Better than the Two: Exceeding Private and Shared Caches via Two-Dimensional Page Coloring

Lei Jin Sangyeun Cho

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

Abstract

Private caching and shared caching are the two conventional approaches to managing distributed L2 caches in current multicore processors. Unfortunately, neither shared caching, nor private caching guarantees optimal performance under different workloads, especially when many processor cores and cache slices are provided on a switched network. This paper takes a very different approach from the existing hardware-based schemes, allowing data to be flexibly mapped to cache slices at the memory page granularity [4]. Using a profile-guided execution-driven simulation method, we perform a limit study on the performance-optimal two-dimensional page mappings, given a multicore memory hierarchy and on-chip network configuration. Our study shows that a judicious data mapping to improve both on-chip miss rate and cache access latency results in significant performance improvement (up to 108%), exceeding the two existing methods. Our result strongly suggests that a well-conceived dynamic data mapping mechanism will achieve similarly high performance on an OS-managed distributed L2 cache structure.

1 Private Caching vs. Shared Caching

Multicore processors have hit the market at all fronts. Processors with two to eight cores are available [5, 8, 10], and they are widely deployed in PCs, servers and embedded systems. While improving program performance in general, the trend of integrating many cores on a single chip can make performance more sensitive to how the on-chip memory hierarchy is managed, especially at a lower level (*e.g., L2 caches*).

The conventional L2 cache management schemes are *private caching* and *shared caching*. In the private caching scheme, a local cache slice always keeps a copy of the accessed data, potentially replicating the same data in multiple cache slices. This replication of data results in reduced effective caching capacity, often leading to more on-chip misses. The benefit of the private caching scheme is a lower cache hit latency. On the other hand, the shared caching scheme always maps data to a fixed location. Because there is no replication of data, this scheme achieves a lower on-chip miss rate than private caching. However, the average cache hit latency is larger, because cache blocks are simply distributed to all available cache slices.

To remedy the deficiencies in the two existing schemes, many works have been done. Zhang and Asanović [11] proposed *victim replication* in a shared L2 cache organization. In their design, L2 cache slices can store a replaced cache line from their local L1 caches as well as their designated cache lines. Essentially, the local L2 cache slice provides a large victim caching space for the cache lines whose home is remote, similar to [7]. A downside of this approach happens on a local L1 cache miss or an external coherence request; both L1 cache and L2 cache (in parallel or in sequence) should be checked, because it is not readily known if a (remote) cache block has been copied in the local L2 cache slice. Chishti *et al.* [3] 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, in-situ communication, and capacity stealing. Compared with a shared cache organization, however, CMP-NuRAPID requires a more complex coherence and cache management hardware. For example, it implements a distributed directory mechanism by maintaining forward and reverse pointers between the private tag arrays and the shared data arrays. Chang and Sohi [2] 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

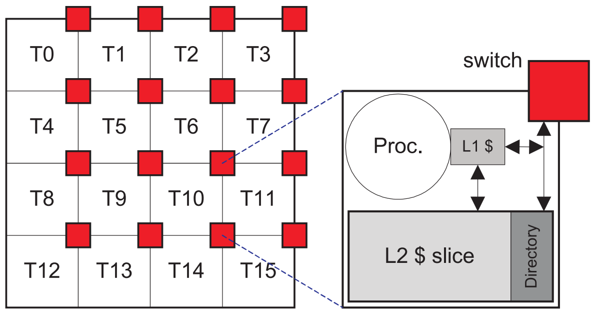
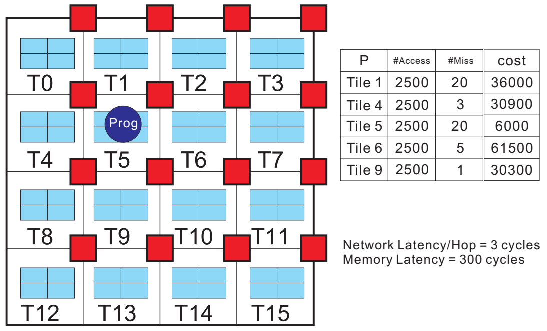


Figure 1. An example of a 16-core tile-based multicore processor. Each tile consists of a processor core, a private L1 cache, a slice of global shared L2 cache and a switch. Communication among tiles are through a mesh-based on-chip interconnection network.



P	#Access	#Miss	cost
Tile 1	2500	20	36000
Tile 4	2500	3	30900
Tile 5	2500	20	6000
Tile 6	2500	5	61500
Tile 9	2500	1	30300

Network Latency/Hop = 3 cycles
Memory Latency = 300 cycles

Figure 2. Program is running on tile 5. We only consider neighboring tiles for simplicity's sake. Network latency on each hop is 3 cycles, while off-chip memory access takes 300 cycles. Page P can be placed in any tile, with different expected resultant misses as shown in a seperate table.

global replacement of inactive data. Experimental results show that the proposed optimizations effectively limit cache block replication and thus result in a higher on-chip cache hit rate. However, the optimizations come at the expense of a more complex central directory than that of a baseline private cache design.

In summary, the central ideas found in these works are to balance between the best on-chip miss rate (*i.e.*, shared caching) and the best cache access latency (*i.e.*, private caching). Unfortunately, none of these previous works directly answers the key question: *what is the optimal point between these two opponents?* This work investigates the *optimal trade-off* between the cache access latency and the on-chip miss rate, given many distributed, non-replicating L2 cache slices. This work is based on an OS-managed distributed L2 cache structure [4] and profile-guided two-dimensional data mapping at the page granularity.

In the remainder of this paper, we will briefly discuss the OS-managed L2 cache framework first. The profile-guided two-dimensional coloring algorithm with its results are then presented.

2 OS-Managed Distributed L2 Cache

Data distribution becomes a critical performance factor in a multicore processor architecture, especially when a non-uniform latency cache architecture (NUCA) is employed. Throughout this paper, we assume a tile-based 16-core processor model as shown in Figure. 1. The data distribution granularity in a conventional shared caching method has been cache block [5, 8, 10], which is determined mainly by the bandwidth requirement. This is no longer optimal in a large-scale NUCA processor, however, where the cache access latency can vary significantly depending on the data location.

In our previous work [4], we proposed a flexible data to L2 cache slice mapping scheme at the memory page granularity with help from the OS memory management framework. The OS establishes mapping between a virtual page and a physical page upon request. At the same time, the OS assigns a home cache slice for the mapped page. Afterward, a cache miss targeting a cache block in the mapped page will bring the cache block into the designated L2 cache slice. Due to its flexibility, the proposed approach can easily implement the conventional private caching scheme, the conventional shared caching scheme, and other hybrid schemes.

One possible strategy for exploiting the flexibility is to choose between the two conventional schemes, private and shared caches, depending on the program's working set size and its sensitivity to the cache access latency. A better idea than this simple strategy is to find a mapping which takes both the cache miss rate and the cache access latency into consideration at the same time. By mapping a hot page to a cache slice near to the program, for example, a reduction of the cache access latency will be achieved. Likewise, by not allocating a set of conflicting pages into a same cache slice, a reduction of the cache miss rate can be obtained. A simple example would make this point more clear. As illustrated in Figure. 2, there is a new request to page P in tile 5. If the access information of the page is available, we can easily estimate the cost of placing the page on different tiles. In this example, tile 5 is obviously the best choice even though it has more severe cache contention. The table in Figure. 2 shows that placing the page in a neighboring

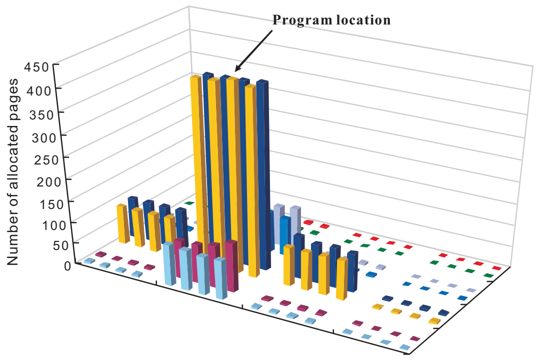


Figure 4. The number of pages mapped to different cache slices in GCC.

tile leads to fewer L2 misses; however, the interconnection network latency ($2500 \times 3\,cycles \times 4\,hops = 30000\,cycles$) overwhelms the benefit of fewer off-chip accesses.

3 Two-Dimensional Page Coloring

In order to assess the potential provided by an optimal L2 data placement, we assume an oracle memory system where we can make data placement decisions based on the memory access information of the whole program execution. The evaluation takes three steps: First, a detailed page reference trace with inter-page conflict information is collected. Second, An off-line page coloring algorithm is then applied on the collected trace. Finally, the program is re-executed with the hints fed from the second step. Our page coloring algorithm is similar to the one proposed in Sherwood *et al.* [9], which studied a profile-based coloring scheme targeting a single cache slice (“one-dimensional”). However, we consider not only colors within a cache slice but also colors among cache slices (*i.e.*, different cache slices), thus the name *two-dimensional page coloring*.

3.1 Optimal page coloring algorithm

The goal of our optimal page coloring algorithm is to find a color assignment for each page, balancing between miss rate and cache access latency for optimal resultant performance. We note that it is not practical to derive the optimal coloring decision, since the coloring problem is NP-complete [9]. Hence, we take a heuristic approach as follows.

First is to profile a target program and collect the page conflict information. Two references are deemed conflicting if referenced addresses are different from each other and mapped to the same cache line. However, we do not know whether these two references will be conflicting before their pages are allocated. Therefore we make an assumption that if two pages A and B are referenced in order and there is no reference to page B in between, these two references can cause a conflict miss. Our algorithm is based on this assumption to calculate the number of misses between any pair of pages. Note that this number tends to over-estimate the actual number of misses since two potentially conflicting pages may not actually interfere with each other if they are put in different colors, or references are made to different portions of the same color. The algorithm is given below.

```
$While\ trace\ is\ not\ empty$ {
    $PN = page\ number\ of\ next\ reference$
    $Pi = the\ array\ index\ of\ page\ PN$
    $for(i = 0;\ i < total\ number\ of\ pages;\ i++)$ {
        $Reference[i][Pi] = 1$
        $if\ (Reference[PI][i] == 1)$ {
            $Conflict[PI][i] = Conflict[PI][i] + 1$
            $Reference[PI][i] = 0$
        }
    }
}
```

Now, assume that the *Conflict*[.][.] matrix carries the inter-page conflict information between any two pages. Every row or column corresponds to a page. The value at the cross section of a row and a column indicates the number of conflicts between these two pages. We define the maximum number in each row as the *conflict capacity* for that page. The algorithm evaluates a cost function from a page with the largest conflict capacity and proceeds in a decreasing order. The cost of assigning a particular color *c* to a page *P* is computed by the cost function:

$$Cost(P,c) = \alpha \times Total\ conflicts(P) \times Memory\ latency$$
$$+ (1-\alpha) \times Total\ accesses(P) \times (L2\ latency + NoC\ delay(c))$$
$$Total\ conflicts(P) = \sum Conflict[Pi][Xi]$$
$$for\ any\ page\ X\ already\ mapped\ to\ c.$$

In the above, $Total\ accesses(P)$ is the total number of accesses of page P. Pi and Xi are the array indices of page P and X. $NoC\ delay$ includes routing and wire delay, and it varies depending on the relative location of the program and the assigned color. The parameter α ranges between 0 and 1, which controls the page aggregation density. With a smaller α, more weight is put on the NoC delay, thereby placing pages closer to the program location. When α is 0, the algorithm will simulate a private cache. On the other hand, with α equal to 1, the algorithm will simulate a shared cache, ignoring network latency. For each page, the cost of assigning every color is calculated and a color with the smallest cost is picked. This process is repeated

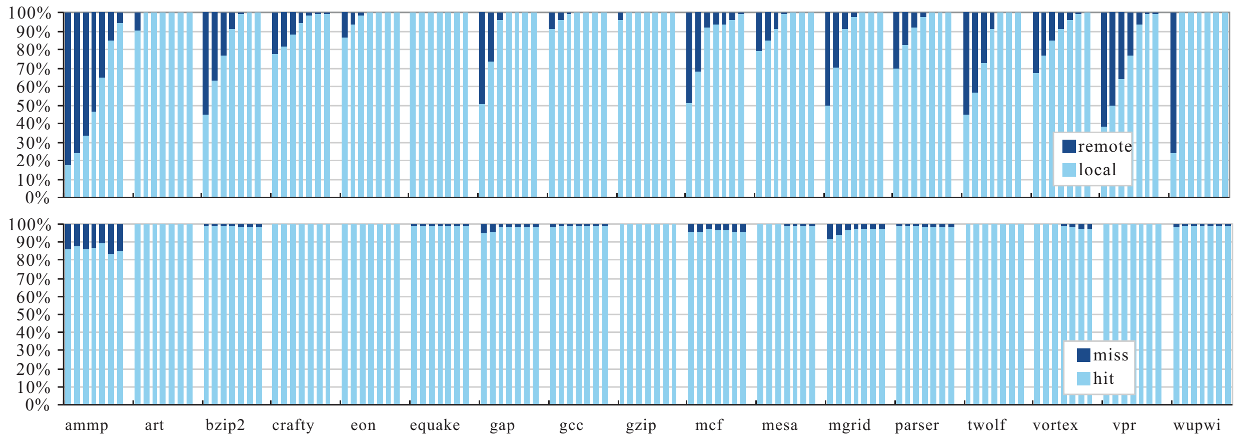


Figure 3. Proportions of local and remote accesses (upper graph) and proportions of on-chip hits and misses (lower graph). Results with α ranging from $\frac{1}{32}$ to $\frac{1}{2048}$ (from left) are shown.

until all pages are colored. The derived color assignment information is then used to direct OS page coloring.

3.2 Results

We use a simulator derived from the SimpleScalar tool set [1], which models a tile-based 16-core processor with 2-cycle 32KB L1 I/D caches and a 8-cycle 256KB shared L2 cache slice per tile. The NoC delay is 3 cycles per hop. The main memory latency is 300 cycles. In this study, we only assume a simple program on the multicore chip, without any network contention. The optimal page coloring algorithm is performed with different α values ($\frac{1}{2} \sim \frac{1}{2048}$), and the best results are chosen for presentation.

Figure. 3 shows that with a smaller α, more references become local. Also shown is that careful data placement keeps the miss rate at the same level even with a smaller α. In some cases such as *gap* and *mgrid*, the number of misses decreases with a smaller α. This may result from the greedy nature of our heuristic algorithm and the page granularity we use when computing conflict information. Figure. 4 shows an optimal page distribution for *gcc*. Clearly, most pages are allocated local to the program location. This “condensed” mapping reduces the average cache access latency tremendously while hardly impacting the miss rate.

Finally, Figgure. 5 shows the performance improvement of the profile-guided page coloring techniques (one-dimensional and two-dimensional). The baseline configuration is a shared caching scheme with conventional page coloring scheme [6]. It is clearly shown that the profile-guided two-dimensional page coloring achieves higher performance. It obtains 150.6% and 141.6% improvement for *mcf* and *mgrid*, which have relatively intense L2 accesses, with an average of 11.0% speedup over 17 programs simulated. Compared with shared caching with the same profile-guided page coloring, it achieves up to 107.6% improvement (*mgrid*), with an average of 6.4%. The results under the private caching scheme is not shown because they are generally worse than the shared caching results. We observed that the improvements achieved with the two-dimensional page coloring over the private caching is over 130% on average, which is significant. In certain cases, such as *art*, *twolf* and *vpr*, the improvement is as high as $3\times$ to even $6\times$.

4 Final Remarks

This paper presented a study on optimal data to cache slice mapping under an OS-based distributed L2 cache management framework. When pages are optimally mapped, the observed performance improvement is 11% (over shared caching) and 130% (over private caching) on average. The result suggests that there is a large room for performance improvement over the existing two hardware-based cache management schemes. It also suggests that a well-conceived dynamic data mapping scheme (*e.g.*, *without static profile information*) will achieve similarly high performance on an OS-managed distributed L2 cache organization. Naturally, we have the following directions for future work. First, dynamic page mapping algorithms to achieve high performance and low power will be investigated. Next, we will consider multiprogrammed, multithreaded, and server workloads.

References

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

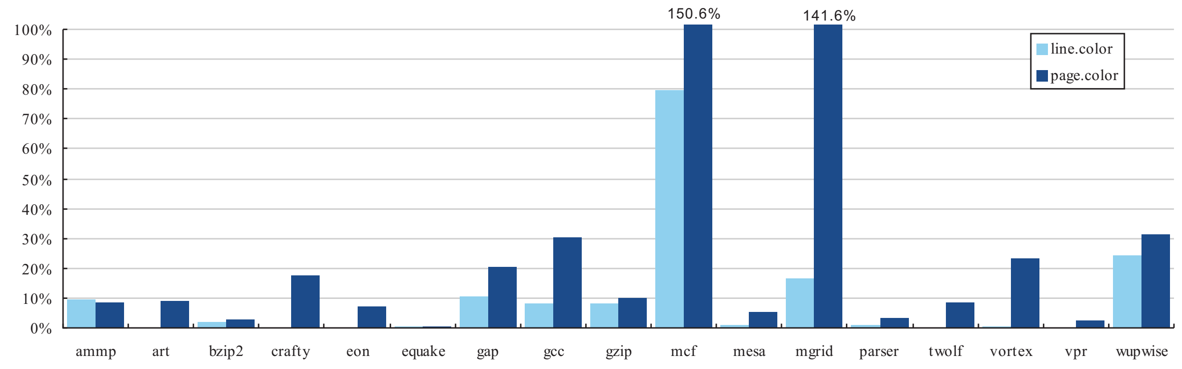


Figure 5. Performance improvement of profile-guided coloring schemes over a conventional shared caching scheme with simple page coloring.

2002.

[2] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors," *Int'l Symp. Computer Architecture*, June 2006.

[3] Z. Chishti *et al.* "Optimizing Replication, Communication, and Capacity Allocation in CMPs," *Int'l Symp. Computer Architecture*, June 2005.

[4] S. Cho and L. Jin. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," *Int'l Symp. Microarchitecture*, Dec. 2006.

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

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

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

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

[9] T. Sherwood *et al.* "Reducing Cache Misses Using Hardware and Software Page Placement," *Int'l Conf. Supercomputing*, June 1999.

[10] J. M. Tendler *et al.* "POWER4 System Microarchitecture," *IBM J. Res. & Dev.*, 46(1):5–25, Jan. 2002.

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