class data citizen, and leverages the best of both the shared and private cache approaches: Like shared caches, it allows for flexible allocation of cache lines accounting for differences in working set sizes; it further supports storing a single copy of a shared data line, eliminating storage waste and the need for cache coherence transactions. Like private cache architectures, Nahalal preserves the vicinity of reference for processors to both shared and private data.

Nahalal combines the two types of caches (private and shared). Its floorplan resembles the layout of the cooperative village Nahalal (see Figure 1), which, in turn, is based on an urban design idea from the 19th century [14]. In the village Nahalal, public buildings are located in an inner core circle of the village, enclosed by a circle of homesteads. Private tracts of land are arranged in outer circles, each in proximity to its owner's house. We project the same conceptual layout to CMP. A fraction of the L2 memory capacity budget is used for hot shared data, and is located in the center of the chip, enclosed by all processors. The rest of the L2 memory is placed in the outer area of the die, and provides private storage space for each core.

Figure 1. Aerial view of Nahalal village.

To demonstrate the Nahalal concept, in Section 3, we implement two design examples in the context of an 8-way CMP with NUCA-based shared cache. We consider a design where the cache is partitioned into few large memory banks (one per core) and also a highly banked architecture which pushes the envelope of the NUCA idea.

In Section 4, we show that these implementations of Nahalal significantly improve L2 cache access times (compared to previously suggested designs for 8-way CMPs with NUCA). Such improvements are exhibited over a range of commercial and scientific benchmarks, as well as a typical transactional memory benchmark. Nahalal improves cache access time performance by up to 41% compared to traditional CMP designs, yielding performance gains of up to 12.65% in run time.

Beyond this particular example, the Nahalal concept of placing public data where it is easily accessible by all sharers may be more broadly applicable, and is expected to benefit performance, reduce power, and improve available bandwidth in various settings. Furthermore, we believe that the trend towards improved platforms' performance/power will drive towards asymmetric architectures, which can use the most appropriate execution (or storage) element for each task [15]. We see Nahalal's approach as part of the overall trend towards asymmetry at the platform level (e.g., asymmetric memory). Some such future research directions are outlined in Section 5.

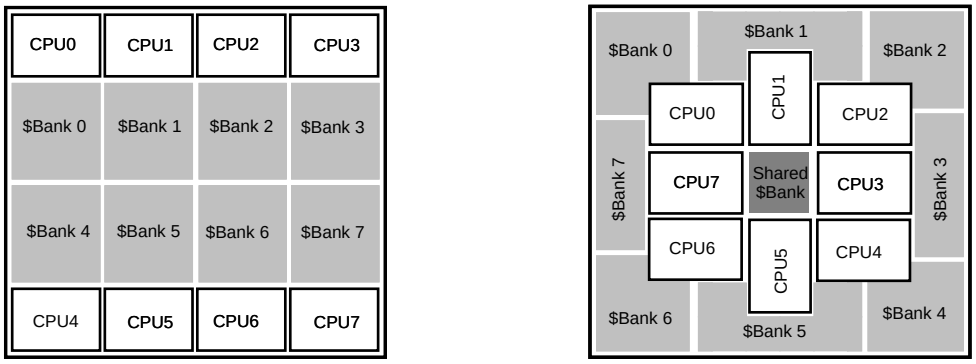
In summary, this paper challenges the ways in which caches are organized. Traditionally, uniprocessor microarchitectures partitioned the cache based on content (e.g., Instructions Cache versus Data Cache in the Harvard architecture) and hierarchically (e.g., cache levels: L1, L2, etc.). We argue that with CMP architectures, additional cache dimensions will prove valuable, for example, based on data sharing, data coherency, and other CMP characteristics. This paper develops the CMP data sharing paradigm.

2. Related Work

Two principal layout alternatives were proposed in recent studies of CMP cache organization: a number of proposals locate a multi-banked L2 cache at the center of the chip, surrounded by all processors [10][11][13][16][17]; several others consider a tile based architecture [18][19][20]. Both of these alternatives are *symmetric*, that is, all cache banks have the same function. In contrast, Nahalal's cache structure is *asymmetric* [12][21], with the bank (or banks) in the center having a different function than other banks.

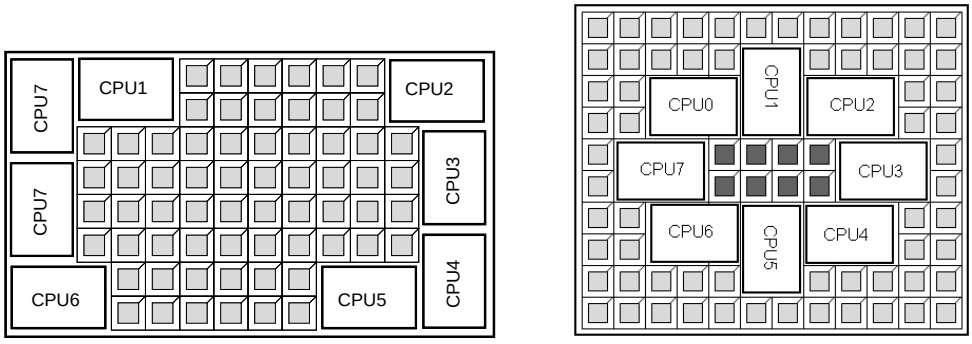
Previously proposed designs were generally either based strictly on shared caches [10][11][17], or strictly on private caches [13][16]. Since each type of cache has inherent drawbacks (as explained above), these works employ various mechanisms to mitigate these limitations. In contrast, Nahalal dedicates part of the cache to shared data, and the rest to private data. Recently, Jin and Cho [22] suggested a hybrid solution based on a tile architecture, where the cache in each tile can be configured as either shared or private by the operating system. Their approach differs from ours in that the designation of caches as public or private occurs at run-time and relies on software support; therefore, their physical (hardware) layout cannot optimize for the intended usage as in Nahalal.

Previous works on CMP cache design have recognized the need for shared L2 caches that follow a Non Uniform Cache Architecture (NUCA), where access times vary according to the distance between the data and the client processor [10][11][13][17][23]. Beckmann and Wood [10] and Huh et al. [11] have studied Dynamic NUCA (DNUCA), which allows data lines to migrate towards processors that access them. Both studies have concluded that access to shared data hinders the



(a) Cache In Middle (CIM) layout. **(b) Nahalal layout.**

Figure 2. Two cache organizations for an 8-way CMP with one L2 memory bank per core.



(a) Cache In Middle (CIM) layout. **(b) Nahalal layout.**

Figure 3. Two cache organizations for an 8-way CMP with a heavily banked (64 banks) L2 cache.

effectiveness of DNUCA, since such data ends up in the middle of the chip, far from all processors. The Nahalal concept of bringing shared data close to all processors can solve this Achilles heel of DNUCA, and may provide a platform where DNUCA can realize its potential [21].

Several works have used line replication to ease the shared data problem [1][17][23]. Such replication, however, reduces the effective cache capacity, further increasing the on-chip capacity pressure. Moreover, line replication is only cost-effective when the shared lines are read-only, since writing entails invalidation of all copies which may impact performance. Nahalal reduces the need for replication by allowing a single copy to reside close to all the processors that share it.

Liu et al. [21] were the first to suggest adding a central cache cell to a highly-banked DNUCA-based CMP, in order to improve access times to shared data. In the brief version of this paper [12] we have conducted an application study and outlined the benefits of using a shared cache bank, in the context of a CMP machine with one L2 bank per core. In this paper, we take a more general approach and study the implementations of the Nahalal concept under both design choices (heavily-banked and one bank per core). We elaborate on aspects of the implementation that were not covered by previous work, such as several alternatives for searching cache lines and the ability to predict line locations. Our evaluation examines the effect of these issues on performance and power. We also study a wider range of applications, including transactional memory.

3. Cache Organization and Management

We now discuss a possible realization of the Nahalal concept in the context of an 8-way CMP.

3.1 Layout

The main concept in Nahalal is placing shared data in a relatively small area in the middle of the chip, surrounded by the processors, and locating private data in the periphery. This solution is feasible thanks to the *shared hot lines* phenomenon described in Section 1 (and in the Appendix), which suggests that a relatively small structure is sufficient for serving the majority of shared data accesses. We implement this concept in two typical 8-way CMP architectures – one with 8 memory banks (one per core), as suggested, e.g., in [13][16], and one 64-bank architecture, pushing the envelope of the NUCA idea, as suggested in [10]. In both cases, we compare Nahalal to a traditional *Cache In the Middle (CIM)* layout.

Figure 2(a) depicts a typical Cache In the Middle (CIM) organization for an 8-way CMP with one L2 bank in the proximity of each core [13]. Figure 2(b) portrays our alternative layout based on the Nahalal concept. In both designs, each core has a private L1 cache, and the L2 cache is banked. The L2 cache capacity is partitioned among the cores so that each core has one cache bank in its proximity (depicted in light grey). In Nahalal, some of the total L2 cache capacity is designated for shared data, and is located in the center (depicted in dark grey). Figure 3(a) shows a suggested layout for the more aggressive DNUCA with

64 banks [10], and Figure 3(b) shows a Nahalal layout for the same number of banks.

3.2 Cache Management

A broad range of potential cache management strategies can be implemented given the layouts depicted in Figure 2 and Figure 3. We now present one possible cache management policy, based on shared L2s, for each of the four layouts (CIM and Nahalal, 8 or 64 banks). We implemented these policies in Simics [24] and experimented with them as reported in the next section. The CIM management policies follow the ones proposed in [10]. Nahalal's management closely follows that of CIM, with the necessary adaptations.

In all layouts, we focus on a *shared cache paradigm*, where all processors can access all banks of the L2 cache. Each address can be located in multiple banks. Thus, cache management needs to decide *where* to place the line when it is first fetched, and subsequently, if and *when* to migrate a line from its current bank, and to *where* (to another bank or to be evicted from the cache). Placement and migration of cache lines are discussed in Section 3.2.1 (for the case of one bank per processor) and Section 3.2.2 (for the heavily-banked case). In addition, a *search* mechanism is required in order to discover cache lines in the banks where they reside. Search is discussed in Section 3.2.3.

3.2.1 Placement and Migration – One Bank per Processor

In both implementations of Figure 2, (CIM and Nahalal), when a line is first fetched, the line is placed in the bank adjacent to the processor that made the request. In the implementation of the 8-bank CIM, the line remains in its initial location as long as it is not evicted from the cache. In contrast, Nahalal uses migration in order to divert *shared hot lines* to the cache bank at the center of the chip. In order to prevent pollution of the center bank with unpopular lines, a line is migrated to the center bank only after N accesses (for some threshold N) from different processors. In our implementation, the threshold is set to 8 accesses, and the policy is implemented by adding a 3-bit counter to each cache line.

The central bank is managed using an LRU eviction policy; that is, when space needs to be made for a new line, the least recently used line (LRU) among all cache lines in the center is evicted. However, the victim line is not evicted from the entire L2 cache. Instead, the locations of the two lines (the one moving to the center and the evicted one) are swapped.

Since saving usage time statistics for all the lines in the center may be costly in terms of hardware complexity, we organize the central area as an 8-way cache structure, tracking LRU statistics over each set. Such a structure is more feasible in terms of hardware complexity, while still keeping the most used shared lines in the center.

3.2.2 Placement and Migration – Heavily Banked DNUCA

In CMPs with many cache banks, data typically migrates among banks at run-time [10][11]. Banks are typically partitioned into groups called *banksets*, so that the members of each bankset are distributed in all areas of the chip. Thus, for a given core, each bankset includes banks residing at various different distances from the core. Every data line is mapped to exactly one bankset according to its address, and may reside in any bank pertaining to the designated bankset. When a processor accesses a cache line that already resides in one of the banks, the line can migrate to another bank that belongs to the same bankset and is located closer to the processor. In our implementations, each bankset is comprised of 16 banks, and thus each line has 16 possible locations, as in a 16-way cache. Similar settings were used in previous studies of heavily banked DNUCA [10].

In the CIM layout of Figure 3(a), we implemented the approach of Beckmann and Wood [10], whereby migration is gradual. When a processor accesses a data line that resides in a remote bank, that data line is not immediately transferred to the vicinity of the requesting processor. Instead, the data line makes a single step towards the processor— the line is moved to the bank closest to its current location among the banks in the same bankset that reside closer to the processor. The line is swapped with one of the lines in the chosen bank, creating a process that resembles a bubble sort.

We follow a similar strategy in the Nahalal design of Figure 3(b), except for the special treatment of shared data. To identify shared data, we use the sharing status vector of the cache coherence mechanism. (Note that although we do not store multiple copies of the same data line in the L2 cache, multiple copies may still reside in the processors' L1 caches. Hence, a cache coherence mechanism is employed for L1 cache management.) When a line is accessed, we first check the sharing status vector; if more than one bit is set, the line is deemed shared, and is migrated to one of the banks in the center of the chip, (unless it is already there). If the line is not shared, it is migrated towards the requesting processor's private area.

3.2.3 Search

All of our implementations are based on the DNUCA paradigm, where a given cache line may reside in multiple banks, and its location in the cache is determined at run-time. This raises the question of how to search whether or not a requested line is in the cache, and if it is, where it is located. We now describe several alternatives for doing so.

The fastest way to search is to send queries to all the relevant banks in parallel (to all banks in the 8-bank case, and to the ones pertaining to the appropriate bankset in the 64-bank case), and have a bank that contains the requested line respond. If no bank responds within an appropriate timeout, it can be concluded that the line is not in the cache. While this parallel search allows lines to be located very quickly, it is also highly inefficient in terms of power dissipation. Furthermore, it burdens the on-chip interconnect with many requests.

In order to conserve energy and reduce the interconnect's load, it is preferable to stagger the search, and thus reduce the number of queries sent in case the line is found early. The extreme version of this approach is a sequential search, in which the requesting processor checks the relevant banks one at a time. The processor checks banks in increasing order of their distances from it, starting from the closest bank. The search continues until either the line is found or all relevant banks have been searched. Sequential search was implemented for CIM layouts in [10][11].

In Nahalal, there are two relevant banks at approximately the same distance from the processor— one local, i.e., in the processor's "private back yard", and one in the center. This raises the question where to begin the search. In some benchmarks more accesses are made to private data, while in others (most notably commercial ones), accesses to shared data exceed those to private

data. (See the Appendix.) In order to reduce the load on the shared central structure, Nahalal's sequential search first checks the requesting processor's closest relevant local bank. If the line is not found there, the processor checks the relevant bank (or banks) in the center of the chip, and then checks all other relevant banks as needed, in order of their distance from it.

In both the CIM and Nahalal layouts, sequential search may take a long time to complete, the worst case occurring when all banks are searched. However, we observe that Nahalal has an advantage in average search time (as well as average power dissipation) thanks to its more predictable placement of shared lines, as we now explain and is confirmed in simulations in the next section. Consider, for the remainder of this section, the 8-bank design (the case of 64 banks is similar; for clarity of the exposition, we focus on one design). When a frequently used private cache line is accessed, either in Nahalal or CIM, it is found by the first search query in the requesting processor's local bank. However, if a frequently used line is not found by the first query, the line is most likely shared, and hence resides elsewhere in the cache. In this case, with the CIM approach, it is equally likely for the line to reside in each of the other seven banks. Therefore, it takes an average of 3.5 additional queries to locate the line. In contrast, in Nahalal, the line is most likely located in the center, and will thus be found by one additional query.

There is a tradeoff between the increased latency of the sequential search and the higher power costs for the parallel search. This predictable placement of shared lines in Nahalal allows us to balance these two considerations. We devised a *hybrid* search approach, whereby two banks are first searched in parallel, namely the processor's local bank and the center bank. If neither query locates the requested line, the search continues sequentially. The hybrid approach sends at most one query more than the sequential search, and a superfluous query is sent only in case the line is in the local cache. In contrast, parallel search may send up to 7 superfluous queries, and sends superfluous queries in almost all cases (except when the entire cache needs to be searched). In terms of performance, the hybrid approach can resolve most searches within one step, since most accesses are made to lines that reside either in the local bank or in the center bank.

Though it provides a good tradeoff between power and performance, the hybrid approach still suffers from two drawbacks. First, although its power dissipation is modest compared to that of parallel search, due to the increasing importance of energy saving in modern architectures, it is undesirable to expend even this modest cost. Second, the hybrid approach queries the central bank for every access, which may create excessive contention among the cores at the center. In order to mitigate both of these shortcomings, we implemented a simple *predictor* at each core, which records the last 1K lines that were fetched from the center. Nahalal's *sequential search with predictor* proceeds as follows. The processor first checks if the accessed line is stored at the predictor. If it is, the line is first searched in the center bank. Otherwise, the line is first searched in the local bank. In both cases, if the first query fails to locate the line, the search continues sequentially. For a modest overhead of 0.56% of the total cache capacity (given the cache sizes simulated in the next section), the sequential search with predictor achieves almost the same performance as the hybrid approach, with the minimal energy cost of the sequential approach.

Finally, for CIM, such a simple predictor is not feasible, because there are more than two plausible locations for each cache line. However, it is possible for each core to track the exact locations of frequently used cache lines. Such an approach has been implemented by Ricci et al. [25] in the context of highly banked DNUCA.

4. Evaluation

In Section 4.1 we present the evaluation environment and parameters. We then proceed to present our results. Nahlal's principal benefit is in reducing the distances between processors and their data. In order to isolate the impact of this phenomenon from secondary artifacts like search time, we first run experiments using parallel search for all layouts (Sections 4.1 to 4.3). This scenario actually favors the CIM design, since the more energy-efficient search approaches work faster with Nahalal than with CIM, as we show in Section 4.4, where we study the effect of the different search mechanisms on both performance and energy. In Section 4.2, we measure cache access delays and the distances between processors and their data - the direct artifacts of the Nahalal layout. In Section 4.3, we examine Nahalal's impact on overall performance, as well as performance trends under increasing wire delays.

4.1 Methodology

To demonstrate the potential performance gain of the Nahalal topology, we implemented the four 8-processor CMP design examples of Figure 2 and Figure 3 in the Simics [24] full-system simulator. Our simulations parameters closely follow those of previous work [10], wherever applicable. The processors are implemented using the x86 in-order processor model. In all designs, each processor has a private 32KB L1 cache, and the processors share a 16MB L2 cache.

In the CIM 8-bank layout of Figure 2(a), each core is adjacent to a 2MB cache bank; in the corresponding Nahalal topology (Figure 2 (b)), each processor has a 1.875MB bank in its "private back yard", and the central bank's capacity is 1MB. In the 64-bank case (Figure 3), each bank's capacity is 256KB. Access times are shorter for small banks (6 cycles) than for large ones (15 cycles). The distances between the smaller banks are also shorter, and hence take a shorter time to traverse (2 cycles versus 5). All system parameters are summarized in Table 1.

Table 1. System parameters

Parameter	Value	
	8-bank	64-banks
L1,L2 line size	64B, 64B	
L1 caches size, ways, access	32KB, 2-way, 3 cycle	
Main memory access	300 cycles	
L2 cache size	16MB	
Bank size	2MB (1.785 in Nahalal)	256KB
L2 bank access time	15 cycles	6 cycles
Link delay between adjacent banks	5 cycles	2 cycles

In order to evaluate Nahalal over a broad range of applications, we study three families of benchmarks. First, we run sample

scientific benchmarks from the Splash-2 [26] and SPEComp [27] kits. Second, we experiment with three commercial workloads (*apache* [28], *zeus* [29], and *SPECjbb2000* [30]). For the web benchmarks, we use the SURGE [28] toolkit to generate a representative workload of web requests from a 30,000-file, 700MB repository with a zero backoff time. This toolkit generates a Zipf distribution of file accesses, which was shown to be typical of real-world web workloads. Finally, we study two representative software transactional memory benchmarks, *HashTable* and *RBTree*, from the RSTM [31] kit; we give an equal probably for insert, delete, and search in each data structure.

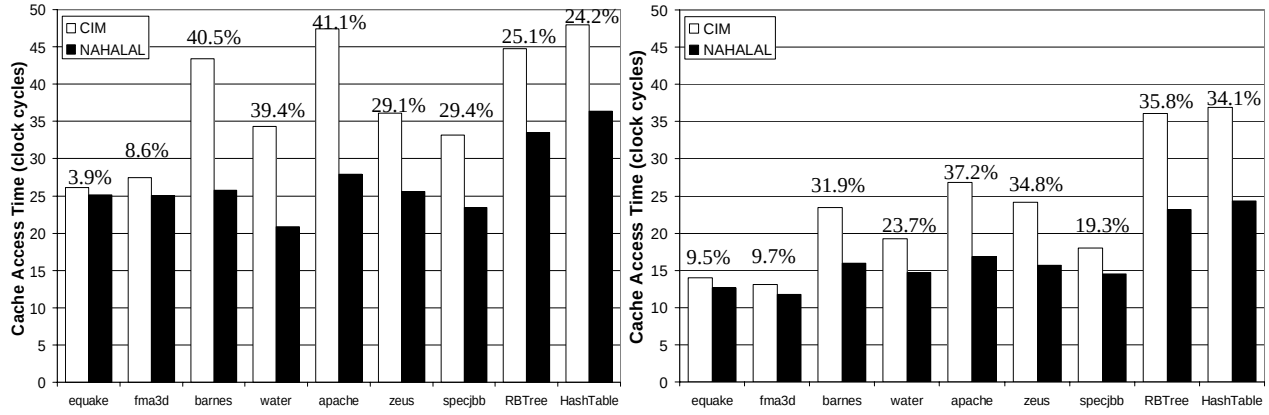
4.2 Cache Delay and Relative Distance

Figure 4 presents average L2 cache access times for Nahalal and CIM in various benchmarks. Nahalal achieves superior results in all benchmarks, regardless of the number of banks. It reduces the average L2 cache access time compared to CIM by an average (over all benchmarks) of 26.8% and 26.2% for the 8-bank and 64-bank cases, respectively. The most significant improvement, of 41.1% for 8 banks (37.2% for 64 banks), is obtained for the *apache* benchmark.

Nahalal's faster average access time stems from faster access to shared data, as shown in Figure 5. In all layouts, most of the private data is located in the local banks of each processor. But while Nahalal is able to serve most of the accesses to shared data from the center of the chip, in CIM layouts, most accesses to shared data go to remote banks, thus suffering long access times. Consequently, Nahalal benefits from short average access time even for benchmarks with many accesses to share data.

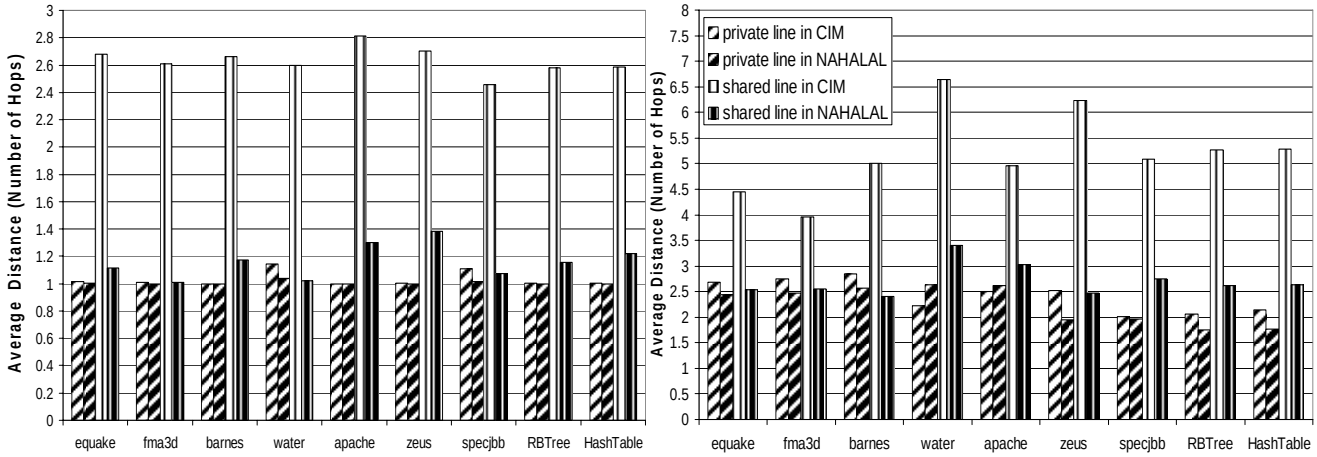
4.3 Overall performance

Nahalal achieves an average speedup of 7% in total execution time compared to CIM. The best speedup is obtained for commercial benchmarks - 9.32% and 12.65% for *zeus* and *apache* respectively for the 8-bank case (13.98% and 10.31% for the 64-bank case). On transactional memory benchmarks, the speedup is 6.42% and 7.42% (for the *RBTree* and *HashTable*, respectively). Nahalal is most advantageous when there are many accesses to shared data in L2. Thus, for benchmarks with a low L2 access rate (e.g., *barnes*), where L2 is not the bottleneck, or for benchmarks with almost no sharing, (e.g., *fma3d*), Nahalal's improvement in overall performance is more modest. In the future, one can expect



(a) 8-bank CMP **(b) 64-bank CMP**

Figure 4. Average L2 cache access times; labels indicate Nahalal's reduction in L2 hit time compared to CIM.



(a) 8-bank CMP **(b) 64-bank CMP**

Figure 5. Breakdown of average distance to shared and private lines for CIM and Nahalal; distance is measured in hops, (which is the number of intermediate banks on the path to the data).

CMP applications to have larger memory demands and exhibit more sharing, while growing wire delays will increase the importance of locality of reference. Hence, Nahalal's benefit will become more significant.

Figure 6 projects the importance of Nahalal in future architectures, where wire delays will become more dominant. The figure demonstrates the effect of increasing wire delays on performance for the *apache* benchmark (in an 64-bank design). We consider a reference system where each line's location is determined statically, according to its address (this is called *Static NUCA*, or *SNUCA)*. We depict the speedup in runtime of both CIM and Nahalal over the reference system as the per-hop link delay increases. Although both systems exhibit more speedup as the wire delay increases, the relative gain of Nahalal grows more as technology scales. This is because distance-related delay becomes dominant, and Nahalal is more effective in reducing the average access distances. Overall, the Nahalal solution is more scalable as wire delays become dominant.

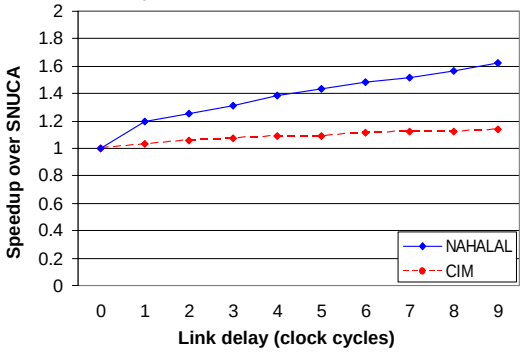


Figure 6. Runtime speedup over static line placement for both CIM and Nahalal for increasing link delays (in clock cycles). The results are given for the apache benchmark, in the 64-bank cache design. Nahalal's performance gain increases as the wire delay becomes more dominant.

While the number of cores that can reside in physical proximity to a shared cache is limited, we note that this limitation is even more severe in the CIM layout, where shared data ends up in the middle, and its distance from all cores grows as the number of cores increases. Moreover, given that Nahalal features a single designated cache for hot shared data, one can effectively mitigate this limitation by investing more resources in the designated shared cache, e.g., by laying out direct fast wires from all cores to the designated cache, by using stronger drivers, and by implementing multiple ports.

4.4 The effect of search

We now turn to consider more energy-efficient search mechanisms. Our baseline for comparison is parallel search in CIM layout (denoted *CIM_par)*. Figure 7 compares the average search times of parallel and sequential search in Nahalal (*Nahalal_par* and *Nahalal_seq*, resp.), and sequential search in CIM (*CIM_seq*), normalized to the search time of *CIM_par*. Nahalal_par is faster than CIM_par because closer data is located faster, and as we saw above, Nahalal's average distance to data is shorter. In both cases, sequential search is more energy efficient than parallel search, but also slower— by 44.5% and 37% for CIM and Nahalal, respectively. The penalty for sequential search is smaller in Nahalal thanks to the more predictable location of frequently accessed shared data, as explained in Section 3.2.3. Thanks to the closer location to data along with the smaller penalty for using sequential search, Nahalal_seq's average search time is only between 1% (for *equake)* and 40% (for *zeus)* longer than that of CIM_par. In contrast, CIM_seq can be up to 129% slower than CIM_par (e.g., *barnes)*.

Next, we consider two search schemes that balance the tradeoff between power and latency: (1) *Nahalal_hybrid*, which first searches the local and central banks in parallel, and then proceeds with sequential search; and (2) *Nahalal_seq_predictor*, sequential search augmented with a predictor in order to decide which of the two nearest banks to search first (local or central). The normalized average search times of these two for various benchmarks appear in Figure 8. We observe that Nahalal_hybrid's search time is only 13% slower than that of Nahalal_par, on average. Finally, Nahala_seq_preditor, which consumes even less energy than Nahalal_seq (because it searches in the correct place in the first step more often), is nearly as fast as Nahalal_hybrid (only 10% slower on average). Recall that our predictor merely saves the last 1K lines that were served from the center bank, and its overhead is thus negligible.

Finally, we study the efficiency of each scheme using a combined power and performance metric. More specifically, we multiply the number of search queries sent in each fetch by that fetch's search time. The product, denoted *delay·search transactions, is* low when the search is most efficient. The results for Nahalal's four search schemes are presented in Figure 9; the

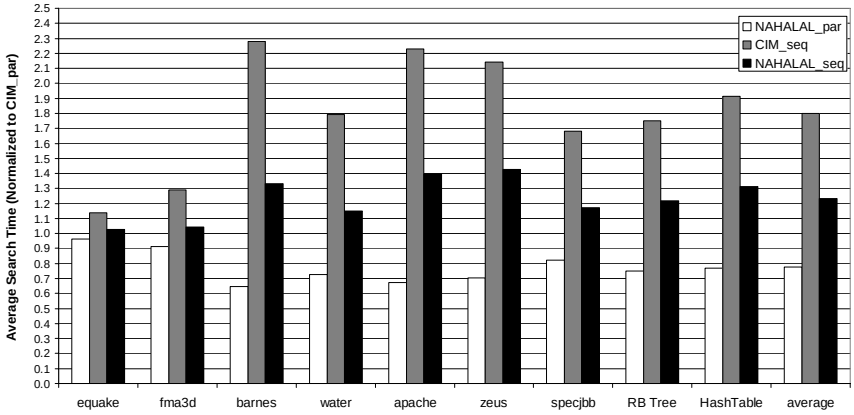


Figure 7. Average search time normalized to CIM_par search.

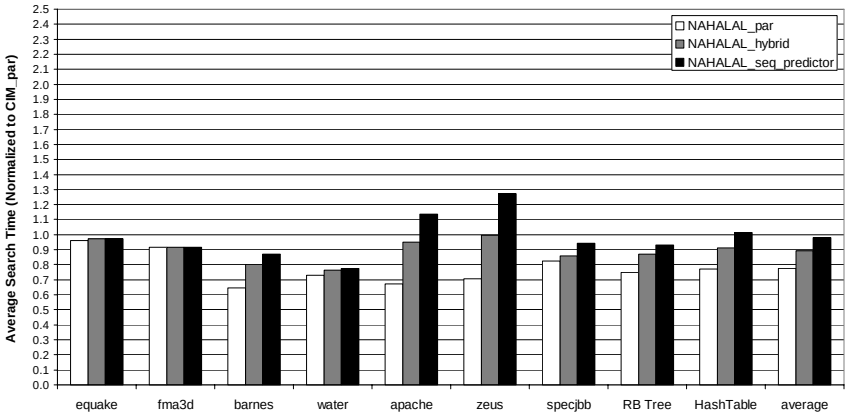


Figure 8. Average search time normalized to CIM_par of Nahalal's parallel search, hybrid search, and sequential search with a predictor.

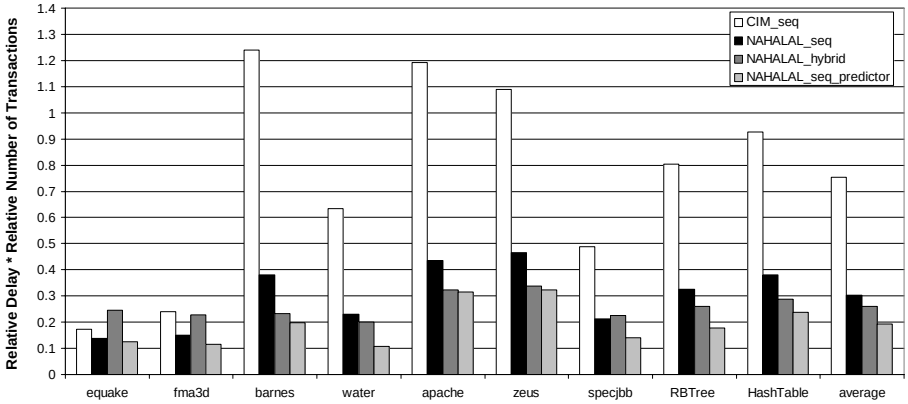


Figure 9. Delay·Number-of-search-transactions product for Nahalal's sequential, hybrid, and sequential with predictor search schemes, relative to CIM_par.

results are again normalized to *CIM_par*. As expected, sequential search (CIM_seq and Nahalal_seq) performs well when most of the data is private, since it hits the right bank in the first step (e.g., in *equake* and *fma3d*). When the percentage of accesses to shared data increases, CIM_seq degrades drastically, since it requires an average of 4.5 searches to find shared data. Nahalal_seq also degrades, but to a much smaller extent, since shared data is typically found within 2 steps. It remains superior to CIM_par in all benchmarks. Nahalal_hybrid better balances the energy versus latency tradeoff, when shared data is involved. Finally, Nahalal_seq_predictor exhibits the best results over all benchmarks, outperforming the hybrid search by 25.7% on average. We conclude that the (modest) 1K overhead needed for Nahalal_seq_predictor is well worth its benefits; this predictor-based sequential search is the most appropriate scheme for future CMPs, which need to take into consideration both performance and power costs.

5. Summary and Future Directions

The on-chip memory system is becoming a performance bottleneck in chip multiprocessors. Therefore, cache architectures should be specifically adapted and optimized to the emerging multi-processing environment. In this paper, we proposed partitioning the shared on-chip cache according to the level of data sharing. This approach is motivated by the observation that, in many multithreaded applications, a small set of shared cache lines accounts for a significant portion of the memory accesses.

We have leveraged the shared hot-lines phenomenon to devise Nahalal - a new CMP topology that locates shared data close to all sharers and still preserves vicinity of private data for all processors. We have demonstrated the potential of the Nahalal concept via two CMP design examples, a shared cache with one L2 bank per core, and a heavily-banked shared cache leveraging NUCA. In both cases, Nahalal greatly improves average cache access times, with savings up to 41.1%. We have considered several searching schemes to locate a line within the cache and have shown that the Nahalal design allows for more efficient search in terms of power and performance than traditional CMP layouts. Moreover, we presented a predictor-based search scheme that achieves high performance at a very low power cost.

While this paper presents a specific optimization of CMP cache architectures, namely optimizing shared data access, it fits within the broader picture of re-thinking traditional designs, for example, by breaking symmetry. Nahalal's asymmetric treatment of cache memory can be seen as but one thread in the overall trend towards architecture asymmetry in CMPs, which has been previously considered in the context of making the *processors* asymmetric [15][32].

Table 2. Cache line sharing characteristics

Sample Benchmarks		Shared lines		Modified shared lines	
		Percentage out of all accesses	Percentage out of all lines	Percentage out of all accesses	Percentage out of all lines
SPEComp	equake	32.05	0.73	2.78	0.40
	fma3d	8.93	0.16	0.37	0.14
Splash2	barnes	15.36	7.07	3.14	0.61
	water	24.90	11.96	17.55	10.85
Commercial	apache	58.25	34.33	47.91	25.26
	zeus	56.85	37.76	41.64	28.16
	specjbb	44.24	13.78	15.39	0.89
RSTM	RBTree	88.71	53.85	83.46	50.11
	HashTable	86.49	58.48	81.13	54.52

We have used asymmetry explicitly in partitioning the cache in a novel way - based on data sharing characteristics. Nevertheless, handling shared data is only one of numerous challenges unique to CMPs. Future designs may well benefit from asymmetry in additional aspects. Some examples for future study may include partitioning caches according to required coherence levels, transactional versus non-transactional memory, special-purpose areas for semaphores, etc. Overall architectural solutions will need to account not only for memory access, but also for cache affinity, interrupt handling, and interconnects.

The Nahalal concept may well prove useful beyond the realm of L2 memory, at the system level. For example, at a system-level, the Nahalal idea may be manifested by deploying some of the on-die memory close to processing elements such as a DSP, a decoder, and a network processor, and remotely from the main (general purpose) cores. Such a “shared yard” for special purpose processing elements may greatly reduce the overhead of the general purpose cores’ L2 memory and its access port by eliminating the need to copy data back and forth to the main L2. These directions are all subjects for future research.

Appendix - Memory Access Characterization

In this appendix we study memory access characterization of multithread workloads. We have reported these results in [12], and they are presented here for the sake of completeness with the addition of transactional memory benchmarks that were not present in [12].

We characterize memory access patterns occurring in multithreaded workloads for a CMP of 8 nodes. We run three types of benchmarks - scientific programs from the SPEComp and Splash2 kits, commercial benchmarks, and software transactional memory (RTSM). For further details regarding the benchmarks, see Section 4.1. We use the Pin program analysis tool [33] to profile accesses to each cache line of 64 bytes (either L1 or L2), and consider a line to be shared if multiple processors access it within a window of 10 million instructions. The profiling results, summarized in Table 2 lead to several observations.

First, in many workloads, accesses to shared data comprise a substantial fraction of the total memory accesses (e.g., up to 58.25% in the *apache* workload and a whopping 88.71% in the *RBTree* benchmark). Moreover, in commercial workloads, many of these accesses involve shared lines that are modified by at least one of the sharers (e.g., in *apache*, 82% of the memory accesses to shared data are to modified shared lines). This phenomenon is also true for the transactional memory benchmark, in which more than 90% of the accesses to shared data are to modified shared lines.

Second, there is a clear discrepancy between the number of accesses to shared lines and the number of shared lines in the working set: a small number of cache lines, shared by many processors, accounts for a significant fraction of the total accesses to memory [10][13]. We dub this phenomenon the *shared hot lines* effect. Furthermore, we observe that a small number of shared lines - some very hot lines- are more popular than others, accounting for most of the accesses to shared data. This phenomenon is shown in Table 3. Each column shows, for a given cache size, what percentage of the accesses to shared data are to a working set of this size. As can be seen from the table, a small fraction of the lines is responsible for the majority of the accesses. For example, in *equake*, the most popular 1MB of shared data accounts for 97.59% of the accesses to shared data.

Table 3. Access distribution of shared cache lines

Sample Benchmarks		% Out of all access to shared data			
		0.5MB	1MB	1.5MB	2MB
SPEComp	equake	96.87	97.59	98.23	98.82
	Fma3d	99.89	99.92	99.59	99.97
Splash2	barnes	93.44	96.67	98.83	99.50
	water	100	100	100	100
Commercial	apache	80.37	89.79	94.08	96.67
	Zeus	84.38	90.75	83.82	95.99
	specjbb	99.98	100	100	100
RSTM	RBTree	98.21	98.49	98.69	98.87
	HashTable	97.95	98.27	98.50	98.69

We have also found that typically the same data is shared throughout the program's lifetime; and that shared data is typically shared by many processors. Together, our observations indicate that reducing the access latency to a modest number of shared data lines can have a significant impact on CMP performance; and that a small subset of the memory capacity suffices for holding the shared lines to which the majority of accesses are made.

References

[1] R. Ho, K. Mai, and M. Horowitz, “The future of wires,” Proceedings of IEEE,89(4), April 2001.

[2] L. Hammond, B. A. Nayfeh, and K. Olukotun. “A Single-Chip Multiprocessor”. IEEE Computer, September 1997

[3] V. Agarwal, M. S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate vs. IPC: The end of the road for conventional microprocessors. ISCA-27, June 2000

[4] WJ Dally and S. Lacy. VLSI Architecture: Past, Present, and Future, In Proceedings of the Advanced Research in VLSI conference, Jan. 1999, pp. 232--241.

[5] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” In ASPLOS X, pages 211–222, Oct. 2002

[6] S. Gochman, A. Mendelson, A. Naveh, A, and E. Rotem, "Introduction to Intel® Core™ Duo Processor Architecture," Intel Technology Journal, Volume 10, Issue 02. May 2006.

[7] AMD white paper, “Key Architectural Features AMD Athlon™ 64 X2 Dual-Core and AMD Athlon™ X2 Dual-Core Processors,” http://www.amd.com/gb-uk/Processors/ProductInformation/0,,30_118_9485_13041%5E13043,00.html

[8] AMD technical articles, “Barcelona's Innovative Architecture Is Driven by a New Shared Cache,” http://developer.amd.com/article_print.jsp?id=173

[9] W.A. Wulf and S.A. McKee, “Hitting the Memory Wall: Implications of the Obvious,” Computer Architecture News, vol. 23, no. 1, pp. 14-24, Mar. 1995

[10] B. M. Beckmann and D. A. Wood, “Managing wire delay in large chip multiprocessor caches,” MICRO 37, Dec. 2004

[11] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S.W. Keckler, “A substrate for Flexible CMP Cache Sharing,” ICS 05, June, 2005

[12] Z. Guz, I. Keidar, A. Kolodny, U. C. Weiser, "Nahalal: Cache Organization for Chip Multiprocessors", IEEE Computer Architecture Letters, vol. 6, no. 1, May 2007

[13] B. M. Beckmann, M. R. Marty, and D. A. Wood, “ASR: Adaptive Selective Replication for CMP Caches,” MICRO 39, December 2006

[14] E. Howard, “Garden Cities of To-Morrow,” London: Swan Sonnenschein & Co. Ltd, 1902

[15] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé, “Performance, Power Efficiency, and Scalability of Asymmetric Cluster Chip Multiprocessors,” In Computer Architecture Letters, Volume 4, July 2005.

[16] J. Chang and G. S. Sohi. “Cooperative Caching for Chip Multiprocessors,” ISCA-33, June 2006

[17] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, “Optimizing Replication, Communication, and Capacity Allocation in CMPs,” ISCA32, 2005.

[18] J. Brown, R. Kumar, and D. Tullsen. "Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures", 19th ACM Symposium on Parallelism in Algorithms and Architectures , SPAA, San Diego, June 2007

[19] L. Jin and S. Cho, “Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring”, in Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2007.

[20] M. R. Marty and M. D. Hill ,“Virtual Hierarchies to Support Server Consolidation”, ISCA-34, June 2007.

[21] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, “Enhancing L2 organization for CMPs with a center cell,” IPDPS'06, April 2006.

[22] L. Jin, and S. Cho, “Better than the two: Exceeding private and shared caches via two-dimensional page coloring,” in Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2007.

[23] M. Zhang and K. Asanovic, “Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors” ISCA32, 2005.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, and G. Hallberg, “Simics: A full system simulation platform,” IEEE Computer, 35(2):50–58, Feb. 2002.

[25] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian, “Leveraging Bloom Filters for Smart Search Within NUCA Caches”, 7th Workshop on Complexity-Effective Design (WCED), June 2006.

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. ISCA-22, June 1995.

[27] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In Workshop on OpenMP Applications and Tools, pages 1–10, July 2001.

[28] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In Measurement and Modeling of Computer Systems, pages 151–160, June 1998.

[29] http://www.zeus.com/products/zws/

[30] http://www.spec.org/jbb2000/

[31] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the Overhead of Nonblocking Software Transactional Memory," TRANSACT 2006

[32] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. "Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance." ISCA-31, June 2004.

[33] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,” PLDI 2005.