A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors

Lei Jin   Hyunjin Lee   Sangyeun Cho

Department of Computer Science
University of Pittsburgh
{jinlei,abraham,cho}@cs.pitt.edu

ABSTRACT

This paper proposes and studies a distributed L2 cache management approach through page-level data to cache slice mapping in a future processor chip comprising many cores. L2 cache management is a crucial multicore processor design aspect to overcome non-uniform cache access latency for high program performance and to reduce on-chip network traffic and related power consumption. Unlike previously studied "pure" hardware-based private and shared cache designs, the proposed OS-microarchitecture approach allows mimicking a wide spectrum of L2 caching policies without complex hardware support. Moreover, processors and cache slices can be isolated from each other without hardware modifications, resulting in improved chip reliability characteristics. We discuss the key design issues and implementation strategies of the proposed approach, and present an experimental result showing the promise of it.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory, Allocation/deallocation strategies*

General Terms

Design, Management, Performance

Keywords

Non-uniform cache architecture (NUCA), page allocation

1. INTRODUCTION

Multicore processors have emerged as the mainstream computing platform in major market segments, including PC, server, and embedded domains. Processors with two to eight cores are commercially available now [9, 16, 21]. Moreover, projections suggest that future processors may carry many more cores–10's or even 100's of cores within a single chip [3]. This trend is accelerated by the unprecedented technology advances and the limited single core scalability [4].

In a future multicore processor, the ever widening processor-memory speed gap as well as the severely limited chip bandwidth exacerbates the dependence of program performance on the on-chip memory hierarchy design and management [7]. The desire to keep more data on chip will lead to a large L2 cache comprising many *banks* or *slices*, likely distributed over the chip space [14]. Unfortunately, the wire delay dominance in nanometer-scale chip implementations and the distributed nature of the L2 cache organization result in *non-uniform cache access latencies*, making the L2 cache design and management a challenging task [5, 6, 8, 17, 26].

So far, L2 cache research and development efforts have been based on the two baseline designs: *private cache* and *shared cache*. A private cache is associated with a specific processor core and replicates data freely as the core accesses them. This automatic *data attraction* allows each processor core to access data quickly, leading to a low average L2 hit latency. However, the limited per-core caching space provided by a single private L2 cache often incurs many capacity misses, resulting in expensive off-chip memory accesses. On the other hand, shared caches form a single logical cache by having each cache slice accept only an exclusive subset of all memory blocks. Overall, a shared cache design will result in better utilization of on-chip caching capacity than a private design because accesses are finely distributed over a large caching space. Unfortunately, the average L2 cache hit latency will be longer than that of a private cache since the cache slice keeping critical data may be far off.

Previous works have shown that neither a pure private design, nor a pure shared design, achieves optimal performance under different workloads [7, 8, 12, 18, 26]. For example, a program with its working set entirely fit into a cache slice will perform better with private caching, while large applications with a high degree of data sharing may perform better on shared caches. Therefore, researchers have further examined optimizations such as cache block migration [14] and data replication [26] in the context of a shared design to improve data proximity. Alternatively, private caches can be optimized to provide more capacity by limiting the degree of data replication or reserving a portion of cache space for other cores to use [5, 6].

In this work, we investigate a dynamic data to cache slice mapping approach at the *memory page level* in the context of a shared L2 cache organization in future many-core processors, similar to Figure 1. Unlike previous researches, the primary focus of this paper is on achieving *flexibility* of using available L2 cache slices. For example, our approach, using a conventional shared cache hardware, allows a cache slice to be shared by a dynamically controllable number of processor cores. Similarly, a cluster of cache slices can be used by a group of processor cores or by a single processor core. Furthermore, cache slices can be excluded from use. The ability to isolate some cache slices is especially important for achieving low power consumption and higher error resilience. For instance, caches that malfunction due to aging phenomena [23] can be pulled out without affecting the overall chip operation. As future processors will incorporate many, possibly 100's of processor cores and cache slices [3], flexible L2 cache management will become extremely important for achieving high performance, low power, and resilience to hardware faults.

The key to achieving high flexibility in the L2 cache management lies in how data items (*i.e.*, memory blocks) are mapped to a cache slice and how efficiently the mapping information is created and maintained. In our proposed approach, *OS performs this mapping at the memory page granularity when it allocates a physical page to a virtual page.* In one implementation, OS can attach a cache slice number to each page table entry. The assigned cache slice number, also stored in TLB (Translation Look-aside Buffer), is used to direct an L2 cache access request on an L1 cache miss. Since mapping decisions are made by software, a very high degree of flexibility is achievable. Moreover, run-time information available to OS, such as scheduled process locations and page allocation status, can be used to direct performance- and priority-aware mappings. Indeed, our preliminary results suggest that the proposed approach can achieve higher performance than (pure) private and shared designs.

The rest of this paper is organized as follows. Section 2 gives a discussion on previous related works. The proposed page-level data mapping approach is described in Section 3. Section 4 presents preliminary experimental results, comparing two hardware-based L2 cache management schemes and a scheme based on our approach. Finally, conclusions and future works will be given in Section 5.

2. RELATED WORKS

This section discusses previous related works on L2 cache management in multicore processors. Among many cache design and behavioral aspects, we focus on *how memory blocks are mapped to a cache slice.*

2.1 Private cache scheme

In a private cache scheme, data mapping is not explicit; all the memory blocks accessed by a processor, regardless of their address, are copied to the local cache slice [5, 18]. Therefore, the set of memory blocks residing in a cache slice at a certain point of time is dictated by the programs that have been running.

Since the behavior and interplay of running programs are neither directly controlled nor easily predictable by the underlying hardware or software, memory blocks in different L2 cache slices should be tracked and book-kept by a (hardware-based) coherence mechanism. In Chang and Sohi [5], for example, a centralized directory is provided to record who (*i.e.*, processor cores) has a copy of a cache block. When another processor core wants to modify the same cache block

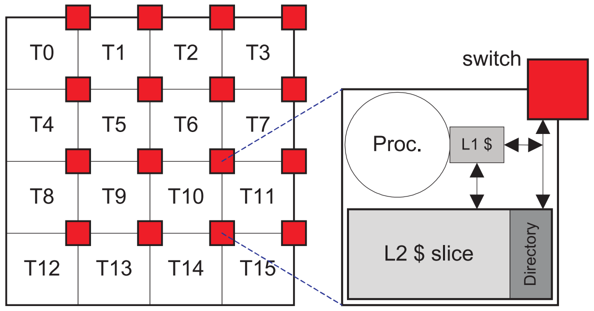

Figure 1: An example 16-tile multicore processor chip and its node.

later, it must request an exclusive, up-to-date copy of the block to the directory; it will then forward the request to the set of cores that have cached the block, so that they can provide the cache block and invalidate their local copy.

There are two important disadvantages in the private cache scheme, compared with the shared cache scheme. First, the limited caching space provided by a single private cache can result in a high on-chip miss rate and accordingly more frequent off-chip accesses [26]. Unbalanced use of caching space is also undesirable. Second, the overhead of enforcing coherence, in terms of hardware resources and run-time coherence actions, is higher [19].

2.2 Shared cache scheme

In a shared cache design, data to cache slice mapping is explicit. That is, given the address of a datum, its location (*i.e.*, the home cache slice) is directly determined. In recent commercial products, this mapping of data to cache slice is done at the *cache line granularity* in a round-robin fashion [16, 21]. This mapping method improves the L2 cache bandwidth by distributing temporally close loads and stores to multiple cache slices.

To describe a shared cache design more formally, a mapping function $f(\cdot)$ is defined on the cache line address which produces the home slice number. The commonly used mapping function is a modulo-N function, where N is the number of available cache slices [16,21,26]. In a future tile-based processor, as sketched in Figure 1 [3, 26], this fine-grained mapping method will likely lead to balanced utilization of overall caching space. This mapping is however very rigid; memory blocks are mapped to cache slices regardless of the distance to the processor accessing them, leading to a longer average L2 cache hit latency. Moreover, losing a cache slice leaves $1/N$ of the total memory space *uncacheable*, resulting in large performance degradation.

On the other hand, enforcing coherence in shared caches is more straightforward than in private caches, because there is no replication of data and the location of a cache line is uniquely determined. Nayfeh *et al.* [19] showed that much of the coherence-related traffic and the overhead of coherence enforcement hardware can be eliminated by sharing cache memory among the processor cores.

2.3 Shared/private hybrid schemes

Speight *et al.* [22] presented an eight-core chip where each two cores form a processor cluster. Within a cluster, four cache banks are shared, meaning that a cache line can be

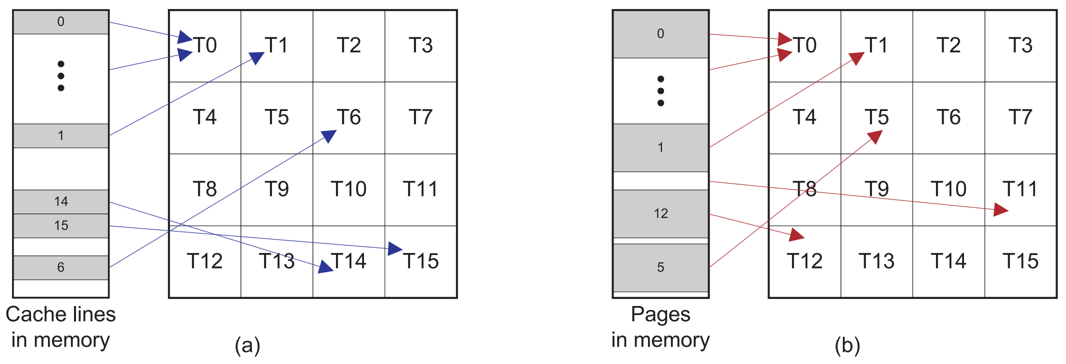

Figure 2: (a) Memory to cache mapping at the cache line granularity. (b) Mapping at the page granularity.

found only in one of the four cache banks. The same cache line, however, can be found in other clusters. Therefore, each cache cluster can be considered a private cache from the viewpoint of the processor clusters and the chip. Coherence should be maintained among the four clusters.

Zhang and Asanović [26] proposed *victim replication* in a shared L2 cache organization. In their design, L2 cache slices can receive a replaced cache line from their local L1 caches as well as their designated cache lines. Essentially, the local L2 cache slice provides a large victim caching space for the cache lines whose home is remote, similar to [15]. A downside of this approach happens on a coherence request; both L1 cache and L2 cache (in parallel or in sequence) should be checked, because it is not readily known if a (remote) cache block has been copied in the local L2 cache slice.

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 (to save capacity against migratory clean copies), in-situ communication (to eliminate coherence overhead for write-read access patterns), and capacity stealing (to better utilize on-chip capacity). Compared with a shared cache organization, however, CMP-NuRAPID requires a more complex coherence and cache management hardware. For example, it implements a distributed directory mechanism by maintaining forward and reverse pointers between the private tag arrays and the shared data arrays.

Lastly, Chang and Sohi [5] proposed a *cooperative caching* framework based on a private cache design with a centralized directory scheme. They studied several optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement, and global replacement of inactive data. Experimental results show that the proposed optimizations effectively limit cache block replication and thus result in a higher on-chip cache hit rate. However, the optimizations come at the expense of a more complex central directory than that of a baseline private cache design.

2.4 Other flexible mapping schemes

Liu *et al.* [17] proposed an L2 cache organization where a different number of cache banks can be dynamically allocated to processors, connected through a shared bus. In their scheme,the OS manages a hardware table which records the mapping between the processors and the available cache banks. When an L1 cache miss occurs, this table is looked up, and a request is generated with the proper cache banks selected for access. Despite its flexibility, the proposal has several drawbacks: (1) their overall design is tuned for a shared bus interconnect and the achievable L2 cache bandwidth is severely limited thereof; (2) the coarse mapping granularity (*i.e.*, cache bank) can result in unbalanced cache capacity usage; and (3) since their memory address to cache bank mapping is not deterministic (similar to a private cache scheme), when there is an L2 cache miss, a request should be broadcast to all the other cache banks to find data and to enforce coherence. The result is a large amount of L2 cache traffic and cache access intervention.

Huh *et al.* [8] studied a processor to cache bank mapping scheme where cache banks can be configured as shared caches, private caches, or a mixture of both. The mapping information, maintained by the OS, affects the behavior of the L1 cache controllers, the bank controllers, and the central on-chip directory. To make this scheme fully configurable, the *sharing status vectors* both in the L2 tags and the central directory should have enough bits to represent all processors, requiring much hardware resources devoted to cache management.

There are two fundamental differences between the above works and the approach proposed in this paper. First, both the previous works maintain mappings between cache banks and processor nodes; the mapped cache banks become private to processor(s). In our approach, mappings are created between a memory page and a cache slice. Therefore, more fine-grained control of cache space utilization and performance tuning is possible. Second, our approach does not require a large amount of hardware resources spent on enforcing cache coherence and data mapping. A much simpler organization equivalent to that of a shared cache will suffice. By comparison, the above works require either a large central directory [8] or a broadcast-based coherence protocol [17].

3. PROPOSED APPROACH

There are clear advantages of mapping memory data to cache slices at the memory page granularity rather than at the cache block granularity [16, 21, 26]. Figure 2 illustrates the difference between the two choices. First, it fits naturally in the current memory management practice of a modern OS like UNIX [2]. Only after a page is allocated does a processor start to see and access cache lines in the page. Therefore, the page allocation event in OS is an ideal trigger-

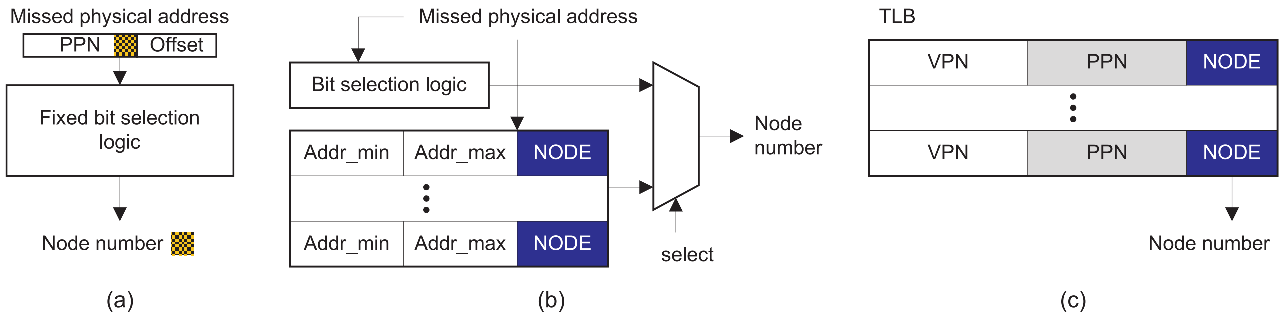

Figure 3: Node selection strategies. (a) Simple, fixed address bit selection method; (b) Region-based, programmable hardware table method; (c) TLB (page table)-based method.

ing point to decide where (*i.e.*, which cache slice) the page should be mapped to. Second, the hardware cost of necessary architectural support is none or minimal. For example, the TLB (Translation Look-aside Buffer) can be extended with a field to keep the cache slice number, copied from a page table entry. The necessary number of bits per entry is $\log_2 N$ where N is the number of cache slices. Lastly, the memory access patterns seen by a monolithic cache is preserved to a large extent. For instance, sequential accesses to consecutive cache lines (within a page) will be directed to the same cache slice, instead of being dispersed to multiple slices. This property allows previous memory optimization techniques, such as hardware prefetching or OS page coloring [13], to be easily adapted to a multicore processor.

3.1 Architectural support

In this subsection, we present three architectural techniques to efficiently support page-level memory data to cache slice mapping. They differ in their implementation cost and achievable flexibility. Figure 3 depicts the techniques.

The first technique ("Selection") is to derive the cache slice number from the address bits directly. In an N-slice system, the $\log_2 N$ low-order bits from the physical page number (PPN) of an address are selected (*i.e.*, mod-N on PPN). By comparison, current-generation multicore processors use the low-order bits from a cache line address [16, 21, 26]. In essence, this technique partitions the physical memory into N equal-sized partitions, each of which is called a *congruence group*. Memory blocks in a congruence group are all together mapped to the same L2 cache slice. By allocating a physical page belonging to a particular congruence group, the OS can control where a virtual page will be placed. In terms of hardware organization, this technique is simplest; it does not require additional logic to compute the cache slice number if N is a power of two. This scheme is less flexible than the other two schemes however in that (1) memory pages are tied to a particular cache slice and the number of pages per cache slices is fixed; (2) therefore, the OS is unable to allocate more pages to a cache slice if the corresponding congruence group does not have enough free pages; and (3) the OS may exclude certain cache slices from normal use, but only at the expense of unusable physical memory space.

The second technique ("Region"), shown in Figure 3(b), provides a *region mapping table* comprising a set of registers to hold two addresses specifying the beginning and end of a region. Each region is mapped to a specific cache slice (node); depending on the number of entries in the mapping table, several regions may be assigned a same cache slice number. In essence, a congruence group is composed of regions, mapped to the same node. This scheme is more flexible than the first technique, in the sense that physical pages are not tied to specific cache slices. In the first technique, physical pages are first mapped to cache slices, and based on the mapping the OS tries to allocate physical pages to virtual pages. When the second scheme is used, the OS first partitions the physical memory space and assigns each partition to a cache slice. Subsequent page allocations will take advantage of this mapping information. Note that it is more straightforward to exclude certain pages using this scheme. Moreover, one can easily integrate the simple bit selection scheme within the framework of the second scheme, as shown in Figure 3(b). One can choose between the two methods, or use them together by defining priorities between the two methods (*e.g.*, override the bit selection result if a mapping is found in the table).

The third and most flexible technique ("PTable") is to assign a slice number to each individual (virtual or physical) page. The mapping information is stored in the per-process page table, as well as in the TLB. Since the TLB is looked up on each memory access, the mapping information becomes available immediately, to be used on an L1 cache miss. In this scheme, the notion of congruence group is not valid any more, because each page allocation is done independently of previous allocations, and the per-page mapping information is individually and explicitly managed.

3.2 OS design issues

3.2.1 Data mapping through page allocation

The OS manages a *free list* to keep track of available pages in physical memory. Whenever a running process needs new memory space, the OS allocates free pages from this list by creating a virtual-to-physical page mapping and deleting the allocated page from the free list. Previously allocated pages are reclaimed by backing up their content to a non-volatile storage if needed, and putting them back to the free list.

If there is no sharing of data between programs running on different cores (*e.g.*, under a multiprogrammed workload), the OS will have the full flexibility of using L2 cache slices as a private cache, a shared cache, or a hybrid of the two, by mapping virtual pages to a specific core, to anywhere, or to a specific group of cores, respectively. For a more formal

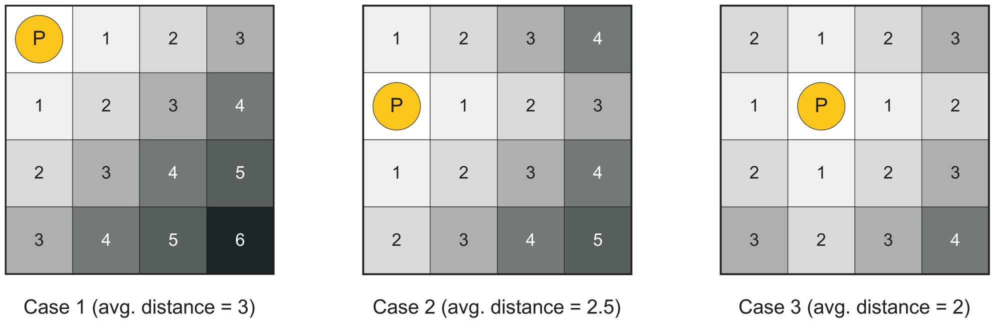

Figure 4: *Program-data proximity*: program/data locations determine the minimum distance to bridge them.

discussion, we define the *congruence group* CG_i ($0 < i < N-1$) given N processor cores and a physical page to core mapping function *pmap*:

$$CG_i = \{\text{physical page (PPN} = j) | pmap(j) = i\}$$

In other words, *pmap* defines a partition ($\{CG_i\}$) of all available physical pages in the main memory so that each partition CG_i maps to a unique processor core i. Note that the SELECTION and REGION techniques implement a *pmap* function. Given *pmap*, the home core for a physical page (and accordingly the cache lines in it) is uniquely defined. Now, the following virtual to physical page allocation strategies achieve different caching schemes:

Private caching. For a page requested by P_i, a program running on core i, allocate a free page from CG_i.

Shared caching. For a requested page, allocate a free page from all the congruence groups $\{CG_i\}$ ($0 < i < N-1$). One can use a *random selection strategy* or a *round-robin strategy* to pick up a free page from a congruence group with available free pages.

Hybrid shared/private caching. First, partition $\{CG_i\}$ into K groups ($K < N$). Then define a mapping from a processor core to a group. For a page requested by P_i, allocate a free page from the group that core i maps to. Here, each group defines the cores that share their L2 cache slices. Again, one can use a random or round-robin strategy within the selected group. Alternatively, one can consider the location of the core that requested a memory page when deciding which free page to allocate.

Although the above descriptions assume the existence of congruence groups (*i.e.*, the SELECTION/REGION schemes), it is straightforward to adapt the strategies to the PTABLE scheme. For example, the necessary action to implement a private cache scheme is to simply record i in the page table and the TLB entry assigned to the requested page.

3.2.2 *Data proximity and page spreading*

To obtain good program performance, it is important to keep program's critical data set close to the processor core on which the program runs [14]. We showed in the previous subsection that the OS has a full control over where (among processor tiles) cache lines will be placed if memory to L2 cache slice mapping is done at the memory page granularity.

In an ideal situation where L2 cache slices are larger than program working sets, the OS can allocate new pages to requester cores; each L2 cache slice then becomes a private cache, ensuring fast data access. On the other hand, if a local cache slice is too small for the program's working set and its performance suffers due to this, subsequent page allocations may be directed to cache slices in other cores to increase the effective cache size. We call this operation *page spreading*. Page spreading can effectively increase the caching space seen by a program by selectively borrowing space from other cache slices. In addition, the OS may be forced to allocate physical pages from other free lists than the most desirable ones, if they have very few free pages currently, below a threshold. This operation is called *page spilling*, and is important for a balanced use of available physical pages, if either the SELECTION or REGION method is used.

While spreading pages, the OS page allocation should consider *data proximity*. Figure 4 shows how tiles are assigned a *tier* depending on the program location. Tiles marked with a same tier number can be reached from the program location in the same minimum latency, if there is no network contention. As an example, consider a program running on tile 5, as in the case 3 of Figure 4. Page spreading will be performed on tile 1, 4, 6, and 9 (*i.e.*, tier-1 tiles) before going to other tiles. This judicious page spreading is important not only for achieving fast data access but also for reducing overall network traffic and related power consumption.

Further, the OS page allocation should consider *cache pressure* in addition to data proximity when spreading pages. Under a heavy cache pressure, a tile may experience a large volume of capacity misses, rendering itself unable to yield cache space to other tiles. If all tier-1 tiles received many pages and suffer from high cache pressure, for example, page spreading must be done to other tiles with fewer pages allocated to their cache slice, even if they are not a closest tile. For this purpose, we define cache pressure to be program's time-varying working set (*e.g.*, approximated by the number of actively accessed pages) divided by cache size.

Finally, by applying different page spreading policies (*i.e.*, by applying different parameters in the page allocation algorithm) to different programs, the system can provide a varied quality of service to them [11]. For instance, a high-priority program may be given two cache slices, while other programs do not see them.

3.2.3 *Embracing parallel workloads*

So far our discussions were concerned mostly with multiprogrammed workloads. In this subsection, we will discuss how the proposed OS-level page allocation approach can help execute a parallel application efficiently on a tile-

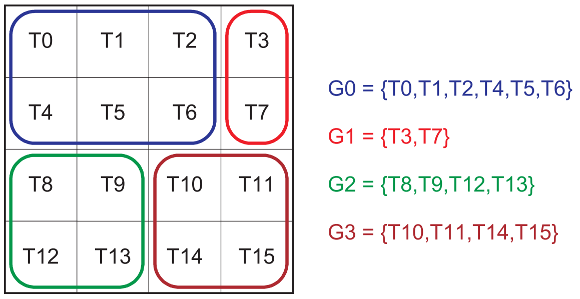

Figure 5: Four virtual multicores on a 16-tile chip.

based multicore processor.

When parallel applications are running, the OS will try to schedule the communicating processes and allocate their pages in a coordinated way to minimize the overall cache access latency as well as the network traffic. When a hardware-based shared caching scheme with line interleaving is used, the OS has no control over data distribution and there is little it can do to improve performance and traffic. Using our approach, on the other hand, the OS will ensure that new page allocations are directed to cache slices close to the requesting core ("data proximity") to reduce the cache access latency. Since a number of processes share their data in a typical parallel application, it is natural to spread pages to the cores that run these communicating processes.

To minimize the network traffic between the cores, the OS will try to schedule a parallel application onto a set of processor cores that are close to each other ("program proximity"). Suppose we have four parallel applications requiring six, four, four, and two processors, respectively. Figure 5 shows a mapping of the four parallel jobs to sixteen cores. The cores are grouped by the OS to form *virtual multicores*. Cache slices within a virtual multicore will be shared by the parallel program running on the virtual multicore. As the OS will opt for data and program proximity, a virtual multicore will comprise tiles that form a cluster on the chip surface.

Within each virtual multicore, OS can allocate pages in a round-robin fashion, in response to the core which requested the memory page, based on a data use prediction model (*e.g.*, obtained through compiler analysis or program profiling), or using a hybrid of them. Depending on the parallel programming model used, migrating processes (considering data access patterns) within a virtual multicore may result in improved data access latency.

Our virtual multicore approach guarantees that coherence traffic and tile-to-tile data transfers are confined within each virtual multicore, potentially improving the overall performance and energy consumption of parallel jobs. Compared with private caching, our approach maximizes the L2 caching space for shared data since there is no data replication, while placing an upper bound on the network latency and minimizing the related network traffic on an L1 cache miss. Compared with line-interleaved shared caching, our approach results in much less network traffic and leads to no cache contention between different (parallel) programs. As discussed in Section 3.2.2, spreading pages outside a virtual multicore should be done in the light of cache pressure and data proximity at the virtual multicore level.

3.2.4 *Data replication*

When a page to cache slice mapping is created, one can allocate a cache slice number to a *virtual page number* or *physical page number*. If physical page numbers are used during mapping, all running programs will have the same view on the mappings, and will access an identical cache slice for a same (physical) memory block. On the other hand, if virtual pages are given cache slice numbers, it is possible that two different programs will look for the same (physical) memory block in two distinct cache slices, leading to the same cache lines cached in multiple cache slices (*i.e.*, data replication). Although a shared cache organization does not allow data replication, we can allow data replication under certain conditions without incurring cache coherence problems. For example, read-only data, such as program codes or a constant table, can be replicated to achieve a lower access latency.

3.3 Discussions

The proposed L2 cache management approach uses a simple shared cache hardware, while providing choices from a wide spectrum of private and shared caching strategies in a dynamically controllable way. Shared caches are especially advantageous for large parallel applications [12], but private caches offer better performance for applications with a smaller memory footprint [8].

The OS will benefit from architectural support to track page access behaviors (*e.g.*, access frequency and sharing degree) for more accurate decisions on page allocations. Detailed program execution information such as working set sizes and data access patterns, obtained through compiler analysis and on-line/off-line profiling, can be used as hints to the OS page allocation.

Finally, we note that our approach does not preclude incorporating previously proposed hardware optimizations. For example, fine-grained cache line-level data replication such as victim replication [26] can be employed within the OS-based L2 cache management framework for even higher performance.

4. PRELIMINARY EVALUATION

4.1 Experimental setup

4.1.1 *Machine model*

We develop and use cycle-accurate, execution- and trace-driven simulators using the source code base of the SimpleScalar tool set (version 4) [1], which model a 16-core processor chip organized in a 4×4 mesh, similar to Figure 1. Each tile includes a single-issue processor with a 16kB L1 I/D caches, similar to [16], and a 512kB L2 cache slice. The single-cycle L1 caches are 4-way set-associative with a 32-byte line size, and each 8-cycle L2 cache slice is 8-way set-associative with 128-byte lines. The aggregate L2 cache size is 8MB. For coherence enforcement, we model a distributed directory scheme.

When data is traversing through the mesh-based network, a two-cycle latency is incurred per each hop. In the worst case, the contention-free cross chip latency amounts to 24 cycles round trip. We modeled contention at L2 caches, network switches, and links. The 2-GB off-chip main memory latency is set to 300 cycles.

NAME	DESCRIPTION	INPUT
gzip	compression	reference
gcc	gcc compiler	reference (`integrate.i`)
mcf	combinatorial optimization	reference
crafty	chess game	reference
parser	English parser	reference
eon	probabilistic ray tracer	reference (chair)
vortex	object-oriented database	reference
twolf	place & route simulator	reference
wupwise	quantum chromodynamics solver	reference
galgel	computational fluid dynamics	reference
art	image recognition	reference (`c756hel.in`, `a10.img`)
equake	seismic wave propagation simulation	reference
ammp	ODE solver for molecular dynamics	reference
sixtrack	particle tracking for accelerator design	reference
apsi	metrology; pollutant distribution	reference
fft	fast Fourier transform	4M complex numbers
lu	dense matrix factorization	512×512 matrix
radix	parallel radix sort	3M integers
ocean	ocean simulator	258×258 grid

Table 1: Benchmark programs, grouped by CINT, CFP, and SPLASH-2.

The execution-driven simulator is used to run a single-threaded application, a multiprogrammed workload composed of a general benchmark and a "hard-wired" synthetic benchmark called *ttg* (described below). The trace-driven simulator is used to run an arbitrary mix of general benchmarks or a parallel application. A trace is generated by a cycle-accurate simulator with a perfect data memory system. Delays due to instruction cache misses, branch mispredictions, inter-instruction dependences, and multi-cycle instructions are accurately captured in the generated traces. When traces are imported and processed in the trace-driven simulator, additional delays due to cache misses and network traversals are taken into account.

We implemented a parameterized page allocation unit in our simulators. Based on the *demand paging* concept commonly used in UNIX-based systems [2], every memory access is caught and checked against already allocated memory pages. If a memory access is the first access to an unallocated page, the page allocator will pick up an available physical page and allocate it according to the selected allocation policy. Given the 2-GB main memory and the benchmark programs, we did not experience any page spilling (see Section 3.2.2) in all the experiments we performed.

4.1.2 Workloads

We use three types of workloads in our experiments: *single-threaded workloads*, *multiprogrammed workloads* and *parallel workloads*. For a single-threaded workload, we use one program from a set of SPEC2k benchmarks [24], four integer (*gcc*, *parser*, *eon*, *twolf*) and four floating-point programs (*wupwise*, *galgel*, *ammp*, *sixtrack*). These benchmark programs were selected because they have largely different memory footprints and access patterns. Programs were compiled to target the Alpha ISA using Compaq Alpha C compiler (V5.9) with the `-O3` optimization flag.

To construct a multiprogrammed workload, we use one program from the above SPEC2k benchmarks, which we call a *target benchmark*, and a synthetic benchmark program called *ttg* (tunable traffic generator). *ttg* generates a continuous stream of memory accesses. We can adjust the memory footprint of *ttg*, the rate of memory accesses injected into the network and memory system, and the level of contention within shared cache slices. During actual simulations, we run our target benchmark on core 5 and *ttg* on all the other cores. This arrangement is required for us to accurately assess the sensitivity of different L2 caching strategies to various network traffic and cache contention levels, which is not achievable by using an arbitrarily constructed mix of benchmark programs. After skipping the initialization phase and a 100M-instruction warm-up period, we collect analysis data during one billion instructions of the target benchmark.

For parallel workloads, we use *fft*, *lu*, *radix*, and *ocean* from the SPLASH-2 benchmark suite [25]. These programs run on four cores (tile 5, 6, 9, and 10). Shared memory program constructs such as locks and barriers are modeled using a set of magic instructions implemented with the SimpleScalar annotations [1]. We use gcc 2.7.2.3 (with `-O3`) to compile these programs to target SimpleScalar PISA. The benchmark programs and their input data used in our experiments are summarized in Table 1.

4.2 Results

4.2.1 Comparing policies, single-threaded workloads

The following policies are compared in our first experiment: *PRV* (private), *SL* (shared, hardware-based line interleaved), *SP** (shared, OS-based page allocation), and *PRV8* (private, 8MB) as a limit case. For page allocation (*SP**), we consider four different page allocation policies: *SP-RR* (page allocated to all cache slices round-robin), *SP80* (80% of pages allocated to the local cache slice and the remaining pages spread to the cache slices in the tier-1 cores, numbered 1, 4, 6, and 9), *SP60* (similar to *SP80* but with 60% of pages allocated locally), and *SP40* (similar to *SP80* but with 40% of pages allocated locally). *SP-RR* is an OS version of fine-

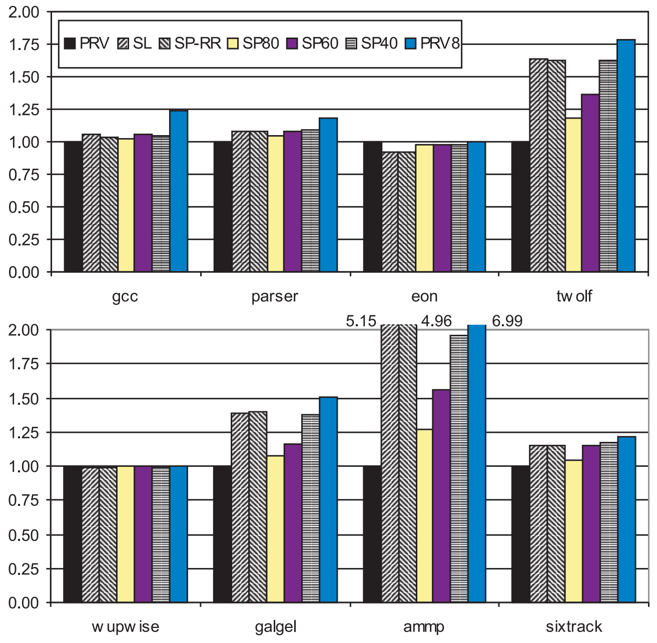

Figure 6: Single program performance ($time^{-1}$) of different policies, relative to *PRV*.

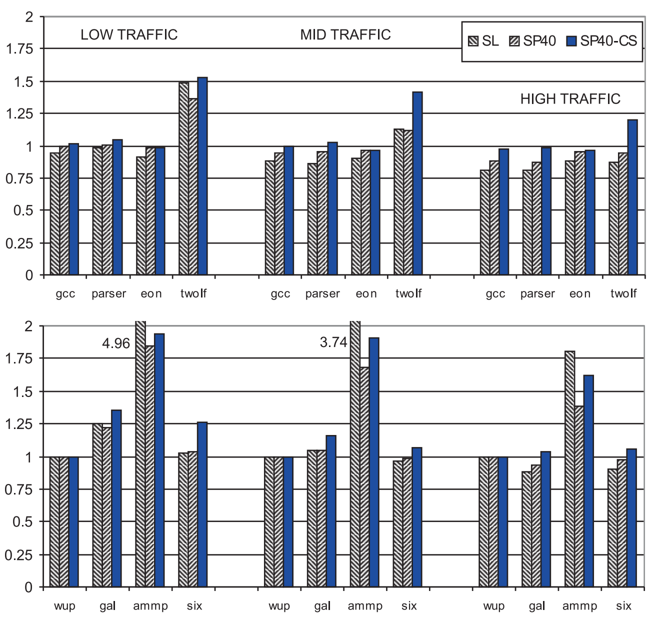

Figure 7: Performance sensitivity to network traffic. Performance relative to *PRV*, no traffic case.

grained shared cache implementation.

In our first experiment, we run and measure the performance of a single benchmark program on core 5 and have all the other cores stay idle. This “single-threaded” configuration is an important special case for a multicore processor [26], and gives us an insight into the maximum performance each different configuration can offer.

Figure 6 shows that *wupwise* is almost insensitive to the L2 caching scheme used since it has a very high L1 hit rate. On the other hand, *twolf*, *galgel*, and *ammp* have a bigger data footprint and get a large performance gain if more caching space is provided (*e.g.*, with *SL*). In case of *eon*, *SL* and *SP-RR* performed worse than *PRV* even though they provide more caching space, due to their increased average L2 cache access latencies. The performance of *SP-RR* closely matches that of *SL*, suggesting that this software-based shared caching policy is a faithful imitator of the hardware-based shared caching scheme.

Comparing with *SL* and *SP-RR*, the tier-1 page spreading schemes (*SP**) achieved comparable performance for programs like *twolf*, *galgel*, and *sixtrack*. In case of *ammp*, *SP** schemes did not provide enough caching space and they perform relatively poorly, topped at around 2× the performance of *PRV*, while *SL* and *SP-RR* achieve about 5×. *SL* performed best for *gcc*, *parser*, *twolf*, and *ammp* in this experiment; It is however very sensitive to cache and network contention as we will see in the following discussions.

Table 2 shows the L2 cache load miss rates and the on-chip network traffic of the studied programs. *PRV* showed larger miss rates than other configurations, leading to higher off-chip traffic levels. Page spreading (*SP**) was shown to be effective in reducing miss rates. As more caching space is used (*i.e.*, moving from *SP80* to *SP40*), increasingly lower miss rates were achieved. In case of *wupwise*, there are only compulsory misses and caching space or caching strategy did not make any impact.

In terms of on-chip network traffic, fine-grained shared cache configurations, *SL* or *SP-RR*, resulted in much more traffic than *PRV*. Again, the *SP** configurations show varying on-chip traffic levels. Clearly, spreading pages leads to more caching space and reduces miss rates and off-chip traffic as a result; In turn, it increases on-chip network traffic. This trade-off should be carefully considered while spreading pages, given the on-chip and off-chip communication overheads and the performance and power consumption goals.

4.2.2 Sensitivity to cache/network contention, multi-programmed workloads

We repeated experiments with a varying level of contention in cache and network, by running multiple copies of *ttg*. Three traffic levels, *LOW* (low traffic), *MID*, and *HIGH*, were generated, by setting the average L2 cache miss interval of *ttg* to 1500, 300, and 60 cycles, respectively. These values were determined after examining the eight SPEC2k programs we study; Their average L2 cache miss interval is in the range of 30 to 500 cycles when there is no contention. In this experiment, We focus on comparing the following three policies: *SL* (shared, hardware-based line-interleaved), *SP40* (shared, page-interleaved, 40% of pages allocated locally and the rest spread to tier-1 tiles), and *SP40-CS* (*SP40* with “controlled spreading”). In *SP40-CS*, we limit spreading unrelated pages onto the cores that keep the data of the target application.

Overall, Figure 7 shows that shared cache designs (*e.g.*, *SL* and *SP40*) are sensitive to cache and network contention, since cache lines are often fetched over the on-chip network. The target benchmark performance of *SL* is comparable to that of *SP40* under a light traffic load, but at a heavier traffic level, *SP40* gradually performs better than *SL* except for *ammp*.

In terms of overall chip throughput (*i.e.* the number of

		PRV	*SL*	*SP-RR*	*SP80*	*SP60*	*SP40*	*PRV8*
load miss rate (%)	*gcc*	2.9	0.1	0.5	2.8	2.1	1.8	0.1
	parser	6.6	0.5	0.6	5.8	3.7	2.6	0.4
	eon	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	twolf	16.3	0.1	0.1	13.1	7.3	1.6	0.0
	wupwise	25.0	25.0	25.0	25.0	25.0	25.0	25.0
	galgel	6.3	0.1	0.1	5.0	3.4	0.9	0.1
	ammp	46.6	0.1	0.4	34.9	26.4	18.9	0.1
	sixtrack	13.5	0.5	0.5	10.4	3.2	1.4	0.5
on-chip network traffic	*gcc*	10.8	270.4	261.7	55.6	76.0	135.9	0.4
	parser	8.7	96.8	96.5	18.2	18.9	40.4	0.5
	eon	0.0	86.9	90.2	17.7	20.4	23.7	0.0
	twolf	35.0	138.2	150.1	37.8	48.4	67.8	0.1
	wupwise	35.1	39.4	39.9	10.1	15.6	20.3	0.1
	galgel	38.0	412.0	406.6	76.2	132.3	185.8	0.6
	ammp	441.7	810.9	803.4	306.9	361.9	424.6	0.5
	sixtrack	9.6	57.2	60.9	15.8	18.9	22.0	0.4

Table 2: L2 cache load miss rate and on-chip network traffic (message-hops) per 1k instructions.

instructions committed over a same period of time), *SP40* was better than *SL* slightly, by 2% to 4%. Interestingly, *SP40* often experienced more cache sharing contention than *SL*, since programs have less network contention and tend to access caches more frequently, and caches are not globally shared as in *SL*, resulting in more hot cache sets.

In terms of network traffic, *SP40* cuts down the on-chip network traffic by at least 50% in all the programs, compared with *SL*. Considering these trade-offs, a complete OS and microarchitecture design of our approach will try to optimize performance and power by controlling both data placement, which will determine the minimum latency to fetch data and the associated network traffic, and data sharing degrees, which will affect contention within each cache slice.

Lastly, *SP40-CS* achieves best performance under heavy traffic (except for *ammp*); This result shows that the proposed software-based cache management approach can differentiate the hardware resources (*i.e.*, execution environment) seen by programs and can give preference to a high-priority program by not allowing other programs to interfere with it. A pure hardware-based scheme like *SL* does not provide this flexibility.

4.2.3 Parallel workloads

In this experiment, we measured the performance of parallel workloads. We compare three caching policies: *PRV*, *SL*, and *VM* (virtual multicore). In *VM*, page allocations were performed in a round-robin fashion on the participating cores. Figure 8 reports the result.

Except *fft*, *PRV* outperformed *SL*. Apparently, the studied programs were highly optimized to maximize data locality even on small caches [25] and as a result their performance on *PRV* is often better than that of *SL*, as similarly evidenced in previous studies [8, 26].

In case of *fft* and *radix*, the studied caching schemes resulted in small performance difference. On the other hand, *lu* and *ocean* had a higher L1 miss rate than *fft* and *radix*, and were affected to a larger extent by the L2 caching scheme employed. *VM* was shown to be consistently better than other policies. In the case of *ocean*, *VM* outperformed *PRV* by 5% and *SL* by 21%.

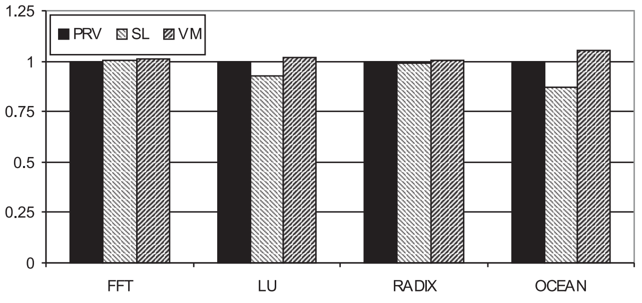

Figure 8: Performance of parallel workloads, relative to *PRV*.

4.2.4 Cache slice isolation

Figure 9 shows how program performance degrades as a variable number of cache slices become unavailable (*e.g.*, due to hardware faults). For this experiment, we use three multiprogrammed workloads: *W1* (*gzip*, *mcf*, *crafty*, *vortex*), *W2* (*mcf*, *crafty*, *wupwise*, *art*), and *W3* (*wupwise*, *art*, *equake*, *apsi*). We compare two policies, namely *SL* and *SP-RR*. *SP-RR* performs a round-robin page mapping to the available cache slices.

The result shows that *SL* is very sensitive to the loss of cache slices; this is because much of the total memory space become *uncacheable*. On the other hand, the performance of *SP-RR* is almost insensitive, clearly showing how the proposed flexible L2 cache management helps harvest good performance even in the presence of unavailable cache slices.

5. CONCLUSIONS

This paper considered an OS-microarchitecture approach to managing on-chip L2 cache slices in a future multicore processor built with an advanced process technology. In such processors, L2 cache management becomes a crucial design aspect in order to overcome non-uniform cache access latency for good program performance and to reduce on-chip network traffic and related power consumption.

The proposed approach is flexible; it can easily mimic various hardware strategies and furthermore can provide differ-

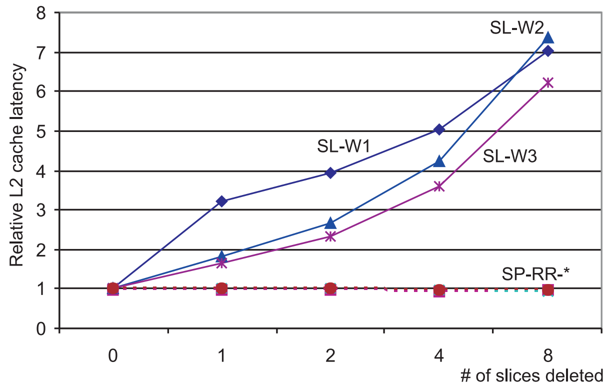

Figure 9: Average L2 cache latency when cache slices are deleted (relative to *SL* configurations).

entiated hardware environment to running programs by controlling network traffic and cache contention due to sharing. Such flexibility can be a critical advantage, because under a severe power constraint it will be of utmost importance to maximize the efficiency of on-chip resource usage with varying workload behaviors [3, 10]. Furthermore, unreliable cache slices can be easily isolated in our approach.

There are several directions for future research. It will be interesting to incorporate architectural techniques such as cache block replication [26] or page recoloring [20] and to study how they will improve data locality and cache traffic in the context of our L2 cache management approach. Also the impact of page level granularity mapping needs to be further investigated. To accurately monitor cache performance and its behavior, light-weight hardware-based monitors will be beneficial. Detailed program information (*e.g.*, data usage pattern) through compiler analysis or profiling may prove very useful.

6. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 35(2):59–67, Feb. 2002.

[2] M. J. Bach. *Design of the UNIX Operating System*, Prentice Hall, Feb. 1987.

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

[4] D. Burger and J. R. Goodman. "Billion-Transistor Architectures: There and Back Again." *IEEE Computer*, 37(3):22–28, March 2004.

[5] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 264–276, June 2006.

[6] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Proc. Int'l Symp. Computer Architecture*, pp. 357–368, June 2005.

[7] J. Huh, D. Burger, and S. W. Keckler. "Exploring the Design Space of Future CMPs," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, pp. 199–210, Sept. 2001.

[8] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," *Proc. Int'l Conf. Supercomputing*, pp. 31–40, June 2005.

[9] Intel Corp. "A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors," *Technology@Intel Magazine*, May 2005.

[10] ITRS (Int'l Technology Roadmap for Semiconductors). 2005 Edition. `http://public.itrs.net`.

[11] R. Iyer. "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," *Proc. Int'l Conf. Supercomputing*, pp. 257–266, June 2004.

[12] A. Jaleel, M. Mattina, and B. Jacob. "Last Level Cache (LLC) Performance of Data Mining Workloads on a CMP – A Case Study of Parallel Bioinformatics Workloads," *Proc. Int'l Symp. High-Perf. Computer Arch.*, pp. 88–98, Feb. 2006.

[13] R. E. Kessler and M. D. Hill. "Page Placement Algorithms for Large Real-Indexed Caches," *ACM Trans. Computer Systems*, 10(4):338–359, Nov. 1992.

[14] C. Kim, D. Burger, and S. W. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proc. Int'l Conf. Architectural Support for Prog. Languages and Operating Systems*, pp. 211–222, Oct. 2002.

[15] J. Kong and G. Lee. "Relaxing the Inclusion Property in Cache Only Memory Architecture," *Proc. Euro-Par*, pp. 435–444, August 1996.

[16] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, 25(2): 21–29, March-April 2005.

[17] C. Liu, A. Sivasubramaniam, and M. Kandemir. "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," *Proc. Int'l Symp. High-Performance Computer Architecture*, pp. 176–185, Feb. 2004.

[18] B. A. Nayfeh and K. Olukotun. "Exploring the Design Space for a Shared-Cache Multiprocessor," *Proc. Int'l Symp. Computer Architecture*, pp. 166–175, April 1994.

[19] B. A. Nayfeh, K. Olukotun, and J. P. Singh. "The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Architecture*, pp. 74–84, Feb. 1996.

[20] T. Sherwood, B. Calder, and J. Emer. "Reducing Cache Misses Using Hardware and Software Page Placement," *Proc. Int'l Conf. Supercomputing*, pp. 155–164, June 1999.

[21] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, J. B. Joyner. "POWER5 System Microarchitecture," *IBM J. Res. & Dev.*, 49(4): 505–521, July. 2005.

[22] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 346–356, June 2005.

[23] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. "The Impact of Technology Scaling on Lifetime Reliability," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 177–186, June 2004.

[24] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Int'l Symp. Computer Architecture*, pp. 24–36, July 1995.

[26] M. Zhang and K. Asanović. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 336–345, June 2005.