Taming Single-Thread Program Performance on Many Distributed On-Chip L2 Caches

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

Abstract

This paper presents a two-part study on managing distributed NUCA (Non-Uniform Cache Architecture) L2 caches in a future manycore processor to obtain high single-thread program performance. The first part of our study is a limit study where we determine data to cache slice mappings at the memory page granularity based on detailed inter-page conflict information derived from program's memory reference trace. By considering cache access latency and cache miss rate simultaneously when mapping data to L2 cache slices, this "oracle" scheme outperforms the conventional shared caching scheme by up to 208% with an average of 45% on a sixteen-core processor. In the second part of the study, we propose and evaluate a dynamic cache management scheme that determines the home cache slice and cache bin for memory pages without any static program information. The dynamic scheme outperforms the shared caching scheme by up to 191% with an average of 32%, achieving much of the performance we observed in the limit study. We also find that the proposed dynamic scheme adapts to multi-programmed workloads' behavior well and performs significantly better than both the private caching scheme and the shared caching scheme.

1. Introduction

Integrating multiple processor cores has become a clear and natural architectural choice of mainstream microprocessors [4, 18, 25]. For one, Moore's law is valid and will be so for the foreseeable future [14]. For another, 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 has become the single most critical design constraint [6, 14]. It is now conventional wisdom to double the number of processor cores on a chip with each silicon technology generation [2].

With no sign of dramatic collapse of the memory wall, the on-chip memory hierarchy design and management will remain a challenge in the "manycore processor" era. Traditionally, different layers in the on-chip memory hierarchy are cascaded *vertically*, and the trade-offs are made between

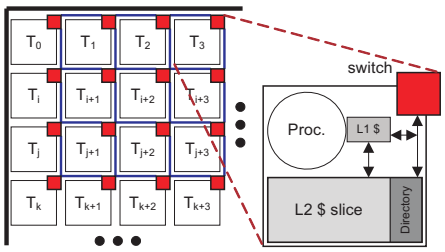

Figure 1. A tiled manycore processor architecture.

the neighboring layers in terms of cache hit rate, hit latency, miss penalty, and bandwidth [11]. When there are many processor cores and lower-level cache slices linked together via an on-chip interconnection network, a *horizontal* hierarchy will begin to form. For instance, non-uniformity in L2 cache access latencies will be inevitable [12, 16]. New innovative design strategies to manage complex on-chip memory hierarchy will be needed for future manycore processors.

Currently practiced L2 cache management approaches, namely *private caching* and *shared caching*, do not scale with the increased amount of on-chip resources. Let's consider a tiled manycore processor, depicted in Figure 1. A private cache design does not directly use the large aggregate on-chip caching capacity, wasting precious resources when there are fewer programs running than the number of cores. On the other hand, a conventional fine-grained shared cache design will experience an increased access latency if designed blindly as we scale the processor. In a 1024-core processor, for example, the average L2 cache hit latency can become as large as 180 cycles; the figure is close to the main memory access latency in 1990's!

In this paper, we are mainly concerned with the single-thread program performance on distributed L2 caches in a future manycore processor and present a two-part study done with a view to achieving more scalable single-thread program performance. More specifically, we develop and evaluate a dynamic *program location aware data placement scheme* on a simple shared cache, extended with the support for flexible data mapping at the memory page granularity [9]. While much work has been done to optimize the performance of parallel workloads, we believe single-thread programs will remain an important branch of applications in

the future and focus in this paper on improving their performance. We note that we can apply the ideas of this paper to parallel workloads and leave this as a future work.

The results we obtain in this study lead to several interesting insights: (1) When many distributed L2 caches are used, cautious data placement is crucial to single-thread program performance; (2) In a manycore processor design, the on-chip network delay is becoming a nontrivial performance-limiting factor, which, if not handled properly, will lead to significantly degraded performance; and (3) As the number of cores increases, a flexible L2 cache management framework will be beneficial and of growing importance.

The rest of this paper is organized as follows. Section 2 briefly introduces the current L2 cache management approaches and summarizes recent related work. Section 3 and 4 describe in detail the profile-driven data mapping scheme and the proposed dynamic cache management scheme, respectively, followed by their quantitative evaluation in Section 5. Conclusions and future work are given in Section 6.

2. Related Work

There are two baseline L2 cache design and management approaches in current-generation multicore processors: *private caching* and *shared caching* [1, 10, 13, 18, 20, 25, 28]. Previous work indicates that neither a pure private design, nor a pure shared design, achieves optimal performance under different workloads [9,12,21,31]. Researchers have thus examined optimizations to balance between latency (by improving data locality) and cache miss rate (by utilizing cache capacity more efficiently).

Zhang and Asanović [31] proposed *victim replication* based on a shared L2 cache organization, where each L2 cache slice can receive a remote cache line replaced from its local L1 cache as well as the designated cache lines. Essentially, L2 cache slices provide a large victim caching space for the cache blocks whose home is remote so that data locality (*i.e.*, L2 hit latency) can be improved.

Chishti *et al.* [8] proposed a cache design called *CMP-NuRAPID* having a hybrid of private per-processor tag arrays and a shared data array. 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 much more complex cache management hardware.

Chang and Sohi [7] 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 data replication and achieve a higher cache hit rate. However, the complex central directory is not a scalable approach for large-scale multicore processors.

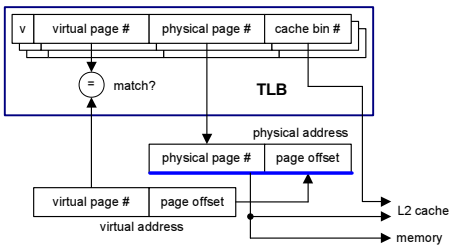

Figure 2. TLB extended with a cache bin number.

Finally, Cho and Jin [9] proposed an on-chip cache management framework where memory data can be dynamically placed into any cache slice. By increasing the data mapping granularity from memory block to memory page, they showed that the OS memory management mechanism can be conveniently extended to handle the task of on-chip L2 cache management. This work however does not study how to achieve high program performance when such a flexible data mapping mechanism is given. Our work uses a similar data mapping mechanism, based on which we investigate effective mapping policies to achieve high performance.

3. Static 2D Page Coloring

In this section, we describe our "oracle" scheme–*static two-dimensional (2D) page coloring*, where data to cache bin mappings are determined before program execution. A cache bin is a smallest group of cache sets which would hold an entire memory page. The number of cache bins in a cache is simply the cache size divided by the product of the page size and the associativity. Our off-line algorithm greedily selects a cache bin (among all cache slices) for a page based on detailed inter-page conflict information derived from a program's memory reference trace.

We call this scheme "2D" page coloring because choosing a target cache bin decides not only the color within a cache slice, but also the cache slice itself, which in turn determines the program-data distance. While collecting and analyzing detailed traces for every program might be impractical, taking this oracle approach provides valuable insights into the "ideal" program performance on distributed shared caches, by optimizing cache access latency and miss rate together. It is also noted that the performance of the proposed profile-driven 2D page coloring places an upper bound on the performance achieved by aggressive static compiler analysis [5]. We assume that the cache bin number assigned to a page can be directly derived from the physical page number or retrieved from the TLB [9], as shown in Figure 2.

The static 2D page coloring scheme consists of three phases: *trace generation*, *trace analysis* and *generating mappings between pages and cache bins*. In the trace generation phase, memory references of the target program are

```
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 3. The algorithm to extract conflict information from a reference trace.

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. 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 [24]. The algorithm is sketched in Figure 3.

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 target in this phase is then 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. Given inter-page conflict information in 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:

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

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. NoCDelay(C) denotes the on-chip network transmission latency. 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,C) is an estimated value, we introduce a parameter α to mitigate the inaccuracy. α can have a value ranging from 0 to 1 and 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 by only considering the aggregate on-chip miss penalty and ignoring the on-chip network latency and cache hit time. 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 [9].

4. Dynamic 2D Page Coloring

In this section, we present a dynamic 2D page coloring scheme which optimizes both cache hit latency and cache miss rate. The dynamic page placement exploits the on-line cache resource usage information to select a home cache bin having the smallest total expected cache access cost for a new page. The key design issues for the proposed scheme are: (1) what information to collect and (2) how to compute the expected cache access cost with the information.

What run-time information do we collect? Selecting a good home cache bin for a page involves estimating the combined cost of cache access latency and cache misses, given that the page is placed in the cache bin. The cache access latency is simply the distance between the program and the cache slice. The cache miss estimation process, however, requires that we continuously monitor the *cache bin hotness*. Because accurately predicting future page access behaviors is difficult, we base our initial placement decisions on the most recent hotness information of the cache bins in consideration. There are potentially many different ways to assess the hotness of a cache bin. For example, one can measure the cache pressure by counting the number of hot pages [9] or derive the cache utility by tracking the LRU stack positions being touched [22]. In this study, we examine an alternative method which simply counts the number of misses (BinMiss()) and accesses (BinAcc()) at each cache bin because the values are directly used in our cost estimation process.

How do we determine a home tile for a page? The actual cost of placing a new page to a cache bin C is computed on-line using the following function:

$$\begin{aligned} \text{Cost(C)} &= \frac{\text{BinMiss(C)}}{\text{BinAcc(C)}} \times \text{MemLatency} \\ &\quad + (\text{L2Latency} + \text{NoCDelay(C)}), \end{aligned} \quad (2)$$

which is simply the average L2 cache access latency to the bin C. The above cost function considers only the current state of cache bins because at the time of initial page placement there is little information about the page usage. Our experimental results indicate that page placement decisions guided by the proposed cost function are effective. In order

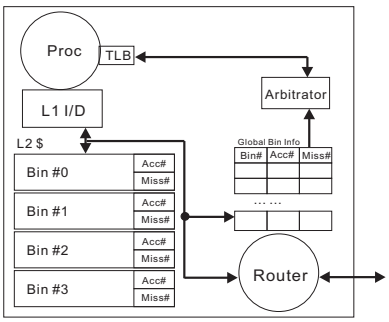

Figure 4. Microarchitectural support for dynamic 2D page coloring.

to reflect program's phase changes, BinMiss() and BinAcc() need be continuously decayed. The decay period (T_{decay}) and decay method are a parameter to the overall cache management scheme. To determine a home tile for a page, we simply choose a cache bin having the minimum cost.

4.1. Architectural support

Figure 4 depicts a block diagram that captures the hardware support for the proposed dynamic 2D page coloring scheme. First, each bin in the L2 cache is augmented with a pair of counters, BinMiss() and BinAcc(). They are updated on every L2 cache access. Second, we add a *global bin usage table*, which provides information about how cache bins are used across the chip. The number of entries in this table is equal to the total number of tiles, since only the optimal bin from each tile is recorded. We keep in the table such information as BinIndex, BinMiss() and BinAcc(). It is noted that the information carried in this table is only a hint to our algorithm; inconsistent and imprecise information is still acceptable. Henceforth, the global bin usage table can be updated from time to time, via periodic messages between tiles, or even utilizing piggybacked information in regular messages in order to reduce synchronization and communication overhead.

To make a decision about the home tile for a page, calculations outlined in Equation (2) have to be performed efficiently. As the cost depends only on the local and remote bin information, calculation is done whenever there is an update in BinMiss() and BinAcc() in the global bin usage table. The cache bin having a minimum cost is continuously tracked so that it can be used whenever there is a bin allocation request. It is noted that only one comparison operation is needed on each BinMiss() and BinAcc() update, in order to keep the "current minimum cost bin" up-to-date.

We find that the cost of counters to collect cache bin usage information is modest. Assuming a counter width of 16 bits, the cost for each bin is only 4 bytes. For a 16-tile processor with 256KB 4-way associativity L2 cache, only 512B storage is needed to keep track of necessary information, assuming that the page size is 8KB.

5. Quantitative Study

5.1. Experimental setup

We extend the SimpleScalar tool set (v3.0) [3] to model a tile-based multicore processor on a 2D mesh, similar to Figure 1(a). The baseline processor configuration is (4×4) tiles. Each tile has a processor with private L1 I/D caches and a globally shared L2 cache slice. The modeled processor is a 4-issue out-of-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 the 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 select 11 integer and 7 floating-point programs from the SPEC2k CPU suite [27] as workloads. After fast-forwarding the initial phase and having a warm-up period, we collect statistics during a period of 800M instructions. For the sake of consistency, we always report our result relative to the baseline shared caching scheme ("SharedBase") and simple page coloring with no profile information.

5.2. Results

Static 2D page coloring. Figure 5 shows how a change in α (in Equation (1)) affects the page mappings and cache access behaviors. Some programs with similar results are removed from this figure due to the space limitation. We examine 9 different values of α, ranging from 0 to 1, with a stride of $\frac{1}{8}$. When α equals to 0, the static 2D page coloring scheme simulates a private cache. On the other hand, when α is set to 1, it behaviors similar to a shared cache.

The upper graph shows the distribution of accesses going to a local or remote cache slice. It clearly shows that increasing the value of α scatters more references to a remote cache slice. The bottom graph shows the distribution of hits and misses. Programs like *gzip*, *twolf* and *art* have a very small number of misses, which is not changed much by the value of α. Much of their conflict misses are removed by static 2D page coloring ("Static2D" from now on) and unavoidable cold misses remain. *crafty* and *eon*, though not shown in the figure, have similar characteristics. As a result, they show their best performance with the small α value we examined. Other programs like *mcf*, *parser*, *vortex*, *bzip2* and *swim* exhibit a concave curve, whose peak is roughly in the middle. *mgrid* and *equake* even show an increase in the number of misses. We ascribe this situation to imperfect trace information used in our algorithm, its heuristic nature and its inability to capture program phase changes. In addition, when α equals to 0, no conflict information is considered. This may lead to nondeterministic behavior as exhibited in some programs such as the elevated miss rate in *parser*, *vortex* and *swim*. In general, a value of α which balances miss rate and

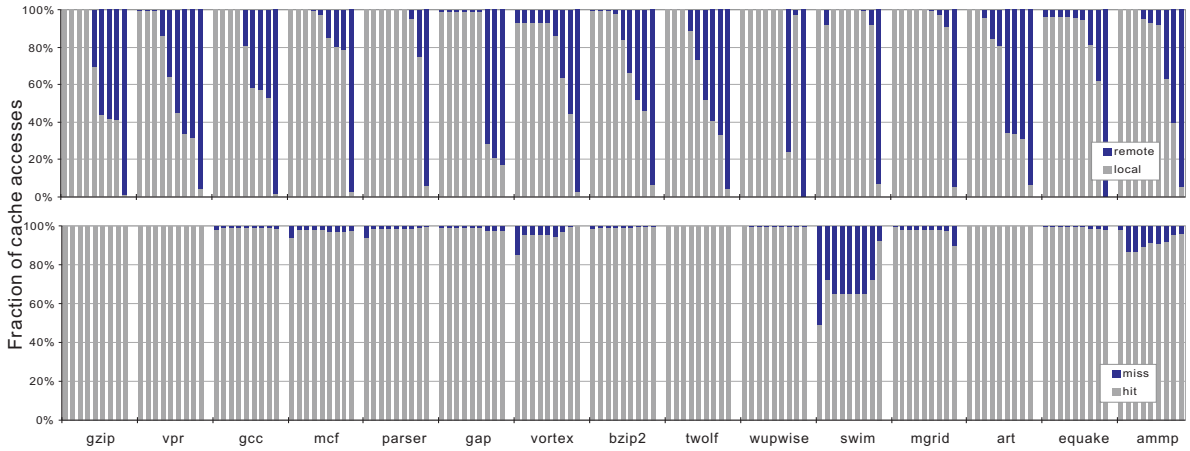

Figure 5. L2 cache access decomposition: local vs. remote (upper) and hit vs. miss (bottom) at different α values. Nine bars for each program represent cases for $\alpha = 0$, $\alpha = 1/8$, ..., $\alpha = 1$.

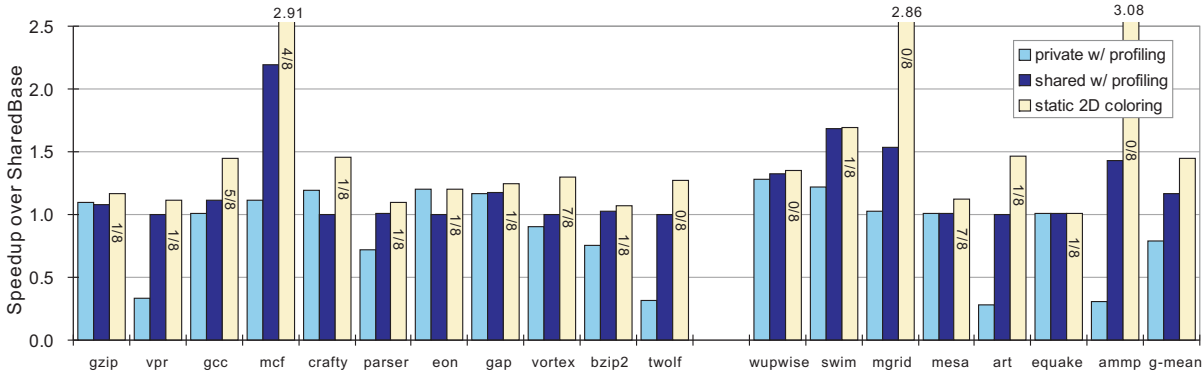

Figure 6. Performance of Private**,** Shared**, and** Static2D**. Numbers on bars show the value of α chosen.**

latency yields the best performance.

Figure 6 shows the performance of the private caching scheme, shared caching scheme and our Static2D. Since we use an aggressive profile-guided page coloring technique [24] on private caching and shared caching scheme for fair comparison, we name them "Private" and "Shared" respectively to differentiate from SharedBase.

Note that all performance numbers are normalized to that of SharedBase. 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 observe a high L2 cache miss rate in these programs, which cannot be simply compensated by L2 cache latency savings.

Shared always shows better performance than SharedBase by reducing conflict misses (but not access latency). Programs like *mcf*, *swim*, *mgrid* benefit much from the miss rate reduction and achieve over 50% performance 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

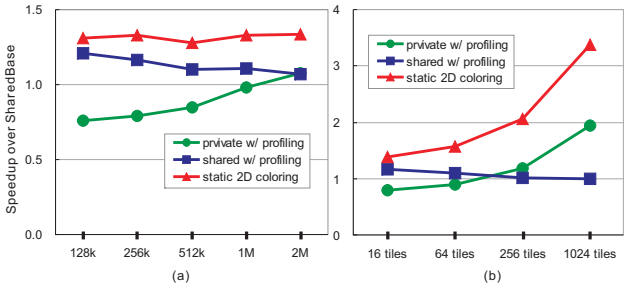

Figure 7. Average performance when (a) cache slice size is varied and (b) tile count is varied.

achieves a better miss rate than Static2D due to its fine-grained block interleaving. On average, Static2D achieves 44.7% performance improvement over SharedBase, 23.7% over Shared, and 83.2% 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

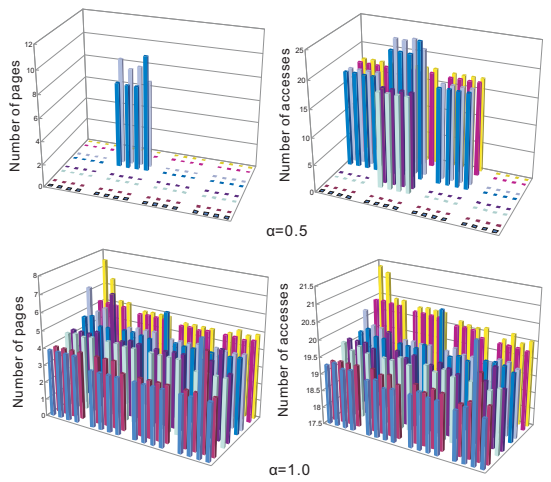

Figure 8. Page mapping and cache access distribution for *ammp* with $\alpha = 0.5$ and $\alpha = 1.0$. Each point in x-y space corresponds to a cache bin. Cache bins are clustered around a tile.

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(b)). 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.

Lastly, Figure 8 gives an example of how Static2D allocates pages to different L2 cache bins. In case of ammp, if α is set to 0.5, most pages are allocated to the local cache slice while a few pages are spread out to neighboring tiles. Access distribution exhibits similar pattern. After increasing α to 1.0, Static2D simulates Shared, except at granularity of page. Pages and memory accesses are almost evenly distributed across the whole chip. It is clearly shown that a larger α value results in more spread memory accesses among the cache bins.

Dynamic 2D page coloring. T_{decay} is an important parameter, which controls the sensitivity to the program phase change. Due to the space limitation, we present results with empirically chosen values for T_{decay} (8,192 L2 cache misses). Figure 9 shows the performance of Static2D, victim replication [31] ("VR") and the dynamic 2D page coloring scheme. Again, the results are normalized to the performance of SharedBase. VR achieves better performance than SharedBase for most of programs, except *swim* and *mgrid*, in which case, VR degrades performance considerably. This degradation is mainly caused by the interference introduced by replications. Overall, the performance improvement of VR over SharedBase is shown to be limited.

Dyn2D provides slightly lower performance than Static2D in general. This is due to the limited scope of runtime information used in Dyn2D. For *mcf* and *swim*, the dynamic scheme performs considerably worse than Static2D. The major reason for this is that the two programs access a lot of pages for a relatively small number of times per page. The dynamic scheme did not react to the page usage changes in a timely manner.

Static2D and Dyn2D perform significantly better than VR, and the improvements come from two factors: (1) Page coloring schemes place data close to the program location when it is not able to fit data in the local cache. VR, on the other hand, does not provide such benefit; and (2) Page coloring schemes provide extra benefit by minimizing miss rate through cautious data placement. VR can potentially introduce more misses, however, by increasing the local cache pressure by injecting replicas without control. Overall, the performance of the dynamic 2D coloring scheme is comparable to that of Static2D and boost the performance by 32.3% and 40.9%, respectively, compared with SharedBase. Compared to VR, the dynamic 2D coloring schemes gain 24.7% (Dyn2D).

Figure 10 takes *mgrid* as an example to show how different schemes, SharedBase, Shared, Static2D, and Dyn2D (from (a) to (d)), create changes in how frequently cache slices (grouped into *tiers*) are accessed and how often accesses hit. Tier 0 refers to the local cache slice, tier 1 refers to the four neighbors in north, south, west, and east, and so on. Compared with SharedBase, Shared does not change the access frequencies to different tiers. However, it reduces the miss rate from over 22% down to 10.4%, resulting in an 1.5× speed-up. Static2D further reduces the miss rate to 2.2%, while attracting almost all pages to the local cache slice. Lastly, Dyn2D trades access latency for an even lower miss rate, thus achieving the best performance.

Multiprogrammed workloads. We also evaluated how the dynamic 2D page coloring scheme performs when multiprogrammed workloads are run. We conducted this study on a full-system simulator built on Simics [26] modeling the same machine configurations used in previous experiments. To form workloads, we group programs into three classes, low pressure, medium pressure, and high pressure, based on their cache usage. We then select a combination of eight programs for each of the three workloads. In the first workload that we label "mix.low," we have *gzip*, *crafty*, *parser*, *eon*, *vortex*, *vpr*, *mesa*, and *swim*. The workload is designed to mimic a situation where each individual program scheduled together requires a small L2 cache space. The next work-

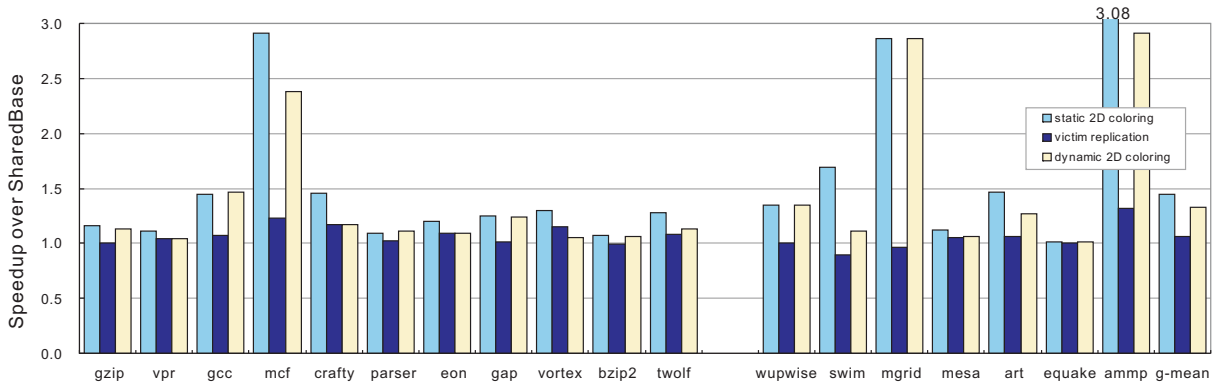

Figure 9. Program performance of the dynamic 2D page coloring schemes.

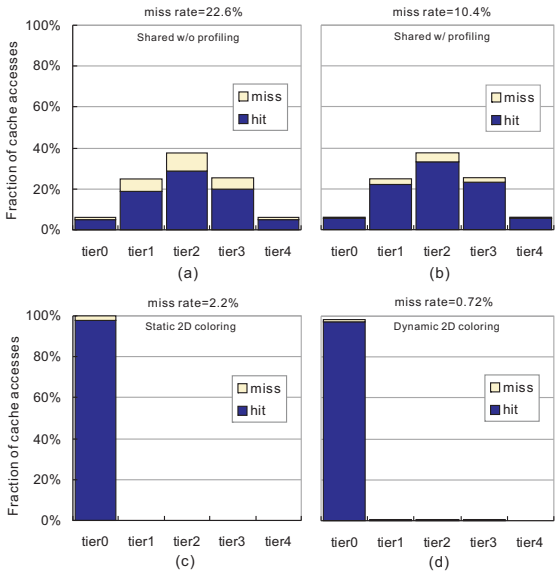

Figure 10. Cache access distribution of *mgrid* under different schemes.

load, labeled "mix.mid," has *wupwise*, *gzip*, *mcf*, *crafty*, *parser*, *mgrid*, *eon*, and *art*. Lastly, the workload "mix.high" comprises of *twolf*, *gap*, *mcf*, *wupwise*, *mgrid*, *art*, *equake*, and *ammp*.

After skipping the initialization phase of the programs, we measure the progress of each program after one simulated second. Program locations are fixed, ranging from tile 4 to tile 11. Additionally, we slightly modified the decay method in the dynamic 2D coloring scheme as follows. When we right-shift counters on every 50k cycles, we add the values in the bin miss counters to the bin access counters. This is to better distinguish between hot bins and relatively cold bins that have many cold misses. The intuition behind our method is that adding the miss count to the access count will decrease the miss rate and hence the effect of cold misses. There may be many different implementations to achieve the same effect, but we find this simple method sufficiently good in our experiments.

Figure 11 shows the performance of these workloads in terms of averaged speedup (a) and aggregate throughput (b). As expected, the performance of the private caching scheme keeps decreasing (compared with that of the shared caching) as the workloads have an increased L2 cache demand. When there is high L2 cache contention ("mix.high"), the private caching scheme performs poorly, worse than SharedBase. The dynamic 2D page coloring scheme outperforms both the private and the shared caching scheme robustly, in almost all the comparison points. In particular, it achieves an improvement of up to 26.4% using the average speedup metric and 38.6% using the aggregate throughput metric when compared to SharedBase. In comparison with the private caching scheme, the dynamic 2D scheme achieves an improvement of up to 14.3% in average speedup and 18.8% in throughput in the best case. The dynamic 2D page coloring scheme had a very slight throughput degradation than the private caching scheme for the "mix.low" workload. This is because the programs in this workload have low cache space requirement each and can run efficiently on private caches. On the other hand, our dynamic scheme introduced some interferences in allocating pages and accesses. An interesting observation is that the dynamic 2D page coloring scheme performs best in the "mix.mid" workload in both the metrics. This is because the workload provides the largest room for trade-off between miss rate and access latency, and our scheme was able to hit the right point in the trade-off span.

6. Conclusions

This research investigated the problem of how to manage distributed L2 caches on a large-scale chip multiprocessor to achieve high single-thread program performance. We make the following contributions in this paper:

- We propose a static off-line algorithm to assign a cache bin to each memory page based on detailed page conflict and access frequency information. The result we obtain using this scheme presents a relative tight limit on the performance with the shared cache hardware

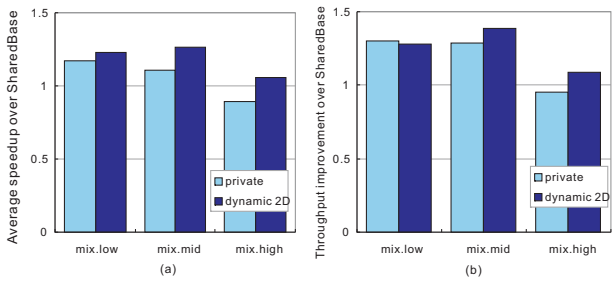

Figure 11. Performance of multiprogrammed workloads under the private caching and dynamic 2D page coloring schemes.

when the cache hit latency and the on-chip cache miss rate are optimized together via flexible data mapping.

- We propose a dynamic on-line algorithm to map pages to L2 cache bins. The proposed algorithm uses only run-time information on cache bin and page usage when selecting a target cache bin for a new page mapping. We discuss the key design issues in detail.
- We evaluate the proposed schemes and compare them with the existing shared and private caching schemes. Our quantitative study shows that the proposed schemes achieve higher performance than the existing schemes because they balance cache miss rate and cache access latency effectively. We showed that the proposed dynamic 2D coloring scheme achieves much of the performance potential identified through our limit study using the off-line algorithm.

Our future work includes (1) exploring the design space of the dynamic 2D page coloring scheme, especially for multiprogrammed workloads; (2) studying how to support a parallel application effectively; and (3) utilizing process scheduling and data placement schemes with synergy.

References

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

[2] K. Asanović *et al.* "The Landscape of Parallel Computing Research: A View from Berkeley," Technical Report UCB/EECS-2006-183, Univ. of California, Berkeley, Dec. 2006.

[3] T. Austin *et al.* "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 35(2):59–67, Feb. 2002.

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

[5] E. Bugnion *et al.* "Compiler-Directed Page Coloring for Multiprocessors," *Proc. Int'l Conf. Arch. Support for Prog. Lang. and Operating Systems (ASPLOS)*, pp. 244–255, Oct. 1996.

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

[7] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Proc. Int'l Symp. Computer Arch. (ISCA)*, June 2006.

[8] 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.

[9] 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.

[10] J. M. Hart *et al.* "Implementation of a Fourth-Generation 1.8-GHz Dual-Core SPARC V9 Microprocessor," *IEEE J. Solid-State Circuits (JSSC)*, 41(1):210–217, Jan. 2006.

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

[12] 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.

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

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

[15] R. E. Kessler and M. D. Hill. "Page Placement Algorithms for Large Real-Indexed Caches," *ACM Trans. Computer Systems*, Nov. 1992.

[16] 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. 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)*, Sep. 2004.

[18] P. Kongetira *et al.* "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] S. Naffziger, B. Stackhouse, and T. Grutkowski. "The Implementation of a 2-core Multi-Threaded Itanium-Family Processor," *Proc. IEEE Int'l Solid-State Circuits Conf. (ISSCC)*, pp. 182–183, 592, Feb. 2005.

[21] B. A. Nayfeh *et al.* "The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors," *Proc. Int'l Symp. High-Performance Computer Arch. (HPCA)*, pp. 74–84, Feb. 1996.

[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] R. M. Ramanathan. "Extending the World's Most Popular Processor Architecture," *Tech.@Intel Mag.*, Oct. 2006.

[24] 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.

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

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

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

[28] T. Takayanagi *et al.* "A Dual-Core 64-bit UltraSPARC Microprocessor for Dense Server Applications," *IEEE J. Solid-State Circuits (JSSC)*, 40(1):7–18, Jan. 2005.

[29] 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. Arch. Support for Prog. Lang. and Operating Systems (ASPLOS)*, pp. 279–289, Oct. 1996.

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

[31] 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.