Thread Owned Block Cache: Managing Latency in Many-Core Architecture

Fenglong Song, Zhiyong Liu, Dongrui Fan, Hao Zhang, Lei Yu, and Shibin Tang

Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{songfenglong,zyliu,fandr,zhanghao,yulei,tangshibin}@ict.ac.cn

Abstract. Shared last level cache is crucial to performance. However, multi-thread program model incurs serious contention in shared cache. In this paper, to reduce average cache access latency, we propose two schemes. First, an implicitly dynamic cache partitioning scheme, *i.e.* block agglutinating. The purpose is to isolate conflicting data blocks. Second, a novel hardware buffer, called thread owned block cache, *i.e.* TOB Cache. The purpose is to store conflicting data blocks. Extensive analysis of the proposed schemes with Splash2 benchmarks and Bioinformatics workloads is performed using a cycle accurate many-core simulator. Experimental results show that the proposed schemes make conflict miss rate of shared cache reduced by 40% compared to traditional shared cache. Compared with victim cache, average load latency of shared cache and primary data cache is reduced by about 26% and 12%, respectively; primary data cache miss penalties are reduced by about 14%, and IPC is improved by 17%.

Keywords: Many-Core Architecture, Shared Cache, TOB Cache, Latency.

1 Introduction

On-chip multi-processor is a promising trend of computer architecture [3,10,11]. It is critical to design a high performance on-chip memory system with lower access latency and larger effective capacity. For the last level cache, it benefits more from shared than private design. However, it is well known that there exists conflict in table-based shared structure of computer systems, such as TLB, cache et al. Although multi-thread program model is promising in many-core architecture [10], it deteriorates conflict in shared cache. It is validated by evaluation via Godson-T many-core architecture simulator [25]. Shared cache misses described by traditional 3C classification model is shown as Fig.1. As number of threads increases, ratio of compulsory misses reduces, while ratio of conflict misses increases. Because input data sets of workloads do not change, absolute number of compulsory misses does not be reduced. The intra-thread and inter-thread conflict misses are divided further, as shown in Fig.1. It shows that as number of threads increases, inter-thread conflict misses occupy majority of conflict misses. The frequent off-chip memory accesses will incur overall performance decrement. Because there are many processing cores access it

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part I, LNCS 6271, pp. 292–303, 2010.

simultaneously, shared cache becomes a hot spot in on-chip network. If most of requests to the shared cache are forwarded to off-chip memory, it may incur tree saturation [5]. Tree saturation incurs larger average memory access latency, and the latency can be reflected by ratio of private level-one data cache miss penalty, as shown in Fig.2. Here, miss penalty means the waiting cycles when level-one cache miss occurs until the requested data block is refilled completely. It motivates us to focus on how to increase effective on-chip memory capacity and on-chip hit rate.

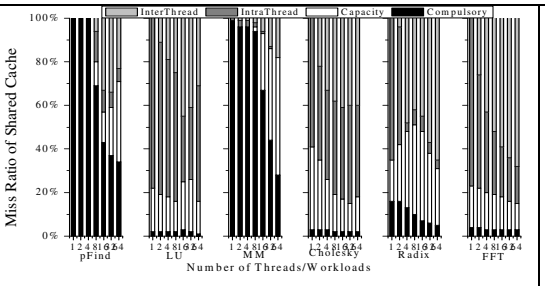

Fig. 1. Breakdown of Shared Cache Misses

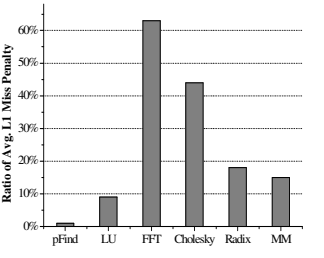

Fig. 2. Ratio of L1 Miss Penalty

(on a 64 processing cores many-core architecture with 32KB, 4-way private data cache; 16KB, 2-way private instruction cache; and 2MB non-inclusive 8-way associative shared level-two cache)
(Ratio of Miss Penalty = Cycle of Processing L1 Cache Miss / Total Thread Executing Cycle)

In this paper, we make following contributions. Firstly, we present a dynamic cache partitioning scheme, *i.e.* block agglutinating. The purpose is to partition shared cache space implicitly and isolate conflicting data blocks between different threads. Secondly, we propose a dedicated hardware buffer, named thread owned block cache, *i.e.* TOB Cache. The purpose is to reduce off-chip misses via migrating conflicted data blocks. Experimental results via a cycle accurate many-core processor simulator show effectiveness of the proposed schemes, and performance improvement compared with victim cache.

The remainder of the paper is organized as follows. Section 2 elaborates the proposed block agglutinating partition scheme and TOB cache. Section 3 presents details about our simulation platform and benchmarks, and experimental results are analyzed in Section 4. Related work is described in Section 5, followed by our conclusions in Section 6.

2 Cache Management Policy

We assume a non-preempted multi-thread execution model, so we use processing core and thread without distinguishing in following sections.

2.1 Block Agglutinating

The purpose of block agglutinating scheme is to eliminate conflicts in shared cache [16]. The scheme divides shared cache capacity dynamically when making the decision

of cache block placement and replacement. In this way, a processing core occupies some blocks of shared cache. The occupied blocks cannot be replaced by other processing cores. To implement it, each cache block is extended by a field to store the identifier of the owner thread, as TID shown in Fig.3. The block agglutinating cache partition scheme does not need support of operating system. The TID field denotes the right of replacement only, so a processing core may be hit in a cache block belong to another processing core. For a many-core processor with n processing cores, a $\log_2 n$ bits' TID field can implement full thread IDs. In our baseline configuration, the overhead is about 1% of shared cache.

2.2 Adaptively Occupancy of Blocks

If blocks are occupied by threads as first-come-first-service allocation policy, it may potentially result in an unfair division of cache space between threads. So it should be an adaptive block agglutinating scheme, which means that a processing core should yield some own blocks adaptively. To implement it, cache block should be extended with a saturating counter, *i.e.* an access frequency counter, as AFC shown in Fig.3.

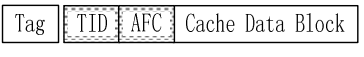

Fig. 3. Structure of Cache Set

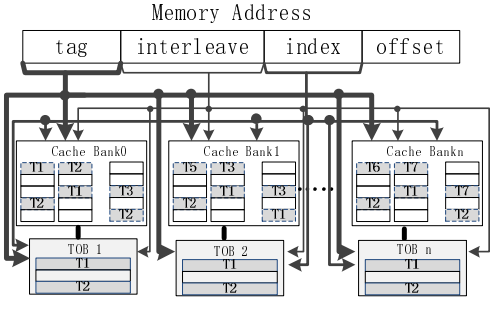

Fig. 4. Block Diagram of TOB Cache

It predefines a threshold to denote the acceptable lowest reuse frequency. When a block is loaded from off-chip memory into shared cache, its AFC is set to be equal to the threshold. When value of an AFC counter is lower than the threshold, it means that the block can be preempted by another processing core as its new block's placement position. Update of AFC is based on the replacement policy, as shown in Fig.5. The value of an AFC counter will be incremented on every hit to this block. When there is a cache miss in the indexed set, all AFC counters excluding its own are decremented. Thus, value of AFC denotes the block's reuse frequency. When its AFC counter is less than the threshold, a block's TID is set to be invalid. Note that a TID is set to be invalid does not mean the cache block is invalidated. In this case, it means that the cache block can be replaced by another processing core, but the already existing cached data is valid and can be accessed until it is evicted from shared cache. In addition, a cache access maybe hit in a block owned by another thread, because the TID means the replace right only.

The implementation complexity of adaptive block agglutinating scheme is negligible. In our baseline configuration, a 4-bit AFC per cache block is adopted, and the

additional hardware cost is about 0.9% of shared cache. And along with TID field, the total additional hardware cost is about 1.9% of shared cache.

2.3 Thread Owned Block Cache

In this section, we present a data block migrating scheme supported by a dedicated hardware buffer for each shared cache bank, called thread owned block cache, *i.e.* TOB Cache. The fully associative TOB caches store data blocks that would incur conflict between threads in cache banks. It is exclusive between cache bank and its corresponding TOB cache. For simplicity, it is assumed that a static NUCA cache [1] in the baseline configuration, but the proposed schemes can be extended to other architectures easily. Each of all shared last-level static NUCA cache banks is implicitly partitioned by the proposed adaptive agglutinating scheme, as shown in Fig.4. The TOB Cache is also implicitly partitioned by block agglutinating scheme.

The hit policy can be illustrated in three cases. In the case of a cache bank hit, it behaves same as traditional shared cache. The request is satisfied immediately. But once a cache bank misses, it should look up the corresponding TOB cache. If it hits in the TOB cache, the request is satisfied by TOB cache without additional latency. If a TOB cache miss happens, an off-chip miss occurs.

2.4 Placement and Replacement Policy

Instead of explicitly partitioning the cache, block agglutinating scheme implicitly partitions the cache and isolates different threads' cached blocks by placement and replacement policies simply. There are two baselines for the cache placement and replacement policy. The first is to ensure cache capacity is fairly divided between threads. The second is to hold cache blocks accessed frequently in cache.

We propose a novel replacement policy, called as LRU-NonPollution. The essential is to replace a block as LRU-like policy on the condition of reducing data pollution between different threads. It ensures the fairness of cache space partition between threads.

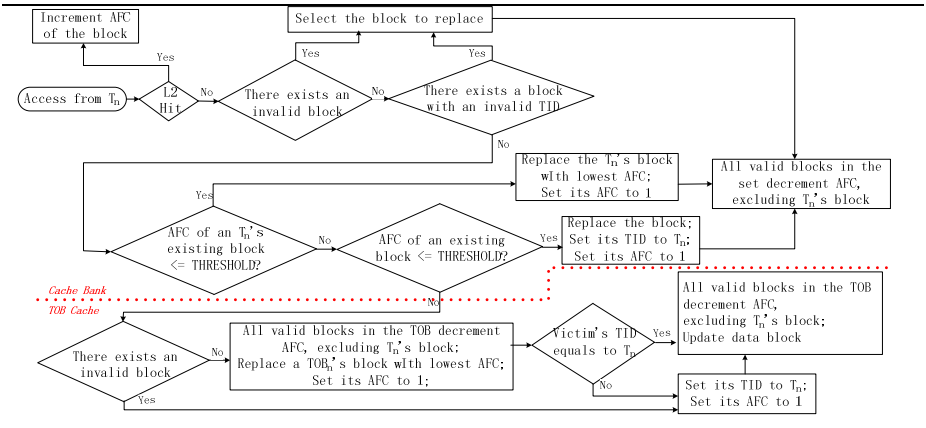

Fig. 5. Flow Chart of Replacement Policy

When a compulsory miss occurs, the required block is loaded into shared last level cache from off-chip memory. It firstly selects an invalid block to cache the newly block. Otherwise, it selects a valid block with invalid TID. Because if so, it means that its owner thread yields its occupancy, so it may be replaced by another thread's newly requesting block. Otherwise, it selects a cache block with lowest AFC value occupied by this processing core in this set. But if all of its occupied blocks have a higher AFC than the predefined threshold, the newly block is placed to TOB Cache. The flow chart of replacement policy is shown as Fig.5.

When replacement occurs in TOB cache, it also selects a victim basing on TIDs and AFCs. It will select the requesting thread's own block with the lowest AFC as a victim. But if the AFC is still larger than a block of another thread, it will replace another thread's block with the lowest AFC. At the same time, its TID is updated to the requesting thread.

When it comes to physical implementation, the selection of victim is implemented by combinational logic circuit, including compare of TID and AFC counter. In addition, update of AFC counter is not in the critical path to refill level-one private cache. So the proposed schemes have negligible effect on shared cache access latency.

3 Methodology

3.1 Simulation Platform

To combine simplicity of dance-hall [3,21] and scalability of tiled architecture [12], we present Godson-T many-core architecture [25], as shown in Fig.6. It is a homogenous many-core processor which is integrated with in-order dual-issue processing-cores, and each processing core is based on the MIPS II architecture. Each core has independent private level-one data and instruction cache, and configurable programmer-controlled Scratch-Pad Memory. It has shared level-two static NUCA cache, synchronization manager and memory controller on chip. It introduces a 2-D mesh as on-chip network. Applications run on Godson-T via the management of a lightweight runtime system, and the runtime system is responsible for thread creation, dispatch, and joins.

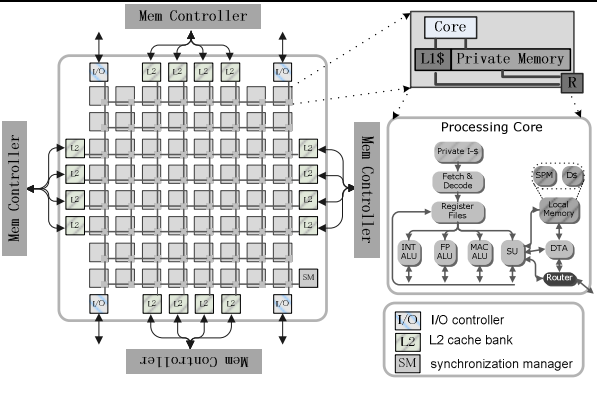

Fig. 6. Godson-T Simulator Architecture

Level-two and level-one cache are non-inclusive, which means that the data cached in L1 and L2 can be replaced respectively and will not incur invalidation to each other. The level-two cache is designed as non-blocking cache. It is composed by interleaved banks. Each level-two cache bank can support four outstanding cache misses. They are located along four sides of the chip die and are shared by all processing cores. All on-chip processing cores access level-two cache through mesh network. The latency of an access from a processing core to different level-two cache banks is different.

The following evaluation bases on the Godson-T cycle accurate C simulator and the simulation configuration parameters are as Table 1 shows.

Table 1. Configuration Parameters

Module	Description
processing core	64 processing cores, 8 stage pipeline, in-order 2-issue, 2 ALU&FPU, 1 DIV/MUL, non- preemptive thread execution model
L1 DCache	private, 32KB, 32B/block, 4-way set associative, 1cycle/hit; write back and write allocate policy outside critical section, write through and write non-allocate policy inside critical section
SPM	configurable from L1 data cache; 1 cycle latency
L1 ICache	private, 16KB, 32B/block, 2-way set associative, 1cycle/hit
L2 Cache	shared, total 2MB, 16 banks (128KB/bank), single read & write port per bank, 64B/block, 8-way set associative per bank; out-standing miss support, 4 outstanding cache misses, hit latency 4 cycles
TOB Cache	fully associative, 8KB per cache bank, 64B/block; AFC width: 4 bits
NoC	8x8 2-D mesh, static X-Y routing, 2cycles/hop
Router	2 stage pipeline, 2 virtual channel, 128-bit data bandwidth
Memory Controller	4 controllers, interleaved address mapping, 512-bit data bandwidth, read latency 52 cycles, write latency 32 cycles

3.2 Benchmarks

We select five kernels of scientific computation, *that is*, matrix multiplication (MM), FFT, LU decomposition, Radix and Cholesky from Splash2 [22], and pFind workload [26] from bioinformatics. PFind is a bioinformatics algorithm which is for automated peptide and protein identification from tandem mass spectra. PFind is a memory access intensive program, and it can evaluate memory systems abundantly.

These benchmarks are executed until completion, and performance data is recorded without pre-warming cache. The program size of workloads is shown as Table 2.

Table 2. Program Size

Workload	Program Size	Optimization	Instructions(10^6)
pFind	32,768 peptides	-O3	547.49
FFT	1,048,576 Complex Doubles		101.8
LU	512x512 Matix 16x16 Element Blocks		618.2
Cholesky	Input: tk 15.O		1,048.29
Radix	1M Keys, 1,024-Radix		318.31
MM	256x256 Doubles Matrix		10.9

4 Experimental Results

In the following graphs, TSharedCache means the baseline platform, *i.e.*, traditional shared cache. The effect of block agglutination scheme without TOB cache has been evaluated in [16]. So in this paper, we evaluate the effect of block agglutinating and TOB cache on shared cache via reduction of miss rate and conflict miss rate, and the hit rate of TOB cache is the effect on reduction of accessing to off-chip memory. The effect on overall performance is reflected by average load latency of shared cache and private cache, average miss penalty of private cache and IPC.

4.1 Cache Miss Rate

Because we do not introduce prefetching, so there is no influence on number of compulsory misses. The actual number of misses is those occur in both cache bank and TOB cache. So the actual average shared cache miss rate is computed as follows:

$$MissRate = \frac{SharedCacheMisses - TOBCacheHits}{TotalL2Accesses}$$

The effect on total shared cache miss rate is shown as Fig.7. The miss rate is normalized to traditional shared cache. It shows that the proposed scheme can improve miss rate of most of benchmarks excluding Cholesky. The reason is that Cholesky has more conflict cache blocks that are mapped to less distinct set indices. And it results that lots of blocks are forwarded to the small TOB cache. As a result, the blocks are always replaced each other frequently. While using random replacement policy, this problem does not exist. On average, the proposed scheme makes miss rate of shared cache reduced by about 18%.

4.2 Conflict Miss of Shared Cache

The proposed block agglutinating scheme is dedicated to avoid conflict between threads in shared cache. The reduction of shared cache conflict misses is shown as Fig.8, in which results are normalized to traditional shared cache. Although it shows in Fig.7 that the proposed scheme has little matter on some workloads, the results show that block agglutinating scheme can reduce shared cache conflict misses for all of these evaluated benchmarks, especially for Matrix Multiplication, in which the scheme avoids almost all of its conflict misses. The reason is that we evaluate a matrix with less numbers, which has a smaller working set, and most of its misses is compulsory misses. On average, the proposed scheme makes conflict miss rate of shared cache reduced by about 40%. The absolute ratio of conflict misses to total shared cache misses is shown above the column in graph.

4.3 Hit Rate of TOB Cache

Because all misses occurred in cache banks will be forwarded to TOB caches, the original miss number of shared cache is the total number accessing to TOB cache. So the average effective hit rate is computed as follows:

$$TOB_{HitRate} = \frac{TotalNumberofTOBCacheHits}{SharedCacheMisses}$$

The compare of average effective hit rate between TOB caches and victim caches is shown as Fig.9. The results are normalized to the hit rate of TOB cache. It shows that the hit rate of TOB cache is higher than victim cache obviously. The cached blocks in victim cache are replaced by demand rate of different threads, so it results in lower hit rate. While by retaining a fraction of work sets frequently used from different threads, the TOB cache can improve hit rate obviously. It also shows that the scalability of TOB cache with number of processing cores increasing. On average, the improvement of hit rate by TOB cache to victim cache is increased by about 52%.

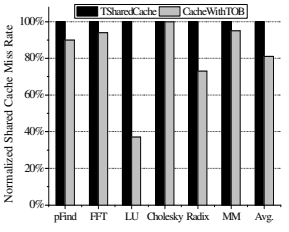

Fig. 7. L2 Cache Miss Rate

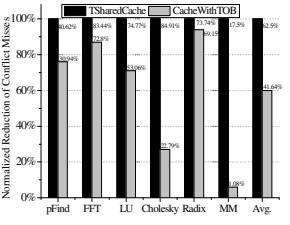

Fig. 8. L2 Conflict Miss Rate

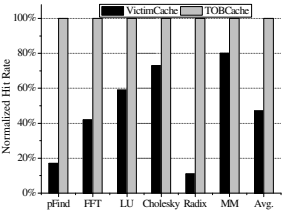

Fig. 9. Avg. Hit Rate of TOB

4.4 Performance Comparison with Victim Cache

The important difference between TOB cache and victim cache [18] is that the victim cache stores the victims after they are evicted, while TOB cache stores blocks that would evict some blocks from cache banks. Similar to TOB cache, every cache bank is equipped with a victim cache, and the victim cache has a comparative implementation cost to TOB cache. The results of shared cache with victim cache and TOB cache are normalized to the corresponding result of traditional shared cache.

4.4.1 Avg. Load Latency of Shared Cache

The proposed schemes' effect on reducing average load latency of shared cache is shown as Fig.10. Here, load latency means that the processing cycles elapse from shared cache accepting a load request to the cycle when level-one cache receives refilling data sent from shared cache. During the elapsed cycles, there may be a shared cache miss to off-chip memory caused by the load request. On average, it makes average load latency of shared cache reduced by about 26%.

4.4.2 Avg. Load Latency of Private Cache

The average load latency including hit latency and miss penalty of primary data cache, and the normalized result is shown as Fig.11. The results show that the proposed schemes reduce average load latency of primary data cache significantly, excluding Matrix Multiplication. The reason is that the evaluated matrix has less numbers, and it incurs less misses in private data cache. The absolute latency of every workload when using TOB cache is shown on the column. On average, the proposed scheme reduces average load latency of private data cache about by 2 cycles, which is about 12%.

4.4.3 Avg. Miss Penalty of Private Cache

The miss penalty of primary cache is processing cycles started from a primary cache miss occurs to completion of refilling from next-level shared cache. And we divide the miss penalty by each thread' total executing cycles to denote the penalty of processing primary cache miss. The result is normalized to traditional shared cache, and is shown as Fig.12. On average, the proposed scheme reduces ratio of primary data cache average miss penalty about by 14%.

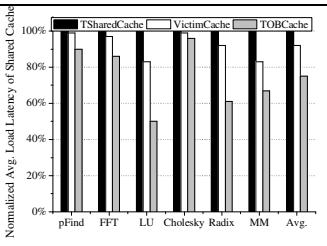

Fig. 10. Avg. Load Latency of L2 Cache

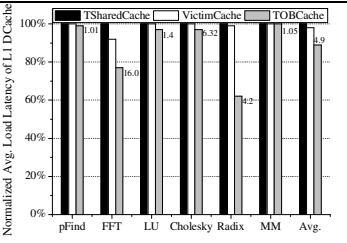

Fig. 11. Avg. Load Latency of L1 DCache

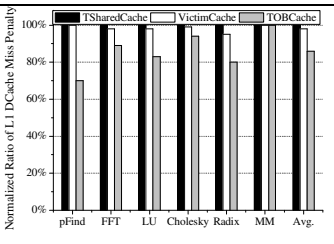

Fig. 12. Ratio of Avg. L1 DCache Miss Penalty

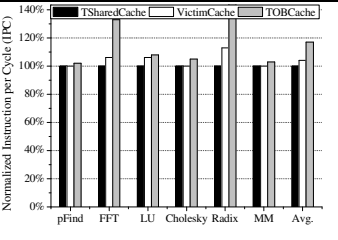

Fig. 13. IPC

4.4.4 Instructions per Cycle

The normalized comparison of IPC is shown as Fig.13. It shows that the proposed schemes can improve most of the evaluated benchmarks, excluding pFind and Matrix Multiplication. Because the evaluated matrix has less numbers, and it incurs less conflicts between threads in shared cache. But for pFind, the main reason may be the stream-like memory access behavior of its algorithm. In this case, the locality is limited and the proposed schemes cannot improve its performance obviously. But on average, the proposed schemes make IPC improved by about 17%.

In summary, the reduction of average load latency of shared cache and private cache means that TOB cache does not length critical path of shared cache. The performance improvement can be explained intuitively as follows. The victim cache is designed to cache all victims replaced from cache bank, while the TOB cache stores only those blocks which would cause conflict. The victim cache may be occupied by a thread while other threads cannot assign enough space to cache their frequently used

victims. However, TOB cache avoids it by replacement right. In the future, besides scientific algorithms and bioinformatics, some real time applications, such as image processing, will be evaluated. In addition, we do not evaluate sensitivity, such as capacity of TOB cache or grain of agglutination since the consuming execution time, and also leave them as future work.

4.5 Implementation Cost

In this section we evaluate area and power's cost of TOB cache via Cacti simulator [27]. The input of Cacti simulator is configuration parameters of shared cache in Godson-T baseline platform and TOB cache, as shown in Table 3. The additional power cost increased by TOB cache is 0.39 nJ, and it occupies about 23% of a shared cache bank. The ratio of TOB cache's area to a shared cache bank's area is about 15%. The area ratio of victim cache to a shared cache bank is about 11%, which both is implemented by TSMC 130nm ASIC technology. Although area of TOB cache is larger than victim cache, the hit rate of TOB is far higher than victim cache, as shown in Fig.9, which is improved about 52%.

Table 3. Architecture Configuration Parameters Input to Cacti Simulator

Module	Capacity (B)	Blocks (B)	Associativity	Tech.	Num. of Banks
L2 Bank	131,072	64	8	0.13μm	1
TOB Cache	8,192	64	FA	0.13μm	/

5 Related Work

We describe the most related previous works concisely and the main difference between these studies and our contributions. There are many static and dynamic shared cache partition schemes that have been presented to provide isolation, such as [4,7,9,15,20,23,24]. Srikantaiah et al. [24] presented an adaptive set pinning scheme which is similar to our block agglutinating. The subtle, but important difference is that the set pinning partitions cache space as grain of set, while our scheme achieves this purpose as grain of cache block. With set pinning, the wasted cache capacity would be larger, and it decreases effective on-chip cache capacity. Furthermore, our proposed adaptive block agglutinating scheme does not incur strictly hard partitions in the shared cache, and a processing core's activity is similar to steal cache capacity from another processing core. So it makes better use of the total available shared cache resources. Zhang, Chang and Qureshi et al presented data migration in [8,13,14,17] respectively. The difference is that we put blocks that would incur conflict into TOB cache rather than to another cache bank, to avoid eviction of an existing block and reduce off-chip access. Many important works address conflicts for uniprocessors in both hardware and software [2,6,18,19]. Jouppi [18] presented victim cache to avoid negative effect of conflict. The main differences between TOB and victim cache are blocks of TOB cache is pinned to a thread, and replacement policy is different. Srikantaiah et al. [24] presented an adaptive set pinning scheme by adding a small POP cache for each processor. The main difference between TOB and POP

cache is that TOB cache is agglutinated to cache bank, while POP cache is pinned to processor, so when it comes to many-core architecture, its implementation complexity and area cost is unacceptable.

6 Conclusions and Future Work

On-chip memory hierarchy is critical to performance of many-core architecture. Shared last-level cache is necessary to increase effective on-chip capacity. However, the conflict in shared cache between threads has been a long standing problem. When it comes to context of many-core architecture, it makes memory access latency increased obviously. In this paper, we present an implicit and dynamic shared cache partition scheme, *i.e.* block agglutinating. The scheme can be implemented with negligible hardware cost and without support of operating systems. Based on the partition scheme, we propose thread owned block cache, *i.e.* TOB cache, a hardware buffer to reduce longer cache access latency via migrating conflicting data. Experimental results show the effectiveness of schemes when compared to the traditional shared cache and victim cache.

For different benchmarks, estimation of the best size of TOB cache would give a good insight on how often conflicts appear and the illustration of reuse distance, and it is an important future work. In the future, we will also extend the schemes to eliminate intra-thread's conflicts and support fairness of shared cache partition scheme.

Acknowledgments. This paper is supported by the National Natural Science Foundation of China under Grant No.60736012, the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321600, the National High-Tech Research and Development Plan of China under Grant No.2009AA01Z103, the National Science Foundation for Distinguished Young Scholars of China under Grant No.60925009, and the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60921002.

References

1. Kim, C., Burger, D., et al.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: ASPLOS 2002 (2002)
2. Zhang, C.: Balanced cache: Reducing conflict misses of direct-mapped caches. In: ISCA 2006 (2006)
3. Almasi, G., et al.: Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. ACM SIGARCH Computer Architecture News 31(1), 26–38 (2003)
4. Suh, G.E., Rudolph, L., Devadas, S.: Dynamic Partitioning of Shared Cache Memory. The Journal of Supercomputing 28(1), 7–26 (2004)
5. Pfister, G.F., Norton, V.A.: 'Hot-spot' contention and combining in multistage interconnection networks. IEEE Trans. Comput. C-34, 943–948 (1985)
6. Collins, J.D., Tullsen, D.M.: Runtime identification of cache conflict misses: The adaptive miss buffer. ACM Trans. Comput. Syst. 19(4), 413–439 (2001)
7. Huh, J., Kim, C., Shafi, H., et al.: A NUCA substrate for flexible CMP cache sharing. In: ICS 2005, June 20-22 (2005)

8. Chang, J., Sohi, G.S.: Cooperative Caching for Chip Multiprocessors. In: ISCA 2006 (2006)
9. Chang, J., Sohi, G.S.: Cooperative Cache Partitioning for Chip Multiprocessors. In: ICS 2007 (2007)
10. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report No.UCB/EECS-2006-183, December 18 (2006)
11. Olukotun, K., et al.: The Case for a Single-Chip Multiprocessor. In: ASPLOLS VII (October 1996)
12. Taylor, M.B., et al.: Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and Streams. In: ISCA-31 (June 2004)
13. Kandemir, M., Li, F., et al.: A Novel Migratrion-Based NUCA Design for Chip Multiprocessors. In: SC 2008, Austin, Texas (November 2008)
14. Qureshi, M.K.: Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In: HPCA 2009 (2009)
15. Qureshi, M.K., Patt, Y.N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In: MICRO 2006 (2006)
16. Song, F., Liu, Z., et al.: An Implicitly Dynamic Shared Cache Isolation in Many-Core Architecture. Chinese Journal of Computer 32(10), 1896–1904 (2009)
17. Zhang, M., Asanovic, K.: Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches. MIT-CSAIL-TR-2005-064, MIT-LCS-TR-1006, October 10 (2005)
18. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: ISCA 1990 (1990)
19. Topham, N., et al.: The design and performance of a conflict-avoiding cache. In: Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 71–80 (1997)
20. Hardavellas, N., et al.: R-NUCA: Data Placement in Distributed Shared Caches. In: ISCA 2009 (2009)
21. Kongetira, P., Aingaran, K., et al.: Niagara: a 32-Way Multithreaded Sparc Processor. In: HotChips'16 (2005)
22. Woo, S.C., Ohara, M., et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: ISCA 1995, pp. 24–36 (June 1995)
23. Kim, S., Chandra, D., Solihin, Y.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In: PACT 2004 (2004)
24. Srikantaiah, S., Kandemir, M., Irwin, M.J.: Adaptive set pinning: Managing shared caches in chip multiprocessors. In: ASPLOS 2008 (2008)
25. Fan, D., et al.: Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions. Journal of Computer Science and Technology 24(6), 1061–1073 (2009)
26. Fu, Y., et al.: Exploiting the kernel trick to correlate fragment ions for peptide identification via tandem mass spectrometry. Bioinformatics 20, 1948–1954 (2004)
27. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0 (CACTI 6.0: A Tool to Model Large Caches.). In: Micro (2007)