Managing Distributed, Shared L2 Caches through OS-Level Page Allocation

Sangyeun Cho Lei Jin
Department of Computer Science
University of Pittsburgh

{cho,jinlei}@cs.pitt.edu

Abstract

This paper presents and studies a distributed L2 cache management approach through OS-level page allocation for future many-core processors. L2 cache management is a crucial multicore processor design aspect to overcome non-uniform cache access latency for good program performance and to reduce on-chip network traffic and related power consumption. Unlike previously studied hardware-based private and shared cache designs implementing a "fixed" caching policy, the proposed OS-microarchitecture approach is flexible; it can easily implement a wide spectrum of L2 caching policies without complex hardware support. Furthermore, our approach can provide differentiated execution environment to running programs by dynamically controlling data placement and cache sharing degrees. We discuss key design issues of the proposed approach and present preliminary experimental results showing the promise of our approach.

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 [13, 19, 25]. Moreover, projections suggest that future processors may carry many more cores–10's or even 100's of cores within a single chip [5]. This trend is accelerated by the unprecedented technology advances and the less-than-desirable single core scalability [6].

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 [11]. 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 [18]. 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. Researchers are actively exploring the L2 cache design space to tackle this problem [8, 9, 12, 21, 30].

So far, L2 cache research and development efforts have been based on the two baseline designs: *private cache* and *shared cache*. In a private cache scheme, each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them. This automatic *data attraction* allows each processor 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 in aggregation form a logically single cache where each cache slice accepts only an exclusive subset of all memory blocks [13, 19, 25]. A shared cache design can potentially achieve better overall utilization of on-chip caching capacity than a private design because memory blocks and accordingly accesses are finely distributed over a large caching space. In addition, enforcing cache coherence becomes simpler because a memory block is found in a unique cache slice, which is easily derivable from the memory address. 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 [11, 12, 22, 30]. 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 [18] and data replication [30] in the context of a shared design to improve data proximity. On the other hand, private caches can be optimized to provide more capacity by

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

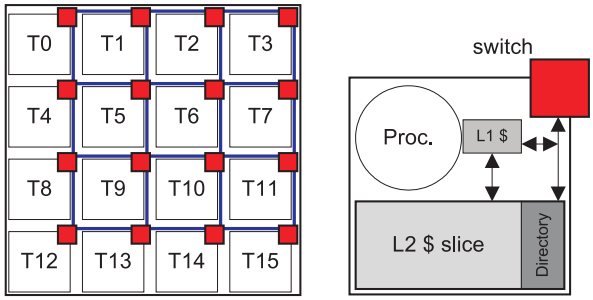

Figure 1. An example 16-core tiled processor chip and its tile (core).

limiting the degree of data replication or reserving a portion of cache space for other cores to use [8, 9].

In this work, we investigate an OS-managed data to cache slice mapping approach at the *memory page granularity* for future many-core processors. Figure 1 depicts an example 16-core processor chip. Our approach is based on the intuition that (i) finely interleaving cache lines to physically distributed L2 cache slices is not generally beneficial for good program-data proximity; and (ii) OS-level page allocation (*i.e.*, virtual-to-physical mapping) can directly control where, among many L2 cache slices, cache lines are placed if data to L2 cache slice mapping is done at the memory page granularity. Using simple shared cache hardware, our approach seamlessly implements a private caching policy, a shared cache policy, or a hybrid of the two *without any hardware support* under a multiprogrammed workload. For shared-memory parallel workloads, we consider a coordinated process scheduling and data mapping strategy based on a program-data proximity model.

The salient aspect of our approach is that process scheduling and data mapping decisions can be made in a dynamic and synergistic fashion with a full flexibility by the OS. For example, available processor cores can be clustered together, where the clusters are each managed with different L2 caching strategies. A high-priority program may be given a differentiated execution environment with more caching space than others. Cache slices can be made offline easily when power and fault management requirements exist. As such, our research is a critical departure from current hardware-oriented research trends and opens up new research directions that have not been fully explored. The initial results we obtained are encouraging.

The rest of this paper is organized as follows. Section 2 describes our idea of managing shared L2 cache slices through OS-level page allocation. Section 3 presents a quantitative evaluation of hardware-based L2 cache management schemes and our approach. Section 4 summarizes other related works. Finally, concluding remarks and future works will be given in Section 5.

2 L2 Cache Location Aware Page Allocation

2.1 Partitioning and allocating physical address space to cache slices

In a shared L2 cache design with multiple slices, memory blocks are uniquely mapped to a single cache slice, as shown in Figure 2(a). For example, recent commercial multicore processors such as IBM Power5 [25], Sun Microsystems T1 [19], and processor models found in research proposals [12, 30] use the following simple equation to define the mapping:

$$S = A \bmod N,$$

where S stands for the cache slice number, A for the memory block address, and N for the number of cache slices. We call the core (cache slice) hosting a memory block *home core* (*slice*) for it. Note that contiguous memory blocks are hosted by different cache slices using this method.

The above mapping method improves the L2 cache capacity utilization by finely distributing memory blocks among the available cache slices. It also increases the L2 cache bandwidth by scattering temporally close loads and stores to different cache slices. This method is not desirable in a future many-core processor, however, since this fine-grained data distribution will significantly increase the on-chip network traffic and effective cache access latency. If memory blocks are equally distributed to all tiles, for example, a streaming program on tile 0 will experience on average a 6-hop network delay (for request and response) on each L2 cache access, let alone the cache access latency.

To capture sufficient data locality within the local L2 cache slice, we propose that the data to cache slice mapping granularity should be of *page*, as shown in Figure 2(b), so that consecutive memory blocks (within a page) can reside in the same cache slice. This change can be achieved simply by replacing A with *physical page number* (PPN) in the above equation. Changing the mapping granularity has other advantages. For example, it will be much easier to adapt previous memory optimization techniques, such as various prefetching schemes [27] and OS page coloring [17], to a many-core processor.

Given this arrangement, here is our key observation: if memory to L2 cache slice mapping is done at the granularity of memory page, *it is the OS that determines to which L2 cache slice a particular data item belongs*, because it assigns each virtual page a PPN which in turn determines the associated L2 cache slice. This point is depicted in Figure 2(c). Our observation leads to the notion of *L2 cache location aware page allocation*, motivating a design approach departing from the current research trend that focuses on hardware mechanisms to statically implement a private, shared, or hybrid scheme [8, 9, 13, 19, 25, 30].

0-7695-2732-9/06 $20.00

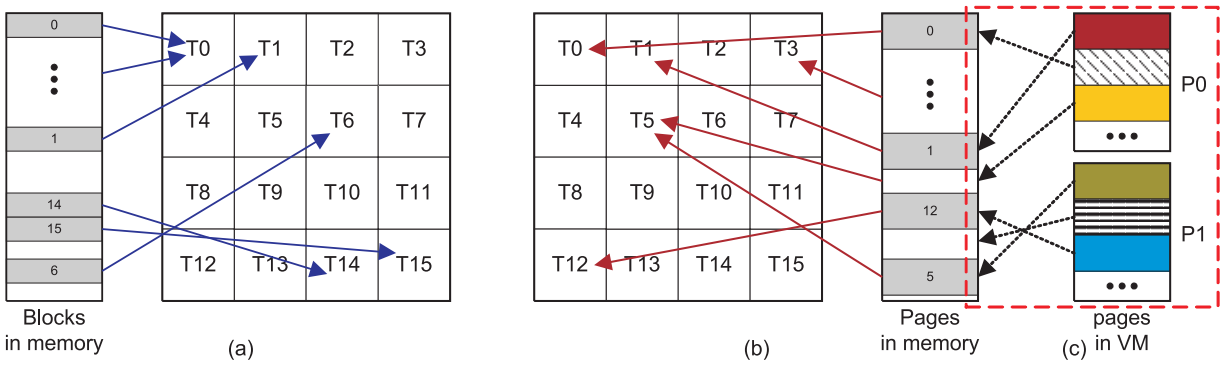

Figure 2. (a) Physical memory partitioning and mapping to cache slice at the cache line granularity. (b) Physical memory partitioning and mapping at the memory page granularity. (c) Virtual to physical page mapping (P0, P1: process 0 and 1, VM: virtual memory).

2.2 Data to cache slice mapping via 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 virtual-to-physical page mappings and deleting the allocated pages from the free list. Previously allocated pages are reclaimed by backing up their content to a non-volatile storage if needed, and putting these pages 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 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 the available physical pages in main memory so that each partition CG_i maps to a unique processor core i. Given *pmap*, the home core for a physical page (and accordingly memory blocks in it) is uniquely defined. A convenient *pmap* function will be modulo-N on PPN, partitioning the physical memory space equally regardless of its size. Now, the following virtual to physical page allocation strategies achieve the following previously proposed 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 a set of cores that share their L2 cache slices. Again, one can use a random or round-robin selection strategy within the selected group. Alternatively, one can consider the location of the requesting program.

Due to the flexibility of the proposed approach, it is possible to further customize the use of available cache slices, beyond the above well-established policies. For example, multiple cache slices can be dedicated to a single program by not allocating other programs' data to them. When there are idle processor cores (possibly not uncommon in a 100-core processor), their cache slices can be utilized by other cores. When a cache slice becomes unavailable due to certain chip operating constraints, such as power consumption or reliability issues, it can be easily made off-line.

Implementing cache location aware page allocation will entail non-trivial changes in the OS page allocation algorithms, however. Instead of a single free list, for example, it needs to manage N free lists, each associated with a congruence group[1]. Moreover, process scheduling should consider existing data mappings when allocating a processor core to a process, which requires adding more information in process control blocks. To obtain high performance and fully utilize the available cache space under changing workload behaviors, accurate, yet efficient performance monitoring schemes will be required.

[1] As a similar yet simpler example, a modern OS already manages multiple free lists to support optimizations such as *page coloring* or *bin hopping* [17], in an effort to reduce L2 cache conflict misses by not allocating too many virtual pages to specific page colors or bins within the cache.

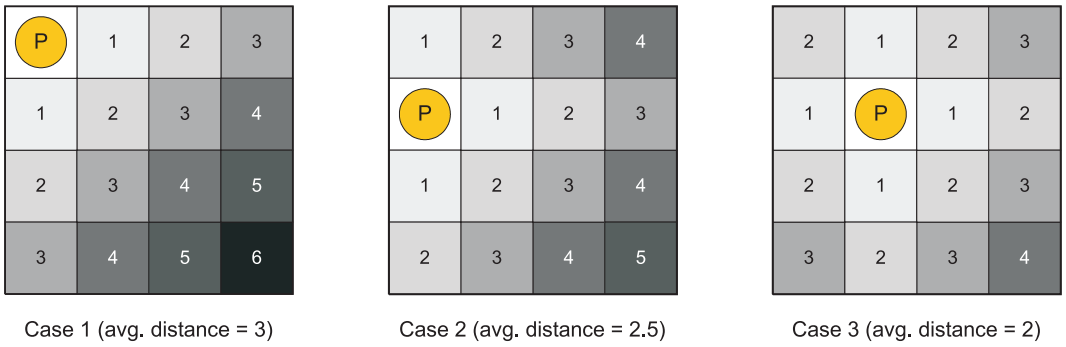

Figure 3. Program ("P") and data locations determine the minimum distance to bridge them.

2.3 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 [18]. We showed in the previous subsection that the OS has full control over where (among processor cores) cache lines will be placed if memory to L2 cache slice mapping is done at the page granularity. In comparison, private caches attract data closest to each processor core by actively replicating data in the local slice. If critical data set is large, however, private caches may perform poorly since each core's caching space is limited to its own cache slice. On the other hand, hardware-based shared caches provide potentially larger caching space than private caches, equally to all cores [13, 19, 25]; unfortunately, they do not have any provisions to improve data proximity.

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 simulates 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*. It can 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.

While spreading pages, the OS page allocation should consider *data proximity*. Figure 3 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 3. 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 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 lower cache pressure, 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.

We note that a modest architectural support can greatly simplify the task of accurately estimating cache pressure. Figure 4 shows an example mechanism to count actively accessed pages using a Bloom filter [4]. Whenever there is an access to the L2 cache, its page number is directed to the Bloom filter for look-up. If the page is found, there is no further action. If the page is not found in the Bloom filter, however, it is recorded there, and the counter is incremented. Essentially, the Bloom filter keeps the identity of all accessed pages and the counter counts the number of "unique" pages accessed. Both the data structures are periodically reset to track phase changes. The counter value can be then used to compute cache pressure. For a 512-kB cache slice, an 8-kB page, and a cache pressure of 4, a 512-byte Bloom filter will be able to accurately track all the accessed pages with a false positive rate of less than 0.5%.

After all, in the proposed approach, the OS determines a home core (*i.e.*, home slice) for each page to achieve a performance goal in a systematic way. Here, a performance goal may be to obtain maximum overall IPC (instructions per cycle) given a set of processes. In other cases, a power consumption constraint or priority levels in the processes may be given. A *home allocation policy* guides OS in de-

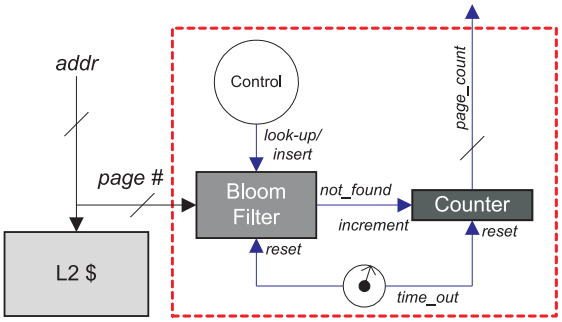

Figure 4. A Bloom filter based monitor mechanism to count actively accessed pages.

termining home slices by defining *profitability* toward the goal, computed for each (candidate) slice at page allocation times. A generic, idealized profitability measure for cache slice i given a program running on core k will look like:

$$P_i(k) = f(M, L, P, Q, C),$$

where M is a vector having recent miss rates of L2 caches, L is a vector having recent contention levels of network links, P is the current page allocation information (*e.g.*, per-process mapping and free list status), Q is the QoS requirements (*e.g.*, cache capacity quota per process), and C is the processor configuration (*e.g.*, number of caches). As a result of computing profitability for candidate cache slices, a cache slice with the highest score will be selected to become the home for the page in consideration. An actual embodiment of a policy will of course pose technical issues such as "how can we efficiently collect run-time information (M and L)?" and "what is an efficient and accurate f for a specific performance goal?".

2.4 Embracing parallel workloads with virtual multicores

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

When parallel applications are running, an 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

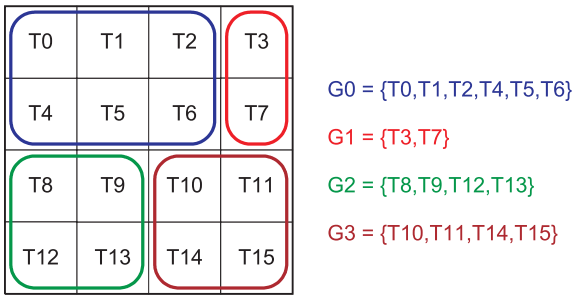

Figure 5. A virtual multicore (VM) example.

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 also 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* (VMs). Cache slices within a VM will be shared by the parallel program running on it. As the OS will opt for data proximity and program proximity, a VM will comprise tiles that form a cluster on the chip surface.

Within each VM, the 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. Our VM approach guarantees that coherence traffic and core-to-core data transfers are confined within each VM, improving the overall performance and energy consumption of parallel jobs.

Compared with the private cache scheme, the VM 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 the shared cache scheme, our approach can result in much less network traffic and leads to no cache contention between different (parallel) programs. As discussed in Section 2.3, spreading pages outside a VM should be done in the light of cache pressure and data proximity at the VM level.

2.5 Perspectives

The proposed OS-level L2 cache management approach uses simple shared cache hardware, while providing choices from a spectrum of private and shared caching strategies in a dynamically controllable way. Table 1 briefly compares the private caching, the shared caching, and the OS-based cache management approaches.

	PRIVATE	SHARED (w/ line interleaving)	OS-BASED
Hardware (tile)	Similar to a conventional uniprocessor core with two-level caches; coherence mechanism (*e.g.*, directory) to cover L1 and L2 caches	Coherence enforcement is simpler and is mainly for L1 because L2 is shared by all cores	(Same as SHARED); simple hardware-based performance monitoring mechanism will help reduce monitoring overhead
Software	(NA)	(NA)	OS-level support, esp. in the page allocation algorithm
Data Proximity	Data items are attracted to local cache slices through active replication; limited caching space can result in performance degradation due to capacity misses	Fine-grained cache line interleaving results in non-optimal data distribution; there is no explicit control over data mapping	Judicious data mapping though page allocation can improve data proximity
Network Traffic	High coherence traffic (*e.g.*, directory look-up and invalidation) due to data replication [23]; increased off-chip traffic due to high on-chip miss rate	High inter-tile data traffic due to remote L2 cache accesses, often 2× to 10× higher than PRIVATE [30]; lower off-chip traffic due to larger caching capacity	Low off-chip traffic like SHARED; improved program-data proximity through page allocation and process scheduling leads to lower inter-tile traffic than SHARED

Table 1. Comparing private caching, shared caching, and OS-based cache management approaches.

Throughout this paper, we base our hardware structure on a simple shared cache organization with no additional hardware support. As a result, only fixed-size congruence groups were considered in Section 2.2, which can become a burden for an OS. With a modest architectural support, such a burden can be relaxed. For example, if each TLB entry (and page table entries) is extended with a field containing the home slice number, the need for handling multiple free lists is completely removed because physical pages are decoupled from cache slices [16]. The OS will further benefit from hardware support to track page access behaviors (*e.g.*, access frequency and sharing degree) for more accurate page allocation decisions. We presented one such scheme in Section 2.3. Directly measuring cache misses (*e.g.*, sampled miss counters [24]) may also be helpful. Detailed program execution information such as data size and data access patterns, obtained through compiler analysis and on-line/off-line profiling, can be used as hints.

We note that our approach does not preclude incorporating previously proposed hardware optimizations. For example, with additional hardware support, fine-grained cache line-level data replication [30] and data migration [9] may be employed within our OS-based L2 cache management framework for even higher performance.

Finally, there are two potential drawbacks to the proposed page-level cache management approach. First, due to the coarser granularity of mapping, available caching capacity may not be fully utilized if there is disparate behavior by cache lines within a page. Second, there is additional software complexity involved, and the OS may not respond to workload changes quickly. An in-depth research including extensive implementation work is called for to fully address these issues, which is beyond the scope of this paper.

3 Quantitative Evaluation

3.1 Experimental setup

3.1.1 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 multicore processor chip with 16 tiles organized in a 4×4 mesh, similar to Figure 1. Each tile includes a single-issue processor with a 16kB L1 I/D caches, and a 512kB L2 cache slice. The single-cycle L1 caches are four-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 a datum 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.

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.

NAME	DESCRIPTION	INPUT
gcc	gcc compiler	reference (`integrate.i`)
parser	English parser	reference
eon	probabilistic ray tracer	reference (chair)
twolf	place & route simulator	reference
wupwise	quantum chromodynamics solver	reference
galgel	computational fluid dynamics	reference
ammp	ODE solver for molecular dynamics	reference
sixtrack	particle tracking for accelerator design	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 2. Benchmark programs.

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 experienced no page spilling instance (see Section 2.3) in all the experiments we performed.

3.1.2 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 [26], four integer (*gcc*, *parser*, *eon*, *twolf*) and four floating-point programs (*wupwise*, *ammp*, *sixtrack*, *apsi*). These benchmark programs were selected because they have largely different working set sizes 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 working set size 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 programs. After skipping the initialization phase and a 100M-instruction warm-up period, we collect analysis data during 1B instructions of the target benchmark.

For parallel workloads, we use *fft*, *lu*, *radix*, and *ocean* from the SPLASH-2 benchmark suite [29]. 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 used 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 2.

3.2 Results

3.2.1 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 per slice) 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 a shared cache implementation at the memory page level. For *PRV8* we assume the same cache access latency as *PRV*, while each cache slice is 8MB in size.

In our first experiment, we measure the performance of a single program running on core 5 whereas all the other cores stay idle. This "single-threaded" configuration is an important special case for a multicore processor [30], and gives

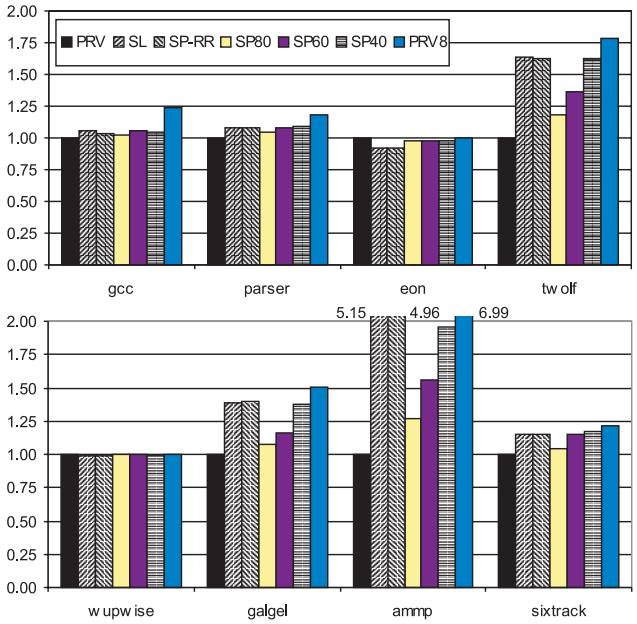

Figure 6. Single program performance ($time^{-1}$) of different policies, relative to *PRV*.

us an insight into the maximum performance each different policy can offer. Figure 6 shows the result.

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 working set 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 OS-based 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 3 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.

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.

3.2.2 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*.

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

Lastly, *SP40-CS* achieves the highest performance under heavy traffic, except for *ammp*. In case of *ammp*, a high local cache miss rate (due to the local allocation of 40% of the total pages) limited the performance of *SP40-CS*. The result shows that the proposed OS-based cache management ap-

0-7695-2732-9/06 $20.00

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

Table 3. L2 cache load miss rate and on-chip network traffic (message-hops) per 1k instructions.

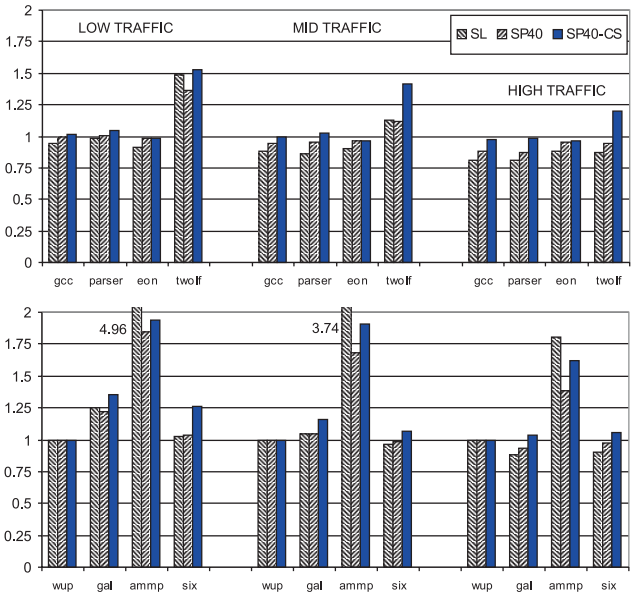

Figure 7. Performance sensitivity to network traffic. Performance relative to *PRV*, no traffic case.

proach 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.

3.2.3 Parallel workloads

In this experiment, we measured the performance of the parallel workloads. We compare three caching policies: *PRV* (private), *SL* (shared, hardware-based line-interleaving), 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 [29] and as a result their performance on *PRV* is often better than that of *SL*, as similarly evidenced in previous studies [12, 30].

In case of *fft* and *radix*, the studied caching schemes did not result in much 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. As a result, the performance of SL was negatively impacted due to a longer average L2 cache access latency. *VM* was shown to be consistently better than other policies. In case of *ocean*, *VM* outperformed *PRV* by 5% and *SL* by 21%.

4 Related Works

Private caching and cache sharing issues had been studied even before multicore processors emerged. Nayfeh and Olukotun [22] studied clustering multiple processors around a shared cache in a small-scale shared memory multiprocessor built on an MCM (Multi-Chip Module) substrate. They showed that shared caches provide an effective mechanism for increasing the total number of processors without increasing the number of invalidations. They further showed in their later work [23] how shared cache clustering helps reduce global bus traffic. In terms of overall performance, however, neither the private caching strategy, nor the shared caching strategy, was shown to be optimal for all the studied workloads, depending on the data access and

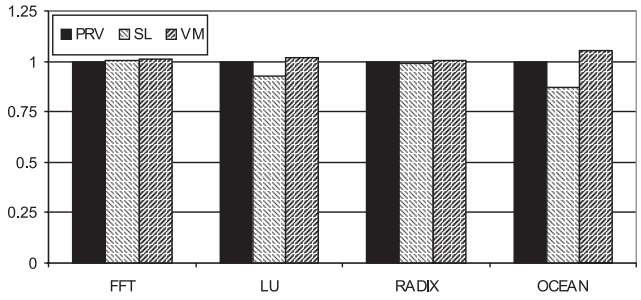

Figure 8. Performance of parallel workloads, relative to *PRV*.

communication patterns. Their results have been confirmed in the context of multicore processor designs [12, 30].

The unprecedented integration of transistors and uneven delay scaling of device and interconnect due to advanced process technologies are posing new L2 cache design challenges in single-chip processor architectures. Kim *et al.* [18] studied a spectrum of design alternatives for a large L2 cache in a wire delay dominated chip environment. Based on a single core model with many, relatively small L2 cache banks, they proposed a dynamic cache block migration scheme called *D-NUCA*. In a multicore processor running a multithreaded application with a great deal of data sharing, however, D-NUCA may become less effective because critical shared data will be often placed in central banks, equally far from all processors, if they are arranged in a dance-hall configuration. To overcome the deficiency, Beckmann and Wood [3] studied a stride-based prefetching technique.

Chishti *et al.* [9] proposed *CMP-NuRAPID*, a hybrid of private, per-processor tag arrays and a shared data array, with hardware optimizations to control replication and communication overheads. Zhang and Asanović [30] proposed *victim replication* in a shared L2 cache structure. In their design, each cache slice keeps replaced cache lines from its local L1 caches as well as its designated cache lines. Essentially, the local L2 cache slice provides a large victim caching space for the cache lines whose home is remote. Chang and Sohi [8] proposed *cooperative caching* based on a private cache design with a centralized directory scheme. They studied optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement, and global replacement of inactive data.

Liu *et al.* [21] proposed an L2 cache organization where different number of cache banks can be dynamically allocated to processor nodes, connected through shared bus interconnects. In their scheme, a hardware-based mapping table should be maintained by the OS so that right amount of cache banks can be allocated to different processors. Since their memory address to cache bank mapping is not deterministic, once an access from a processor misses in its assigned banks, the access should be directed to all other banks to find data, potentially incurring a large amount of network traffic. Huh *et al.* [12] studied a cache organization where cache line-bank mapping can be arranged dynamically. Their organization requires bank mapping tables in L1 cache controllers, bank controllers, and the central on-chip directory that must be maintained by the OS at the cache line granularity. Compared with the hardware-oriented optimization schemes summarized above, our approach is based on a simple shared cache design, with no special hardware support to bookkeep data to cache slice mappings.

Finally, there is a large body of research on the OS-level memory management in the context of NUMA (Non-Uniform Memory Access) multiprocessors [10, 20, 28]. In a NUMA machine, frequent cache misses going to remote memory can severely impact application performance. Cox and Fowler [10] evaluated a page replication and migration policy, backed up by a coherence enforcement algorithm in the OS. LaRowe and Ellis [20] studied a variety of page replication and migration policies in a unified framework. They employed a software-based *page scanner*, which maintains a set of reference counters by aging certain hardware reference bits. Using the collected information, it generates an artificial page fault as a triggering point for the OS to perform new memory management actions. Verghese *et al.* [28] uses hardware-based counters for guiding OS decisions in CC-NUMA (Cache Coherent NUMA) and CC-NOW (Cache Coherent Network of Workstations). Although both our work and these previous works opt for data proximity in a distributed memory structures, there are important differences between them. First, the target level of management (L2 cache vs. main memory) is different. A new mapping in the previous works involves actual data copy and delete operations, while in our work actual cache misses trigger individual memory block transfer. Second, the overhead involved in accessing data in a remote memory structure (remote L2 cache on-chip vs. remote memory via off-board network) is dissimilar. It results in very different trade-offs in applying memory management policies. To the best of our knowledge, our work is first to manage distributed L2 caches using OS-level page allocations.

5 Concluding Remarks and Future Work

This paper considered an OS-level page allocation approach to managing on-chip L2 caches in a future many-core processor built with an advanced process technology. Unlike previously studied hardware-oriented management approaches, the proposed OS-level page allocation approach is flexible; it can easily mimic various hardware strategies and furthermore can provide differentiated hardware environment to running programs by controlling data

placement and data sharing degrees. Such flexibility can be a critical advantage in a future many-core processor, 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 [5, 14].

There are several directions for future research. First of all, a detailed study of different page allocation policies with a full system simulation environment is needed. How to achieve a performance goal under dynamically changing workloads is of particular interest. Second, it will be interesting to incorporate other related architectural or OS techniques with our approach and study how they interact. For example, cache block replication [30], cache block migration [9, 18], page coloring [17], and page recoloring [24] are good candidates. Lastly, to accurately monitor cache performance and its behavior, light-weight hardware- and software-based monitors will be necessary. Detailed program information (*e.g.*, data usage pattern) through compiler analysis or profiling may prove very useful. We expect that more informed page allocation will not only reduce conflict misses in a shared cache when co-scheduled processes are running [7], but also offer adjustable quality of service to these processes [15].

Acknowledgment

We thank anonymous referees for their constructive comments. Our thanks also go to Hyunjin Lee, Rami Melhem, and Youtao Zhang at the University of Pittsburgh for their technical feedback on earlier drafts of this paper.

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] B. M. Beckmann and D. A. Wood. "Managing Wire Delay in Large Chip-Multiprocessor Caches," *Proc. Int'l Symp. Microarch.*, pp. 319–330, Dec. 2004.

[4] B. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, 13(7):422–426, July 1970.

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

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

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. "Predicting Inter-Thread Cache Contention on a Chip Multiprocessor Architecture," *Proc. Int'l Symp. High-Perf. Computer Arch.*, pp. 340–351, Feb. 2005.

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

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

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

[11] 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.

[12] 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.

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

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

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

[16] L. Jin, H. Lee, and S. Cho. "A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors," *Proc. ACM Workshop Memory Systems Performance and Correctness*, Oct. 2006.

[17] 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.

[18] 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.

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

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

[21] 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-Perf. Computer Arch.*, pp. 176–185, Feb. 2004.

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

[23] 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-Perf. Computer Arch.*, pp. 74–84, Feb. 1996.

[24] 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.

[25] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. "POWER5 System Microarchitecture," *IBM J. Res. & Dev.*, 49(4/5):505–521, July/Sep. 2005.

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

[27] S. P. VanderWiel and D. J. Lilja. "Data Prefetch Mechanisms," *ACM Computing Surveys*, 32(2):174–199, June 2000.

[28] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," *Proc. Int'l Conf. Architectural Support for Prog. Languages and Operating Systems*, pp. 279–289, Oct. 1996.

[29] 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 Arch.*, pp. 24–36, July 1995.

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