Achieving Predictable Performance with On-Chip Shared L2 Caches for Manycore-Based Real-Time Systems

Sangyeun Cho, Lei Jin, and Kiyeon Lee
Department of Computer Science
University of Pittsburgh
{cho,jinlei,lee}@cs.pitt.edu

Abstract

Doubling the number of processing cores on a single processor chip with each technology generation has become conventional wisdom. While future manycore processors promise to offer much increased computational throughput under a given power envelope, sharing critical on-chip resources, such as caches and core-to-core interconnects, poses challenges to guaranteeing predictable performance to an application program. This paper focuses on the problem of sharing on-chip caching capacity among multiple programs scheduled together, especially at the L2 cache level. Specifically, two design aspects of a large shared L2 cache are considered: (1) non-uniform cache access latency and (2) cache contention. We observe that both the aspects have to do with where, among many cache slices, a cache block is mapped to, and present an OS-based approach to managing the on-chip L2 cache memory by carefully mapping data to a cache at the page granularity. We show that a reasonable extension to the OS memory management subsystem and simple architectural support enable enforcing high-level policies to achieve application performance isolation and improve program performance predictability thereof.

1 Introduction

The unprecedented technology advances, commonly referred to as Moore's law, have enabled packing over a billion transistors on a single processor chip [4]. It is further projected that the rate of device scaling will be sustained for the foreseeable future [12]. In the era of billion-transistor chips, integrating multiple processor cores has become a clear and natural architectural choice of mainstream microprocessors [3, 18, 24], because the performance scalability of a single processor by pushing for higher clock frequency and instruction-level parallelism is running out of steam, and the power and energy consumption is the single most critical design constraint [4, 12]. It is now conventional wisdom to double the number of processor cores on a chip with each silicon technology generation.

While one may view new multicore processors as a miniature of existing shared-memory multiprocessors, the very fact that more system components are integrated on a single die renders widely different design constraints and therefore design practices. For example, many multicore processors implement a logically shared, yet physically distributed L2 cache organization to maximize the on-chip memory utilization and aggregate bandwidth [11, 18, 24]. Generalized, router-based on-chip networks are seriously considered for adoption [19]. Today, exploring the design space of and optimizing on-chip L2 caches and networks is an active research area [5–8, 13, 14, 16, 17, 20, 22, 28].

We note that employing increasingly more shared hardware resources will make performance prediction on a manycore processor harder, exemplified by the following examples, where, we assume a "tiled" processor having a fine-grained shared cache design similar to [18] and [28], and a 2D mesh network similar to [19]. Figure 1(a) depicts the layout of a tiled manycore processor. First, where a program runs, among many processor cores, will affect its performance. This *performance asymmetry* with regard to program location is caused by the changed cache access latency, as shown in Figure 1(b) and (c). Processor cores located closer to the center of the network (*e.g.*, processor cores labeled "A") will have a smaller average cache access latency than processor cores on the periphery of the chip (*e.g.*, cores labeled "C"). Even without any potentially conflicting co-scheduled processes, an OS process scheduling decision alone can affect a program's performance. Figure 1(c) further shows that the cache hit la-

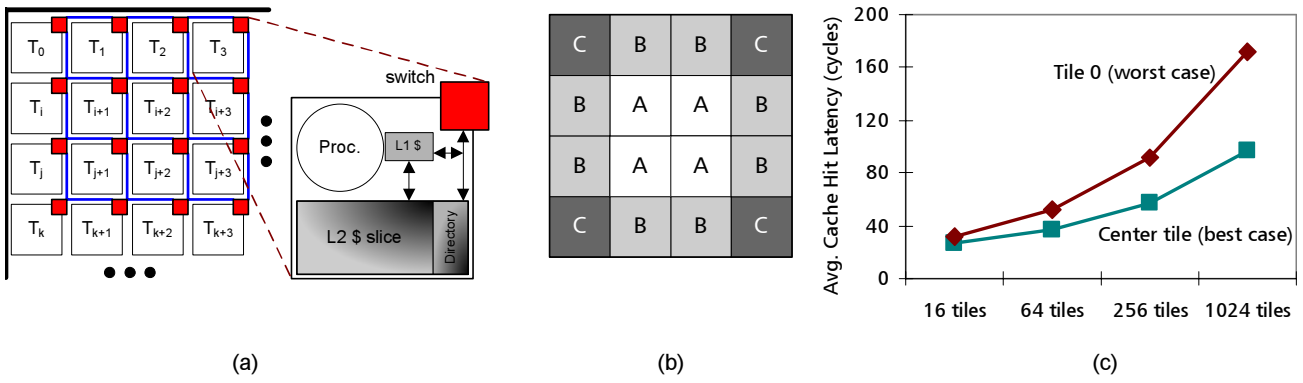

Figure 1. (a) A tiled manycore processor. (b) Three groups of cores having the same average L2 cache hit latency. (b) Average cache hit latency in processors having 16–1024 tiles. Results for the best and worst cases are shown. A mesh network with a 5-cycle hop delay and a cache memory with a 12-cycle hit latency are assumed. Cache blocks are interleaved to all caches [18,24,28].

tency gap between a center tile and a tile on the chip's angular point widens as the number of cores grows.

Second, uncoordinated, "free contention" at various shared resources, such as network routers and L2 caches, will impact performance predictability. Since the network traffic patterns seen by each router will depend on the temporal and spatial interleaving of processes on all processor cores, it is typically very difficult to accurately estimate or predict the impact of contentions at routers on a specific program's performance. Likewise, contentions at the L2 cache level have been considered as the key factor affecting the program performance variability [17,21]. As large as a 5× response time difference has been reported, due to unmanaged sharing of L2 cache resources when only two programs are co-scheduled.

In this paper, we are concerned with the performance variability caused by sharing L2 caches among multiple programs in a tiled manycore processor. Given a distributed L2 cache organization (such as the one in Figure 1(a)), our goal is to utilize the caching capacity *flexibly*, in order to (1) allocate an adjustable amount of caching space to each program (*i.e.*, "on-demand cache resource provisioning"); (2) allocate caching space close to the program location for improved performance (*i.e.*, "program-data proximity" awareness); and (3) limit the performance interference due to contentions at L2 caches among co-scheduled programs (*i.e.*, "performance isolation" [27]). Currently practiced industry designs largely lack capabilities to attain these properties.

Our current approach to achieving the above goal is centered on the idea of *explicit data mapping* via a hardware-software collaborative way. Previously, we observed that if the physical address to L2 cache slice mapping is done at the page granularity, the OS controls where a cache block is mapped to, when it maps a physical page to a virtual page [7,14]. We also showed that simple hardware support allows the OS to more freely map data to different cache slices, and cache block to cache slice mapping information can be retrieved and used without paying extra latency. Based on this framework, this paper discusses several design and management strategies to realize high-level policies governing the on-chip L2 cache usage. Our main conclusion is that the studied approach provides an efficient mechanism to manage the distributed shared L2 caches in a manycore processor in terms of performance isolation, as well as performance improvement.

The rest of this paper is organized as follows. To form adequate background, Section 2 summarizes the current L2 cache management approaches, namely, private cache and shared cache. Section 3 describes our approach – managing distributed shared L2 caches via two-dimensional page coloring, followed by a quantitative evaluation in Section 4. Conclusions will be summarized in Section 5.

2 Private Cache vs. Shared Cache

There are two baseline L2 cache design and management approaches in current-generation multicore processors: *private caching* and *shared caching*. Each private cache, as depicted in Figure 2(a), is associated with and dedicated to a specific processor core [1,26]. When there is a miss in the L1 cache of a core, a request is sent to its private L2 cache. If the access happens to

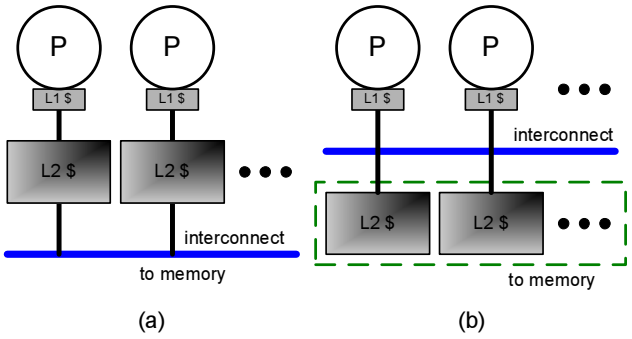

Figure 2. Conceptual view of (a) a private cache design and (b) a shared cache design.

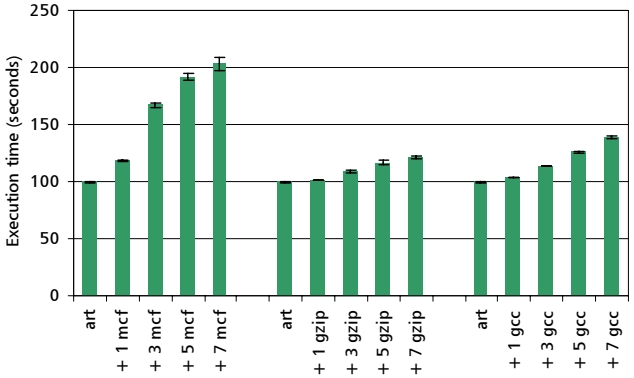

Figure 3. Program execution time of the program "art" (image recognition) in the SPEC2k benchmark, when run alone or with a variable number of other programs. The processor used is T1 [18], an 8-core chip with a 3MB shared L2 cache.

miss in the L2 cache, the missing block will be always brought into the cache. This automatic *data attraction* allows processors to access data quickly thereafter, leading to a low average cache hit latency. However, the limited per-core cache capacity in this scheme may incur many capacity misses, resulting in expensive off-chip memory accesses and degraded performance.

Shared caches form a single logical cache by having each cache slice accept only an exclusive subset of all memory blocks and allowing accesses from all processor cores to all cache slices, as shown in Figure 2(b) [11, 18, 24]. Typically, memory blocks are mapped to a cache slice based on a simple arithmetic function (*e.g.*, modulo) defined on the memory block address. When there is an L1 cache miss, therefore, the L2 cache slice to access is uniquely determined by inspecting the missed address. Overall, a shared cache design will result in better utilization of on-chip caching capacity than a private design because memory blocks and cache 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.

We note that the two caching schemes result in a different degree of program performance variability; because it does not allow sharing of caching space, the private cache scheme leads to limited performance interferences between programs. On the other hand, programs running on the shared cache may exhibit large performance variability. Figure 3 shows how the execution time of a program is affected by running other programs together. We limited the number of co-scheduled programs to ensure that program execution time is affected mainly by the sharing of the L2 cache. It is clearly shown that the execution time of a program on a shared L2 cache is susceptible to variability. Accordingly, the focus of this paper is on achieving low performance variability when a program with real-time constraints is running on a manycore processor employing a distributed shared L2 cache.

There are two streams of work related to this paper. The first set of work tries to overcome the non-uniform cache access latency of a large shared cache [16], by balancing between the private caching and the shared caching approach [6, 13, 28]. For instance, Zhang and Asanović [28] proposed *victim replication* based on a shared L2 cache organization, where each L2 cache slice can keep a replaced cache block from its local L1 cache as well as the designated cache blocks. Jin and Cho [13] showed that careful mapping of data to cache slices considering both data proximity and cache miss rate leads to superior program performance to the existing, private and shared caching schemes. Especially, they presented a profile-driven framework for finding an optimal data to cache slice mapping.

The second set of work considers cache partitioning techniques to achieve fairness (*i.e.*, same performance degradation due to interference) among a small number of co-scheduled programs or improve the overall throughput by limiting interferences [10,17,22]. Kim *et al.* [17] defined and evaluated five cache fairness metrics and studied cache partitioning algorithms. They found that enforcing fairness usually leads to better overall throughput. The utility-based partitioning technique in [22] monitors programs' cache usage and partitions the cache based on the obtained usage information. They reported a throughput improvement of 11% over the conventional, unmanaged shared caching scheme. Iyer [10] presented a cache management framework for achieving priority-based QoS in shared L2 cache. The proposed mechanism includes set partitioning, which is also used in [17, 22]. These works do not consider performance isolation, however.

3 Flexibly Managing Shared L2 Caches

The cache management issues considered in this paper – unpredictable and increased cache latency due to fine-grained data distribution and performance variability due to capacity sharing – are both closely related with the basic question of "where do we map a cache block, among many cache slices?" If we are given a mechanism for defining mapping between data and cache slices and for efficiently exploiting the mapping information on frequent events such as L1 cache misses, we can implement high-level policies describing how the L2 cache should be used by a set of programs.

3.1 Creating data to cache mapping

In our proposed approach, we create a data to cache slice mapping at the granularity of a memory page on a page allocation event in the OS. This mapping granularity allows efficient handling of the mapping data within the OS and the hardware (*e.g.*, TLB) [7, 14].

Now, let us introduce the notion of *cache bin*. A cache bin is a smallest group of cache sets which would hold an entire memory page, as depicted in Figure 4(a). The number of cache bins in a cache is simply the cache size divided by the product of the page size and the associativity (*i.e.*, bin capacity). We note that picking up a specific cache bin will in turn decide the cache slice holding the cache bin (*i.e.*, there are multiple cache slices and each cache slice holds multiple cache bins). Given this, to be more accurate in describing our scheme, *we assign a cache bin to a memory page, so that cache blocks belonging to the memory page are mapped to the cache bin*. While this mapping is determined at the time of page allocation, actual installation of cache blocks will occur on cache misses. Judicious memory page to cache bin mapping has been used to reduce conflict misses in a conventional L2 cache [15, 23].

The created mappings need be kept in the OS and also stored in a hardware structure for fast look-up. Figure 4 presents a TLB design extended with the cache bin number per memory page. Because the mapping information is available by the time an L1 cache miss is detected, the necessary L2 cache request can be immediately launched to the correct target cache slice. Assuming 8kB pages, an 8-way 4MB cache has 64 bins, and each bin can be addressed with a 6-bit number.

3.2 Two-dimensional page coloring

We call the mapping action involving a memory page and a cache bin in a manycore processor *two-dimensional page coloring*, because the mapping determines not only the cache slice (*i.e.*, dimension 1), but also the bin within the cache slice (*i.e.*, dimension 2), as we discussed in the previous subsection. The first dimension differentiates our scheme from traditional page coloring techniques [15, 23] in that it changes the distance between the program and the mapped data. The second dimension will determine the actual cache sharing between the two pages mapped to the same cache slice. It is important to notice that two memory pages mapped to a different cache bin will never compete with each other over cache capacity.

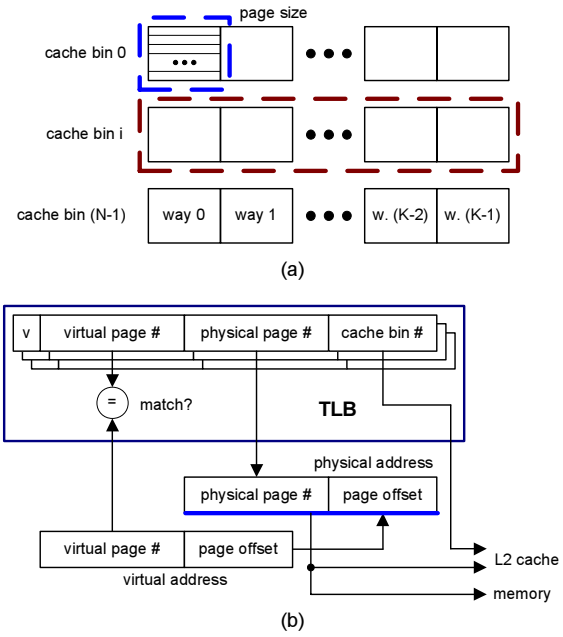

Figure 4. (a) Set-associative cache and cache bins. (b) TLB extended with a (global) cache bin number.

The two-dimensional page coloring can be used for achieving various high-level cache management goals. We have shown in our previous work [7] that the private cache scheme, the shared cache scheme, and the clustered, hybrid cache scheme [9] can be emulated in a rather straightforward way. Optimizing both the dimensions (*i.e.*, data proximity and cache miss rate) leads to better performance than what simple private and shared caching schemes can bring [13]. Cache slices or cache bins carrying hardware faults can be made offline and their impact masked by not mapping pages to them [14]. Likewise, by allocating exclusive sets of cache bins to potentially competing programs, the performance interference between them can be reduced.

In the following two subsections, we will highlight strategies to estimate and achieve the best performance from a program given the cache capacity, and to achieve performance isolation.

3.3 Profile-driven performance optimization

In this subsection, we consider the following two questions: (1) "what is the minimum number of cache bins needed to obtain a near-peak performance level given an application?" and (2) "given a manycore processor organization (esp. cache slices and network) and an application, how can we create the page to cache bin mappings to achieve the highest performance?" The first question is important for cache capacity provisioning and working set aware process scheduling, and the second question is important for optimizing individual programs. We will limit our discussions to an off-line, profile-driven approach here.

Answering the first question with profiling is relatively straightforward; give the largest number of cache bins to the application to obtain the peak performance first, and change the number of cache bins and repeat experiments to obtain a curve. One can utilize a search method (*e.g.*, binary search) or a multi-configuration simulation technique to make the process faster.

The second question is much more involving, and we adopt a methodology with three phases: *trace generation*, *trace analysis* and *generating mappings*. In the trace generation phase, memory references of the target program are collected. To accurately capture the related cost in the trace analysis phase, we collect only L2 cache references. In the trace analysis phase, we count the number of references to different pages and the number of inter-page conflicts, with the scope of the whole trace. The inter-page conflict information is used in the last phase when estimating the potential cache misses caused by placing a page to a cache bin. The first two phases are motivated and derived from an earlier work by Sherwood *et al.* [23].

The main idea of our algorithm is to minimize the overall cost of L2 cache accesses by iteratively computing the cost of assigning a particular page to all cache bins and selecting the cache bin with the smallest computed cost. Obtaining the cache access cost involves calculating both aggregate cache access latencies given the page location and the memory access penalties due to conflict misses. While the number of references per page is easily obtained given the memory reference trace, computing the number of conflicts between pages is impossible before they are assigned a cache bin. To tackle this complication, we assume that if there are two references to page A and B and there is no other reference to page B in between, these two references can potentially cause a conflict miss if page A and page B are placed in the same cache bin [23].

We define two matrices Reference[][] and Conflict[][], each of which keeps track of temporal relationships among references and counts the number of potential conflicts between any pair of pages, respectively. To update the matrices, references in the trace are processed one by one, as shown in Figure 5. For each reference, the column bits in Reference[][] corresponding to the accessed page are set to 1. All the bits in the row corresponding to the same page are then checked. Any bit previously set to 1 indicates that the current reference can cause a conflict with this previous reference. After all the references are processed, Conflict[i][j] will contain the number of potential conflicts when page i and j are mapped to the same cache bin. Note that the numbers in Conflict[][] are over-estimations as the conflicts are summarized for each page.

```
while trace is not empty {
  get the next reference R from trace
  PI = array index of the page accessed by R

  for (i = 0; i < total number of pages; i++) {
    Reference[i][PI] = 1;
    if (Reference[PI][i] == 1) {
      Conflict[PI][i]++;
      Reference[PI][i] = 0;
    }
  }
}
```

Figure 5. The algorithm to extract conflict information from a reference trace.

Given Conflict[][] and other necessary microarchitectural parameters, we can start mapping pages to cache bins. Since the page coloring problem is in general NP-complete [15], we adopt a heuristic approach to make the computation tractable. Our coloring algorithm evaluates pages from the one with the largest number of accesses and proceeds in a decreasing order. The cost of assigning a particular color or bin C to a page P is computed by the following cost function:

$$\begin{aligned} \mathsf{Cost(P,C)} &= \alpha \times \mathsf{TotalConflicts(P,C)} \times \mathsf{MemLatency} \\ &\quad + (1-\alpha) \times \mathsf{TotalAccesses(P)} \\ &\quad \times (\mathsf{L2Latency} + \mathsf{NoCDelay(C)}) \end{aligned} \quad (1)$$

In the above equation, TotalConflicts(P,C) is given as $\sum$ Conflict[P][Xi]/N for any page Xi already mapped to C. N stands for the number of pages that have been allocated to the cache bin. Without losing generality, we assumed in the above that the program location is fixed (and thus not shown) for the clarity of presentation.

Since TotalConflicts(P) is an estimate, we introduce a parameter α to mitigate the inaccuracy. α also controls the *page aggregation density*. With a smaller α, more weight is put on NoCDelay(), thus placing pages closer to the program location. As such, when α is 0,

the algorithm simulates a private cache. On the other hand, with α equal to 1, the algorithm simulates a shared cache, ignoring the network latency. The process of assigning a cache bin to a page using the above cost function is repeated until all pages are colored. The derived color assignment information is then used to direct the OS page allocations at run time.

3.4 Cache performance isolation

Due to its flexibility in managing a shared L2 cache, the proposed approach provides a simple and elegant way of achieving desired performance isolation, given a set of co-scheduled programs: Provide each program with its own, private cache by not sharing cache bins among the programs. To obtain good performance and not to introduce performance interference due to contentions at network routers, one will strive to map data belonging to a program to its local cache slice or cache slices near to the program. Hitting in a local cache slice does not generally create traffic on the network.

Cache partitioning assisted by the proposed two-dimensional page coloring is different from the previously proposed horizontal, within-set partitioning scheme [10,17,22] in several important ways. First, the proposed scheme can accommodate potentially more programs than the horizontal partitioning scheme, because the maximum number of partitions is limited by the number of cache bins in the former case (*e.g.*, 64 bins in a 4MB cache) and cache's associativity in the latter case (*e.g.*, 8 ways). Second, the proposed scheme allows us to simultaneously consider controlling capacity sharing and traffic flow (*i.e.*, two dimensions), in order to minimize performance variability, while the horizontal scheme alone does not. Note that contentions in network routers, as well as a cache slice, can affect program performance. Lastly, partitioning-related actions occur much more frequently in the horizontal partitioning scheme, as often as each cache block installation event. On the other hand, the proposed scheme allocates a cache bin on a page mapping event and does not further require hardware or software support to maintain partitions. In summary, due to their generally different operating requirements and effectiveness, we consider them complementary to each other.

4 Quantitative Study

In this section, we study the efficacy of the profile-guided static mapping scheme (Section 3.3) in overcoming non-uniform cache access latencies and the performance variability of several programs under different cache management schemes (Section 3.4).

4.1 Experimental setup

We extended the SimpleScalar tool set (v3.0) [2] to model a tile-based multicore processor on a 2D mesh. The baseline processor configuration is (4×4) tiles, where each tile has a processor with private L1 caches and a globally shared L2 cache slice. The modeled processor is either a 4-issue out-of-order or a single-issue in-order processor. The 32KB, 2-cycle L1 caches are 2-way associative and the 8-cycle L2 cache slice is 4-way associative and 256KB in size. Cache blocks are 64B (L1) and 128B (L2). A miss in an L1 cache triggers a request sent through the on-chip network to a target L2 cache. Each hop in the network takes 5 cycles and the main memory latency is 300 cycles. Otherwise stated, a program always runs on tile 5. We use 11 integer and 7 floating-point programs from the SPEC2k CPU suite [25] as workloads. After a fast-forward period of 500M instructions and a warm-up period, we collect statistics during a period of 800M instructions.

To study the performance variability of a program in a controlled manner, we introduce a synthetic benchmark program called *ttg* (tunable traffic generator), which 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 a target benchmark on core 5 and *ttg* on all the other cores and homogeneously change each *ttg*'s parameters to stress the target benchmark less or more.

4.2 Performance of static 2D page coloring

Figure 6 shows the performance of the conventional private caching scheme ("Private"), the conventional shared caching scheme ("Shared") and our "Static2D" (Section 3.3). Results are relative to the baseline shared caching scheme ("SharedBase") and simple page coloring with no profile information. We use an aggressive profile-guided page coloring technique [23] for Private and Shared for fair comparison.

It is shown that Static2D consistently outperforms Private and Shared. The performance of Private often suffers due to the relatively small cache slice size of 256KB; *vpr*, *twolf*, *art* and *ammp* are among the most affected. We observed a high L2 cache miss rate in these programs, which cannot be simply compensated by L2 cache latency savings. On the other hand, Shared always shows better performance than SharedBase by reducing conflict misses (but not access latency). Programs like *mcf*, *swim* and *mgrid* benefit much from the miss rate reduction and achieve over 50% perfor-

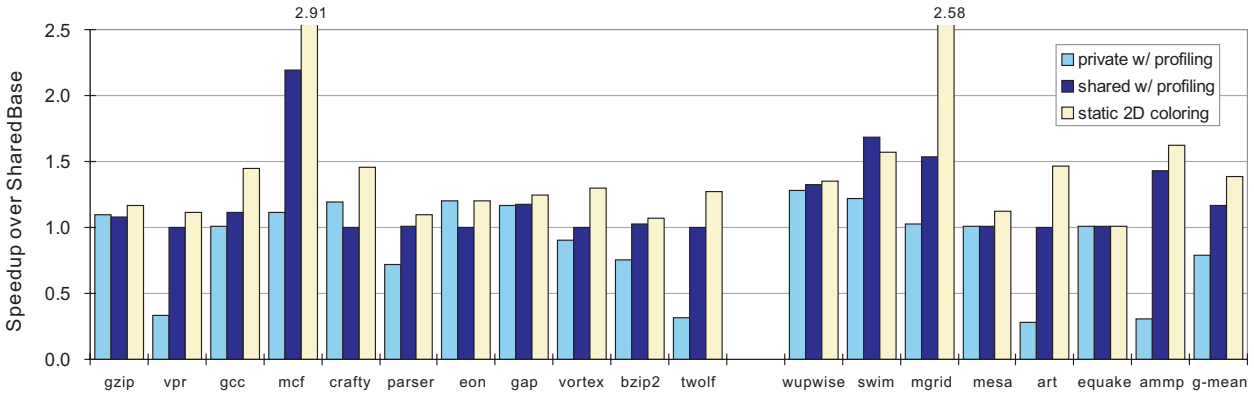

Figure 6. Program performance of Private**,** Shared**, and** Static2D**.**

mance improvement. Static2D achieves higher performance than both Private and Shared by balancing cache miss rate and cache access latency. *swim* is a notable exception, for which Shared achieves a better miss rate than Static2D due to its fine-grained block interleaving. On average, Static2D achieves 38.2% performance improvement over SharedBase, 18.5% over Shared, and 74.5% over Private.

Figure 7 shows how performance of different schemes scales when the cache slice size or the tile count is varied. When cache slices are small, miss rate is the dominant performance factor and Private performs poorly. As we increase the cache slice size, however, the gap between Private and Shared decreases and Private begins to outperform Shared at 2MB, where cache access latency becomes a determining factor. Though not plotted, Private and Static2D will merge finally and Shared will approach 1 (*i.e.*, degenerate to SharedBase) as we increase the cache size indefinitely. When there are more tiles on a chip, the average cache access latency of Shared and SharedBase grows (also shown in Figure 1(c)). Shared is shown to approach 1 as we add more tiles since latency becomes the dominant factor and the benefit of profile-guided coloring becomes negligible. By comparison, the performance of Private and Static2D is largely insensitive to the addition of tiles. Private begins to outperform Shared due to the performance degradation of Shared in larger-scale chips.

4.3 Program performance variability

Figure 8 presents four programs' performance variability under four different cache management schemes at different contention levels. Programs used are: *gzip* (compression), *art* (image recognition), *eon* (visualization), and *vortex* (OODB). Examined cache management schemes are: 2DPriv (all pages allocated to the local cache slice, protected against sharing), 2DSpread20

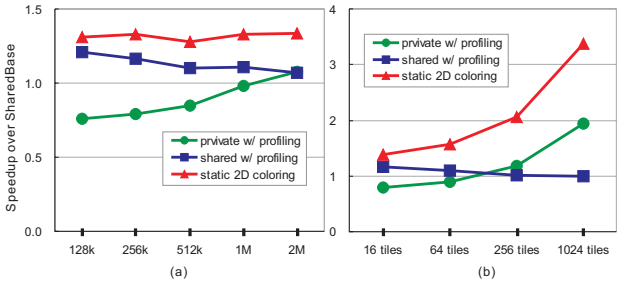

Figure 7. Program performance (average of all studied programs) when (a) cache slice size is varied and (b) tile count is varied.

(20% of pages allocated to other nearby cache slices, protected against sharing), 2DSpread60 (60% of pages spread to other nearby cache slices, protected against sharing), and SharedBase (conventional shared caches with cache block interleaving). We use four contention levels: "no contention" (*ttg* is dormant), "low contention" (*ttg* generates a non-local cache access every 1500 cycles), "mid contention" (every 300 cycles), and "high contention" (every 60 cycles). The parameters were chosen after studying various program characteristics of the SPEC2k benchmark. We used a single-issue processor model in this experiment to emphasize the impact of contention on program performance.

We make the following observations from the result. First, as we expected, the performance of SharedBase suffers much from contention, more severely at a higher contention level. Especially, *art* experienced performance degradation of over 70% under the high contention level. Other programs saw 5–26% performance variability. Clearly, a great deal of care must be taken to guarantee performance when an application with real-time constraints runs on a manycore processor employing shared caches. Second, across all the studied programs, 2DPriv isolates target program per-

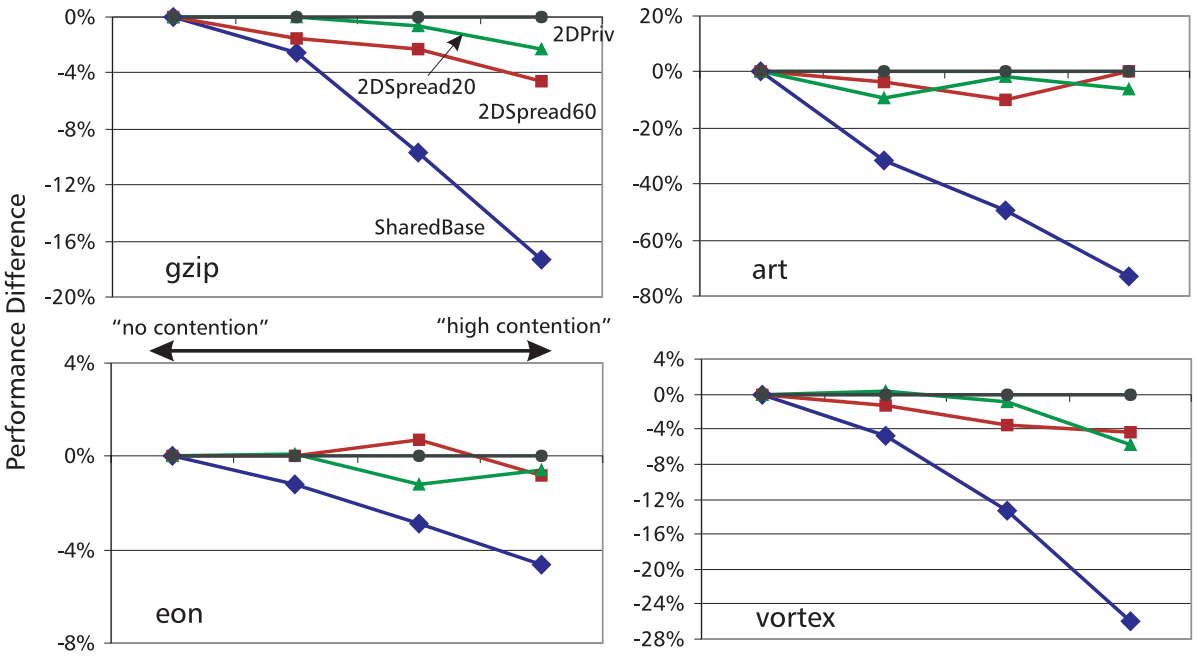

Figure 8. Performance variability of four programs, when contention level is adjusted from "no contention" (left) to "high contention" (right).

formance robustly; target program's performance is not sensitive to *ttg* behaviors. Of course, this does not necessarily mean that the performance of 2DPriv is better than that of other schemes (as can be deduced from Figure 6). The other two schemes, 2DSpread20 and 2DSpread60, show that even when the cache space for an application is protected, depending on the location of the cache slices used, performance variability exists. The possible causes are contention at network routers and contention at cache ports. The performance variability manifested in 2DSpread60 is somewhat more pronounced than that of 2DSpread20, as more accesses have to travel on the on-chip network in 2DSpread60.

To summarize, the studied performance isolation scenarios using the proposed two-dimensional page coloring technique (2D*) showed limited performance variability, within 6% for the studied programs, even under a high contention level. We expect that our approach can provide a differentiated, QoS-aware program execution environment on a manycore processor with distributed, yet shared L2 caches.

5 Conclusions

This paper studied the performance variability issue in a manycore processor employing distributed, shared L2 caches. In order to effectively utilize the large L2 cache capacity in a manycore processor and to achieve desired performance isolation among co-scheduled programs, we argued that explicit mapping of data to a cache slice is instrumental. We further showed that a synergistic, OS-architecture framework to flexibly map data to L2 cache slices at the page granularity provides a mechanism for us to efficiently implement on-demand L2 cache capacity allocation, performance-aware capacity allocation, and performance isolation at the same time.

Our future work includes (1) exploring the design space of a dynamic page coloring scheme for performance isolation; and (2) studying the interplay of process scheduling and various data mapping schemes.

References

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

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

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

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

[5] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. Int'l Symp. Computer Arch. (ISCA)*, 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 Arch. (ISCA)*, pp. 357–368, June 2005.

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

[8] J. Huh, D. Burger, and S. W. Keckler. "Exploring the Design Space of Future CMPs," *Proc. Int'l Conf. Parallel Arch. and Compilation Techniques (PACT)*, pp. 199–210, Sep. 2001.

[9] 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 (ICS)*, pp. 31–40, June 2005.

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

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

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

[13] L. Jin and S. Cho. "Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring," *Proc. Int'l Workshop Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, Feb. 2007.

[14] L. Jin, H. Lee, and S. Cho. "A Flexible Data to L2 Cache Mapping Approach for Future Multicore Processors," *Proc. Workshop Memory Systems Performance and Correctness (MSPC)*, pp. 92–101, Oct. 2006.

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

[16] C. Kim *et al.* "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches," *Proc. Int'l Conf. Arch. Support for Prog. Lang. and Operating Systems (ASPLOS)*, pp. 211–222, Oct. 2002.

[17] S. Kim, D. Chandra, and Y. Solihin. "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," *Proc. Int'l Conf. Parallel Arch. and Compilation Techniques (PACT)*, pp. 111–122, Sep. 2004.

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

[19] M. LaPedus. "Intel tips teraflops programmable processor," *EETimes*, Sep. 26 2006.

[20] N. Muralimanohar and R. Balasubramonian. "Interconnect Design Considerations for Large NUCA Caches," *Proc. Int'l Symp. Computer Arch. (ISCA)*, June 2007.

[21] D. Newell. "Workloads, Scalability, and QoS Considerations in CMP Platforms," Keynote, *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.

[22] M. K. Qureshi and Y. N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Run-Time Mechanism to Partition Shared Caches," *Proc. Int'l Symp. Microarchitecture (MICRO)*, pp. 423–432, Dec. 2006.

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

[24] 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.

[25] SPEC. `http://www.specbench.org`.

[26] T. Takayanagi *et al.* "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE J. Solid-State Circuits (JSSC)*, Jan. 2005.

[27] B. Verghese, A. Gupta, and M. Rosenblum. "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *Proc. Int'l Conf. Arch. Support for Prog. Lang. and Operating Systems (ASPLOS)*, pp. 181–192, Oct. 1998.

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