Runtime 3-D Stacked Cache Management for Chip-Multiprocessors

Jongpil Jung
Dept. of EE, KAIST
Daejeon, Korea
jpjung08@vslab.kaist.ac.kr

Kyungsu Kang
Dept. of EE, EPFL
Lausanne, Switzerland
kyungsu.kang@epfl.ch

Giovanni De Micheli
Dept. of EE, EPFL
Lausanne, Switzerland
giovanni.demicheli@epfl.ch

Chong-Min Kyung
Dept. of EE, KAIST
Daejeon, Korea
kyung@ee.kaist.ac.kr

***Abstract*— These-dimensional (3-D) memory stacking is one of the most promising solutions to memory bandwidth problems in chip multiprocessors. In this work, we propose an efficient runtime 3-D cache management technique which takes advantage of the lower latencies through vertical interconnect as well as the runtime memory demand of applications which varies dynamically with time. Experimental results show that the proposed method offers performance improvement by up to 26.7% and on average 13.1% compared with the private cache organization.**

I. INTRODUCTION

Chip-multiprocessors (CMPs) are quickly becoming a mainstream microprocessor as they overcome limitations of modern super-scalar uniprocessor by exploiting thread-level parallelism among multiple cores [1]. In the meantime, as the number of cores integrated in a chip increases, the on-chip interconnects that exchange data between caches and cores are to be longer and, thus, represent a major bottleneck with regard to both power and performance. Various technologies have been explored to address the interconnect problem such as the use of non-uniform cache architecture (NUCA) [2] and/or that of three-dimensional (3-D) memory stacking.

In the recent years, 3-D memory stacking has received a great attention since it resolves the memory bandwidth challenges of 2-D integration by stacking cache memory onto a multiprocessor die using very dense through-silicon vias (TSVs) [3][4]. Fig. 1 shows a high-level view of a 3-D CMP and its vertically stacked cache. Each processor core has fast and large-bandwidth access to the cache banks directly stacked on it (i.e., local cache banks) though TSVs. The processor core can also address cache banks stacked on the other processors (i.e., remote cache banks), but corresponding memory transaction will have to be transported through a horizontal on-chip interconnect fabric, i.e., crossbar switch [5]. Such communication though a crossbar switch makes remote cache bank access even longer than local cache bank access because of the non-negligible latency caused by protocol translation and limited bandwidth in the crossbar switch.

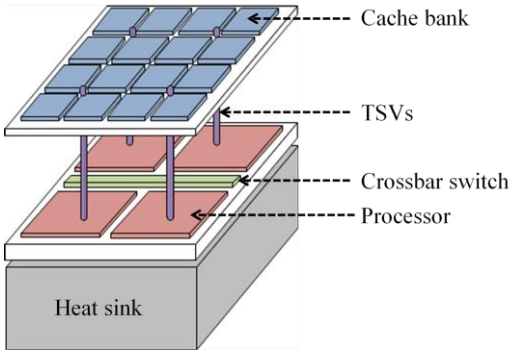

Figure 1. Target architecture of a 3-D CMP and its vertically stacked cache

Proposals for the stacked cache memory hierarchy of 3-D CMPs have borrowed from the memory hierarchies of traditional multiprocessors. I.e., the stacked cache can be either private or shared. A shared cache organization is preferable if reducing the collective number of cache misses (i.e., off-chip memory accesses) is important, whereas an organization by using local cache banks as private caches of each core is preferable if reducing the average cache access latency, which might be longer due to remote cache accesses, is important. The main contribution of this paper is the development of an optimization method specialized 3-D CMP that achieves the benefits of both private and shared cache design. Using shared cache organization as the basic design point, cache bank partitioning [6] based on application's utility for the cache resource (e.g., the amount of cache misses with respect to the assigned cache capacity) is performed to reduce the collective number of cache misses. In order to reduce remote cache accesses for each core, power-gating of remote cache banks is performed based on memory access behavior of applications which dynamically changes at runtime. We demonstrated that such an online power-gating method gives both the capacity of shared cache and the latency of private cache in Section V.

II. MOTIVATION

Fig. 2 (a) shows a 3-D CMP consisting of two cores each of which has four stacked L2 cache banks on it and connects them through TSVs. Let us assume that capacity of each cache bank is 256KB and the cores are located next to the heat sink since cores are major heat sources. We assume that *art* and *gzip*, which are benchmarks in *SPEC2000* [7], are assigned to

cores 1 and 2, respectively. Cache misses per instructions with respect to the assigned cache capacity are shown in Fig. 3.

Table I shows performance results of three different cache management policies, i.e., private cache (*PRIVATE*), shared cache with cache bank partitioning (*CBP*), and shared cache with cache bank partitioning and power gating (*CBP&PG*). In Table I, the first and second columns show the number of cache misses per instruction and the average cache access latency during application execution. In order to evaluate the system performance, we use the *fair speedup* (FS) as the metric in (1). FS is the harmonic mean of normalized *instructions per cycle* (IPC). *M* is the number of cores.

$$\mathrm{FS} = \mathrm{M} \,/\, \left(\sum_{i=1}^{M} \frac{IPC_i(base)}{IPC_i(scheme)}\right) \qquad (1)$$

In case of *PRIVATE*, the cache banks stacked on each core are used as a private cache for each core. This cache management gives the lowest value in average cache access latency because there is no remote cache banks accesses through a crossbar switch as shown in Table I. However, due to inefficient use of the cache capacity (i.e., the same amount of assigned capacity to each core without considering application's utility for the cache resource), such a design result in many off-chip memory accesses, which sharply degrades the system performance.

In case of *CBP*, the size of cache capacity (i.e., the number of cache banks) for each core is determined based on the cache utility shown in Fig. 3 so that the number of cache misses is minimized. The result of cache partition is shown in Fig. 2 (b). Core 1 (*art*) and core 2 (*gzip*) have 6 and 2 cache banks, respectively. The change of cache capacity assigned to each core can be adapted by increasing or decreasing the

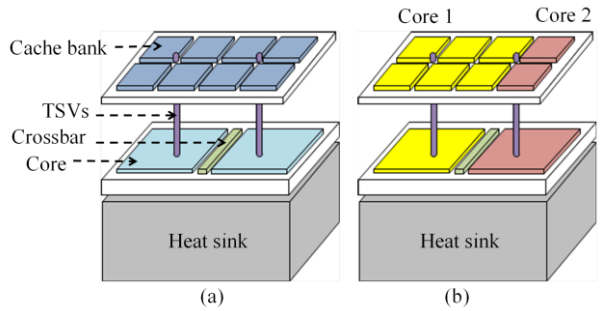

Figure 2. (a) A motivational example of 3-D CMP, (b) The results of cache bank partitioning

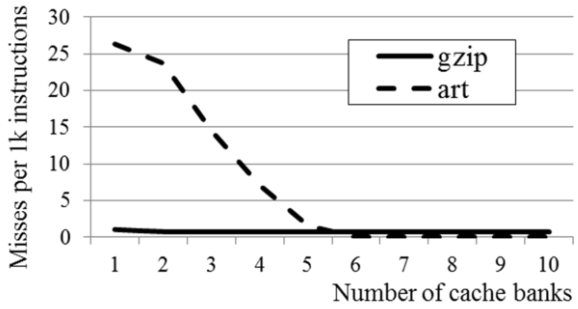

Figure 3. Cache misses per instruction for *gzip* and *art* with respect to the assigned cache capacity

TABLE I. RESULTS OF THE MOTIVATIONAL EXAMPLE

Method	Cache misses rate (#miss/ #instruction)	Avg. cache access latency (cycle)	*Fair. Speedup*
PRIVATE	0.1127	4.0	1.00
CBP	0.0508	6.0	1.12
CBP&PG	0.0777	4.9	1.32

number of ways per cache set but keeping the number of cache sets fixed. Obviously, the potential benefit of *CBP* is exploiting the cache-size sensitivity of the applications that are to be run. However, this approach makes the average cache access latency longer than that of *PRIVATE* because of the overhead for looking up all the banks assigned to each core, which might include remote cache banks accesses (caused from core 1 in the motivational example).

The basic idea of our method is exploiting runtime behavior of memory accesses which dynamically changes at runtime. Fig. 4 shows the average memory access latency of core 1 (*art*) with respect to the process time interval when the assigned cache capacity to core 1 is 1MB (i.e., four cache banks). Note that the large amount of cache capacity is not needed all the time during the application execution. By power-gating the remote cache banks assigned to core 1, when being unnecessary during application execution, *CBP&PG* reduces the number of remote cache bank accesses. As shown in Table I, the average cache access latency of *CBP&PG* is very close to that of *PRIVATE* while the number of cache misses is almost same as that of *CBP*. Thanks to the runtime cache management, *CBP&PG* gives 32% and 17% performance improvement in terms of *fair speedup* compared with *PRIVATE* and *CBP*, respective.

III. PROBLEM DEFINITION AND SOLUTION OVERVIEW

Our paper focuses on a 3-D CMP where last-level cache memory is stacked as shown in Fig. 1. In the system, cache bank partitioning and per-bank power-gating are dynamically performed at runtime. We assume that the utility of applications for cache resource is given by a static analysis. Our problem is to find: 1) the number of cache banks assigned to each core and 2) power-gating schedule of remote cache banks of each core such that the instruction throughput of the system is maximized. Instruction throughput of a multi-core system is defined as the total number of instructions executed by all the cores per cycle.

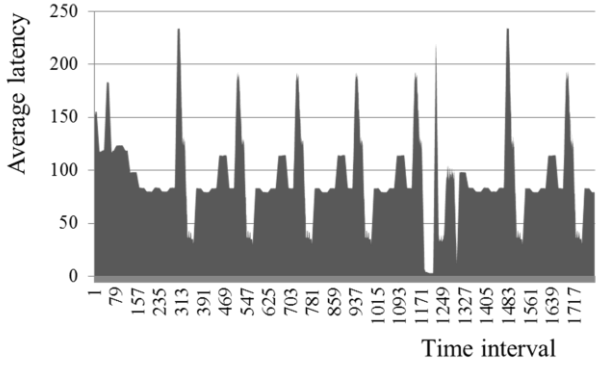

Figure 4. Average memory access latency of *art* benchmark with respect to the progress time interval when the assigned cache capacity is 1MB

Algorithm: Overall flow

1: **for** from i=1 to i=M **do**
2: **if** (start or end of application) **then**
3: Cache bank partitioning (Section VI.A)
4: **end if**
5: **else if** (elapsed time = 1ms)
6: Runtime application profiling and estimation (Section VI.B)
7: Decision of power-gating of remote cache banks (Section VI.B)
8: **end if**
9: **end for**

Figure 5. Overall algorithm flow of the proposed method

Fig. 5 shows the overall flow of the proposed method. Line 2 – 4 present cache bank partitioning which is invoked at the start/end of applications running on cores. Line 5 – 8 present online power-gating of remote cache banks which is invoked at every 1ms. Runtime application profiling and estimation step (line 6) measures runtime information (e.g., the number of cache misses) of application running on each core, updates the historical runtime information, and estimates the runtime information for the next time interval of application execution. Based on the estimated runtime information, the proposed algorithm determine whether performing power-gating of remote cache banks or not.

IV. Runtime Cache Management Algorithm

A. Cache Bank Partitioning

As shown in Fig. 3, the number of cache misses is presented as a discontinuous function with respect to the assigned cache capacity. That means an exhaust search is necessary to find performance-maximal cache bank partitioning which minimizes the total number of cache misses. However this function is also presented as a non-increasing function. Thus, dynamic programming algorithm can be adapted by repeating de-allocation of one cache bank at each algorithm step such that the increased number of cache miss ratio at each algorithm step is minimized. Fig. 6 shows the dynamic programming algorithm. The de-allocation step (line 6 – 11) is repeated until the total number of cache banks assigned to cores is equal to the maximum allowable number of cache banks (i.e., $M \cdot B$).

B. Decision of Power-gating of Remote Cache Banks

As shown in Fig. 4, memory access demand of an application shows some runtime phases with respect to time intervals. That means the cache capacity needed to cores also varies with time. Thus, during program execution, when the needed cache capacity to a core is low, power-gating of the remote cache banks reduces the average cache access time by not accessing the remote cache banks though the crossbar switch. In order to determine cores whose remote cache banks are power gated or turned on, the runtime algorithm shown in Fig. 7 is invoked at every time interval.

For power-gating of remote cache banks of each core, the memory access penalty is estimated (line 9 – 10). Thus, if the gain resulting from power-gating is more than the given threshold, the remote cache banks are power-gated (line 11 – 13). The method for estimating the number of cache misses (line 7) as a result of power-gating of the remote cache banks is derived from Suh *et al.* [8] and works as follows. If there is

Algorithm: Cache bank partitioning

1: B = the number of local cache banks
2: M = the number of cores
3: $Nbank_i = M \cdot B$
4: $TotalBank = \sum_{i=1}^{M} Nbank_i$
5: **while** ($TotalBank \neq M \cdot B$) **do**
6: **for** from i = 1 to M **do**
7: $Miss\_Inc_i = MissRate\ (Nbank_i - 1) - MissRate\ (Nbank_i)$
8: **end for**
9: Find i which has the minimum value of $Miss\_Inc_i$
10: $Nbank_i = Nbank_i - 1$
11: $TotalBank = TotalBank - 1$
12: **end while**

Figure 6. Dynamic programing algorithm for cache bank partitioning

Algorithm: Power-gating of the remote cache banks

1: M = the number of cores
2: RCA_i = remote cache access latency (cycle) from core i
3: LCA = local cache access latency (cycle)
4: MA = off-chip memory access latency (cycle)
5: Hit_i ($Miss_i$) = the number of cache hits (misses) from core i
7: $eHit_i$ ($eMiss_i$) = the estimated number of cache hits (misses) from core i when the remote cache banks are power-gated
8: **for** from i = 1 to M **do**
9: $New_i = \left(\frac{1}{W} \cdot \sum_{k=1}^{W} eHit_i[n-k]\right) \cdot LCA + \left(\frac{1}{W} \cdot \sum_{k=1}^{W} eMiss_i[n-k]\right) \cdot MA$
10: $Old_i = \left(\frac{1}{W} \cdot \sum_{k=W+1}^{2W} Hit_i[n-k]\right) \cdot RCA_i + \left(\frac{1}{W} \cdot \sum_{k=W+1}^{2W} Miss_i[n-k]\right) \cdot MA$
11: **if** ($Old_i - New_i > Thres\_CA$) **do**
12: Power-gating of the remote cache banks of core i
13: **end if**
14: **end for**
15: SC_i = the number of stall cycles of core i
16: **for** from i = 1 to M **do**
17: $New_i = \frac{1}{W} \cdot \sum_{k=1}^{W} SC_i[n-k]; \quad Old_i = \frac{1}{W} \cdot \sum_{k=W+1}^{2W} SC_i[n-k]$
19: **if** ($New_i - Old_i > Thres\_SC$) **do**
20: Turning on the remote cache banks of core i
21: **end if**
22: **end for**

Figure 7. Runtime algorithm for power-gating of remote cache banks

a hit in each cache bank for the requesting core, a counter is increased for that core. This counter represents the number of cache misses that would have occurred if the cache bank is power gated. For turning on remote cache banks of each core, the number of stall cycles is measured at every time interval. Thus, if there is huge change in the number of stall cycles from core *i* which is performing power-gating of remote cache banks, the remote cache banks are turned on (line 19 – 21).

In order to follow the runtime phases of application memory demand and avoid unnecessary power-gating, time-invariant *moving average* (MA) filter [9] is used. When the window size is W and the n^{th} value is $v[n]$, n+1th value is to be $v[n+1] = (1/W) \sum_{i=0}^{W-1} v[n-i]$. To measure the number of cache hits and misses, hardware performance counters are adopted as they are provided in most modern processors. The delay overhead of accessing the performance counters is assumed as negligible.

V. EXPERIMENTAL RESULT

A. Setup

We performed experiments using a 3-D multi-core system consisting of four cores in a layer and a stacked cache layer as shown in Fig. 1. A core studied in our experiments is based on the architecture of an Intel Core 2 Duo Merom processor [10] which is manufactured using 65nm technology. Each core has 32KB L1 instruction and data cache. The number of L2 cache banks directly stacked on a core is four for each cache layer. Capacity of each cache bank is 256KB which is computed with *CACTI* [11] for 65nm technology. For the fast performance simulation, modified *tsim* [12] is used which is a trace-driven multi-processor simulator. Our experiment was performed with *SPEC2000* [7], and *BioBench*. Benchmark programs are divided into two groups: computation-bounded benchmarks and memory-bounded benchmarks as shown in Table II. Based on the memory demand intensity, three combinations of benchmarks are made: computation bound benchmarks (CB), memory bound benchmarks (MB), and mixed benchmarks (Mix).

B. Results

We performed experiments with three different policies:

1) PRIVATE: This technique assumes that the stacked cache banks on each core are used as a private cache for it.

2) CBP: This technique assumes that the stacked cache banks are used as a shared cache and the amount of cache capacity is assigned to each core based on the proposed cache partitioning algorithm in Section IV.A.

3) CBP&PG: This is the proposed method which dynamically performs cache bank partitioning and power-gating of remote cache banks based on the time-varying memory access behavior.

The results normalized with respect to PRIVATE are shown in Fig. 8. The results show that the proposed method is more effective with Mix and MB benchmarks. Cache capacity given in PRIVATE method is large enough for the benchmarks in CB, thus the performance does not vary with cache capacity increase. CBP offers performance improvement up to 13.6% and on average 6.8% by exploiting memory-boundedness of benchmarks. CBP&PG shows performance improvement by up to 26.7% and on average 13.1%. Time-dependent benchmark behavior is considered in the runtime method. When a program need large cache capacity, remote cache banks are accessed to reduce cache misses. On the other hand, when the program does not need large cache capacity, the core only accesses the cache banks above the core with less access latency. Remote cache banks are power gated at the moment.

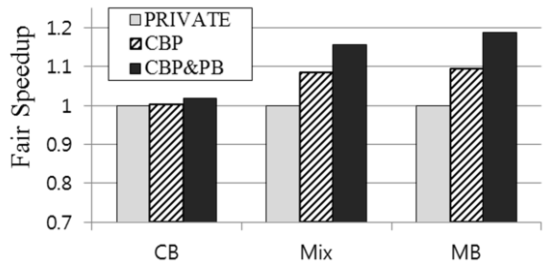

Figure 8. Comparison of performance improvement in the metric of fair speedup with benchmark combinations.

TABLE II. BENCHMARK CLASSIFICATION

Group	Benchmarks
Computation-bounded	*art, mcf, tigr*
Memory-bounded	*equake, gzip, gcc, mesa, mgrid, parser, swim, vortex, wupwise*

Overhead: When performing power-gating of the remote cache banks assigned to a core, dirty cache blocks in the cache banks which will be applied to power-gating must be written back to off-chip DRAM for data coherency. When the power-gated remote cache banks are turned on, it costs wakeup time to switch cache banks back to the active mode from the power-gating mode. According to [13], the wake-up time is negligible (i.e., four clock cycles). The area overhead of implementing power-gating technique in L2 caches is about 5% [14].

VI. CONCLUSION

In this paper, we proposed an online solution of cache management for CMP where 3-D cache memory is stacked. By exploiting the time-varying demand of cache memory for applications, the proposed method employed the benefits of private cache (i.e., lower access latency) and shared cache (i.e., higher cache capacity). The experimental results show that the proposed method achieves up to 26.7% (average 13.1%) performance improvement compared with the solution which uses the stacked cache as a private cache.

REFERENCES

[1] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating amdahl's law through epi throttling," in Proc. ISCA, 2005, pp. 298 – 309.

[2] J. Huh *et al*., "A NUCA substrate for flexible CMP cache sharing," IEEE Trans. on Parallel and Distributed Systems, vol. 18, no. 8, pp. 1028 – 1040 Aug. 2007.

[3] N. Madan *et al*., "Optimizing communication and capacity in a 3D stacked reconfigurable cache hierarchy," in Proc. HPCA, 2009, pp. 262 – 273.

[4] A. Zia *et al*., "A 3-D cache with ultra-wide data bus for 3-D processor-memory integration," IEEE TVLSI, vol. 18, no. 6, pp. 967 – 977, June 2010.

[5] AMBA AXI Protocol v1.0 Specification. http://www.arm.com/products/system-ip/interconnect/axi/index.php

[6] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches," in Int. Symp. Microarchitecture, 2006, pp. 423–432.

[7] Standard Performance Evaluation Corporation [Online]. Available: http://www.specbench.org

[8] G. Suh *et al*., "Dynamic cache partitioning for simultaneous multithreading systems," in Proc. PDCS, 2001.

[9] S.E.Said and D.A.Dickey, "Testing for unit roots in autoregressive-moving average models of unknown order," in Biometrika, 1984, pp. 599-607.

[10] Intel, "Intel core2 duo processors and Intel core2 extreme processors for platforms based on mobile Intel 965 express chipset family," Datasheet, 2008, pp. 23–40.

[11] D. Tarjan, S. Thoziyoor and N. P. Jouppi, "CACTI 4.0," HP Laboratories, Palo Alto, CA, Tech. Rep. HPL-2006-86, Jun. 2006.

[12] S. Cho *et al*., "TPTS: A Novel Framework for Very Fast Manycore Processor Architecture Simulation," In Proc. ICPP, 2008, pp. 446-453.

[13] H. Homayoun *et al*., "Multiple sleep mode leakage control for cache peripheral circuits in embedded processors," in *Proc. Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 197–206.

[14] M. Powell *et al*., "Gatedvdd: a circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. ISLPED*, 2000, pp. 90–95.