SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2015, Vol. 58 012104:1–012104:15
doi: 10.1007/s11432-014-5175-8

Improving multiprocessor performance with fine-grain coherence bypass

WANG Hui[1], WANG Rui[1]*, LUAN ZhongZhi[1], QIAN XueHai[2] & QIAN DePei[1]

[1]*Sino-German Joint Software Institute, School of Computer Science and Engineering, Beihang University, Beijing* 100191*, China;*
[2]*University of Illinois Urbana-Champaign, Urbana* 61801*, USA*

Received May 15, 2014; accepted July 7, 2014; published online September 10, 2014

Abstract Efficient and scalable cache coherence protocol is crucial to high-performance servers with shared-memory. The directory-based cache coherence protocol is more desirable than the snooping-based protocol with respect to the scalability. However, even for the former protocol, scaling to a large number of cores is also challenging due to the additional area requirements of the directories. We observed that a significant percentage of the referenced memory blocks were only accessed by a single core (even in parallel applications) which could be considered as private memory blocks. An intuitive motivation from this observation is that memory blocks accessed by a single core do not require coherence maintenance. The issue is to identify the private block and track the change of its access pattern. We propose a novel hardware approach to (1) dynamically identify the shared memory blocks at the cache block level, and (2) bypass the coherence procedure for the private memory blocks. This approach increases the effectiveness of the directory-based approach and therefore improves the system performance. Experimental results showed that, our approach can on an average (1) avoid the coherence tracking of about 54% referenced memory blocks, (2) reduce the coherence overhead by 77%, (3) avoid 8% L2 cache misses, and (4) shorten the execution time of parallel applications by 13%.

Keywords many-core, cache coherence, private memory block, fine-grain coherence, high performance

Citation Wang H, Wang R, Luan Z Z, et al. Improving multiprocessor performance with fine-grain coherence bypass. Sci China Inf Sci, 2015, 58: 012104(15), doi: 10.1007/s11432-014-5175-8

1 Introduction

Currently existing technologies have failed to bring performance improvements for single-core processor to follow Moore's Law due to energy consumption and wire-delay issues. Therefore, mainstream microprocessor vendors have turned to thread-level parallelism (TLP) by designing chips with multiple cores, namely multi-core processors or chip-multiprocessors (CMP), i.e., Godson-3 [1] and Godson-3B [2]. In the multi-core architectures, each core has one- or multi-level private caches. The coherence of the private caches is maintained by cache coherence protocols. A scalable cache coherence protocol is crucial for multi-core processors to integrate many cores on a single chip. Directory-based cache coherence protocol is adopted by most of current multi-core architectures because of its better scalability as compared to

*Corresponding author (email: rui.wang@jsi.buaa.edu.cn)

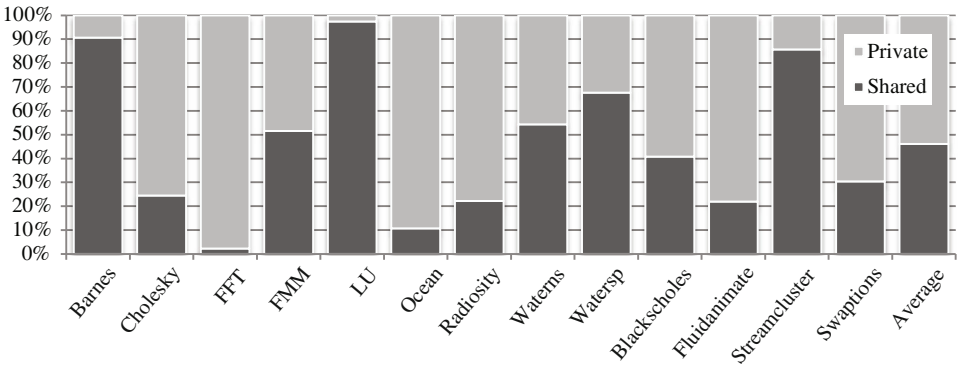

Figure 1 Fraction of private versus shared blocks.

other coherence protocols. However, the circuit area overhead and energy consumption of the directories increase with the number of cores and the size of private caches. Therefore the mere enlargement of the size of the directory would not be a reasonable solution. More efficient coherence protocols are needed for enabling more powerful multi-core processors.

The conventional directory-based approach keeps tracking all memory blocks in the system, which incurs significant storage overhead. To alleviate the overhead, the proposed systems in several recent studies and some commodity systems only track cached memory blocks [3]. Consequently, the directory entries can be kept in the relatively small set-associative directory cache [4]. However, conflicts might arise in the directory cache when the cache is full. Once a conflict occurs, the directory cache must evict an existing directory entry to make room for the new one and send messages to all the victim's sharers (i.e. up-level private caches) to invalidate their copies, which increases the private cache miss rate and affects the system performance. As the directory cannot scale with the increasingly larger system, it suffers high miss rate (up to 74% as reported in [3,5]), leading to poor directory effectiveness and system performance.

On the other hand, it has been shown that a significant fraction of the memory blocks allocated to applications (even parallel applications) are accessed only by a single processor [6]. It is unnecessary for the directory cache to track those memory blocks for coherence maintenance. We carried out experiments to identify the memory access pattern. Our experiment results shown in Figure 1 indicate that on an average about 54% of the accessed blocks are private. Although private blocks do not require coherence maintenance, the conventional directory cache coherence protocols still keeps track all of them. As a result, a significant fraction of directory cache entries are wasted for tracking the private blocks, which considerably reduces the effectiveness of the directory-based cache coherence protocols. A corrective approach would be to stop tracking the private blocks, such that more directory cache entries can be used for the shared blocks that really need coherence maintenance. In this way, the directory cache capacity could be exploited more efficiently.

In this paper, we propose a hardware approach to achieve the aforesaid concept. Our hardware approach can 1) dynamically identify the shared memory blocks, and 2) bypass coherence procedures for the private memory blocks. When a memory block is loaded for the first time into the Shared Last-Level Cache (SLLC), it is assumed to be a private memory block and does not require an entry in the directory cache to maintain its coherence information. In our approach we add a few additional flags to each cache block in SLLC to track the change of access pattern of each block so that we can dynamically identify whether a cache block is shared or not. When a block becomes shared because of accesses by more than one core, a coherence recovery mechanism is triggered to allocate an entry in the directory cache to maintain the coherence information of that block. The significance of this approach is that the coherence information of the private blocks is no longer maintained in the directory cache and the directory cache can be utilized more efficiently, which improves the scalability of the coherence protocol notably. Our proposal works at the fine granularity of cache block and can be implemented with hardware by small modification to the cache structures, which makes it transparent to the operating system and up-level

applications.

We verify our approach in a simulated multi-core system with three level caches. Evaluation results show that on an average up to 54% of the tracking in the accessed memory blocks can be removed from the directory caches. In consequence, the coherence traffic and the miss rate of the private L2 cache decreases by 77% and 8% on average, respectively. Incidentally, our approach enables 13% improvement to the overall system performance.

The rest of the paper is organized as follows. Related work is surveyed in next section. Section 3 discusses the design and implementation of our fine-grain approach to bypass coherence procedures for the private cache blocks. The method for evaluating our approach and the evaluation results are presented in Section 4. The cost of our approach is also discussed in that section. Finally Section 5 concludes the paper by summarizing the characteristics of our approach and proposing some future works.

2 Related work

Several approaches have been proposed to reduce coherence storage overhead by distinguishing shared and private data. For example, POPS [7] optimizes coherence protocol by placing private and shared data on different L2 cache slices in NUCA architecture. It uses predictor and the directory information to identify private data, which also waste the directory entries. SPTAL [8] enables Tagless directory [9], which uses bloom filters to summarize the tags in a cache set; it uses a full map sharing vector to represent the cache block's sharing information. However, it is observed that many bloom filters replicate the same sharing pattern due to the regular nature of applications. So they exploit this observation to decouple the sharing pattern then decompress the coherence directory. Zhang et al. [10] claimed that both data access patterns exhibited by different threads of a multithreaded application and the on-chip cache topology of the target CMP architecture by modifying the compiler to identify the private data and implement automatic data layout transformation. Note our approach is orthogonal to theirs, which means they can be used simultaneously to further reduce the directory's storage overhead.

Unlike our fine-grain approach, Cuesta et al. [11] exploited the operating system to identify private and shared pages and degrade coherence for private memory block. They modified the TLB entries in hardware and the page entries in the operating system. When encountering a TLB miss, the operating system checks the page table entry to find whether the page is shared or private. Although the motivation is similar to ours, the granularity of their software approach is limited to virtual page (coarse-grain). In contrast, we use hardware to detect private and shared blocks at the cache block level, the granularity is much finer than that of virtual pages. The coarse-grain mechanism treats all the data in a page as an atomic bulk, which leads to a small portion of private blocks to be identified. The drawback however is that, once a single data in a private page is identified as shared the whole page needs to recover coherence, resulting in flooding on the network on chip.

Some previous works performed coarse-grain tracking to reduce unnecessary traffic of broadcast-based protocols. Cantin et al. [12] proposed Region Coherence Arrays to identify shared regions and filter unnecessary broadcast traffic. Moshovos et al. [13] proposed RegionScout to avoid sending snoop requests. Region-Tracker [14] provides a framework to reduce the storage overhead. All these techniques share the principle of deactivating the coherence mechanism when it is not necessary. However, those approaches aim at reducing broadcast traffic at coarse-grain granularity, while our proposal works at fine-grain granularity to avoid allocating directory cache entry for private blocks and does not require coherence maintenance.

Other works have employed combination of software and hardware to support cache coherence. Zeffer et al. [15] proposed a trap-based architecture (TMA). TMA uses hardware to detect fine-grained coherence violations. When a violation occurs, it triggers a coherence trap, and maintains coherence by software (the coherence trap handlers). Similar to the approach in [11], TMA adds one bit to each TLB entry and relies on the operating system to detect shared page. The OS-aided approach with TLB modification requires extra hardware support in each core, which makes implementation difficult. Alternatively, Zeffer et al. also proposed a simple hardware mechanism which facilitates the software implementation of the

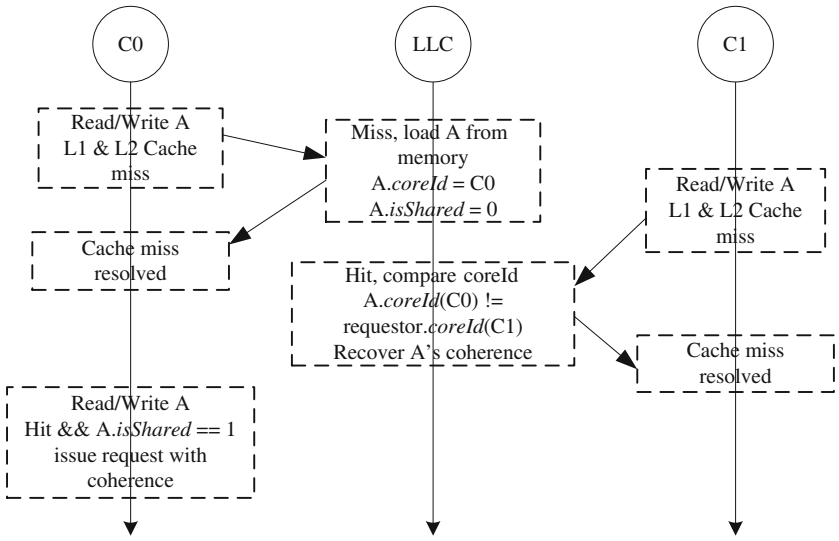

Figure 2 Overview of the fine-grain coherence bypass mechanism. C0 and C1 are cores, LLC is the shared last level cache.

inter-node coherence protocol [16]. However, the software overhead is high in comparison to our exclusive hardware approach.

3 A fine-grain coherence bypass approach

In this section we have described the details of our hardware fine-grain coherence bypass approach and its implementation.

3.1 Overview

Traditional directory cache keeps track of all cached memory blocks. However, as reported in [6], a significant fraction of the cached memory blocks are private, i.e., they are only accessed by one core during their lifetime and no coherence issue arises. Therefore, keeping track of the private blocks in the directory cache entries is futile, which will reduce the effectiveness of directory cache utilization and affect the system performance. We propose a fine-grain hardware approach to address this issue. The strategy is to allow the private blocks to bypass coherence protocol so that the private blocks can be accessed more quickly and more space of the directory cache can be used to maintain coherence of shared blocks. To achieve this goal, our approach needs to dynamically identify the shared memory blocks at the cache block granularity and bypass coherence procedure for the private memory blocks. By doing this, our approach can effectively utilize the space of the directory cache and improve the overall system performance.

In our approach, upon a cache miss in the Shared Last-Level Cache (SLLC), the requested block is retrieved from the main memory. The block retrieved from the main memory is assumed to be private at beginning by default. Thus the directory cache will not allocate an entry to keep track of the private block, and the coherence protocol for that block is bypassed. In order to distinguish private and shared blocks we added additional tags to each cache block in SLLC. By testing those tags we could identify dynamically whether or not a private cache block transforms into a shared one. Once we found a private block being accessed by more than one cores, the block becomes shared and the coherence recovery mechanism is triggered. SLLC will send a coherence recover message to the directory cache to maintain coherence of the shared block.

Figure 2 outlines our mechanism. First, core C0 references the memory block A, since it misses on all C0's private caches (L1 and L2), L2 will issue a non-coherent request to SLLC. The non-coherent request

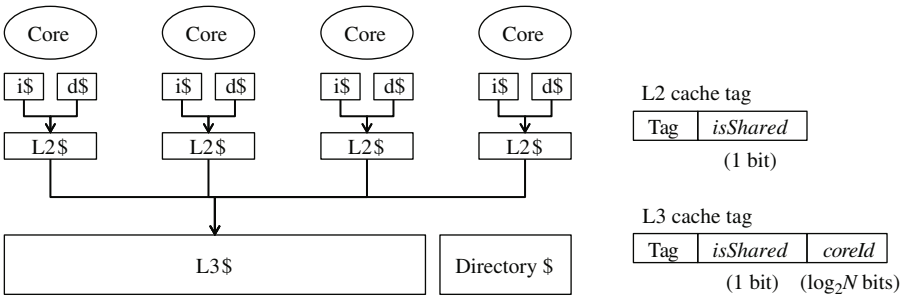

Figure 3 Baseline architecture and cache hardware modification.

also encountered a cache miss on SLLC, which meant that block A has not been loaded from memory, so we treated block A as a private block. After the requested block A is retrieved from the main memory, the directory cache will not allocate an entry to store block A's coherent information. Block A is further passed to C0s L1 and L2. C0 can keep accessing block A as its private data from its L1 cache. Later on, core C1 issues a access to the same memory block A, which is missed on all C1's private caches (L1 and L2). However, when the non-coherent request issued by C1 reaches SLLC, it encounters a cache hit. By comparing the initial loader (C0) and the requestor (C1) of block A, SLLC finds that the initial loader and the current requestor are different, which means that block A becomes shared (Loaded by C0 while accessed by C1). Consequently, SLLC triggers the coherence recovery mechanism to record block A's coherent information in the directory cache. After the recovery process, if C0 references block A again, it must perform the access according to A's coherence state by issuing a coherent request. Take MESI protocol as an example, after the coherence recovery process, block A in both C0's and C1's private cache has the coherence state 'S' (shared), if C0 wants to write into block A, it needs to issue a coherence request to get the write permission before changing A's coherence state to 'M' (modified).

We have explained our approach in detail by walking through each key aspect such as the baseline architecture and hardware modification (Subsection 3.2), the coherent and non-coherent requests (Subsection 3.3), the detection of shared cache blocks (Subsection 3.4), and the coherence recovery mechanism (Subsection 3.5). Finally, we have discussed some implement issues (Subsection 3.6).

3.2 Baseline architecture and hardware modification

We took a CMP system with three-level cache hierarchy as our baseline system (shown in Figure 3). In this system, each core has its own instruction cache (i$), data cache (d$) and private L2 cache (L2$). The L3 cache (L3$) is shared by all cores of the system as an inclusive cache. The baseline system implements a directory-based MESI cache coherence protocol by an additional directory cache (Directory $). To simplify the discussion, we assume the instruction and data cache are write-through caches.

From the principle of our approach, we learn that we need to detect dynamically if a private block resident in L3 becomes a shared one by comparing the identities of the requesting core and the initial loading core. In order to do that, we need to record extra identity information for each cached blocks so that this detection can be performed on the fly. Considering the baseline architecture in our study, we made small modification to L2 and L3 caches structures. As shown in right part of Figure 3, we added a flag, *isShared* (one bit in size), to the flag part of each L2/L3 cache block to indicate whether the block is shared or private. We added another flag field, *coreId*, to each L3 cache block to record the identity of the core which loads the cache block initially into L3. The size of the *coreId* flag is $\log_2 N$ bits, where N is the number of cores sharing L3. We also added a comparison logic to L3 to compare the identifier of the core accessing the block and *coreId* in L3 access, which is detailed in Subsection 3.6.

3.3 Coherent and non-coherent requests

In our approach, we distinguished coherent and non-coherent requests. We assumed every cache block loading into the L3 cache the first time is private and is marked so in L2 and L3. The core which

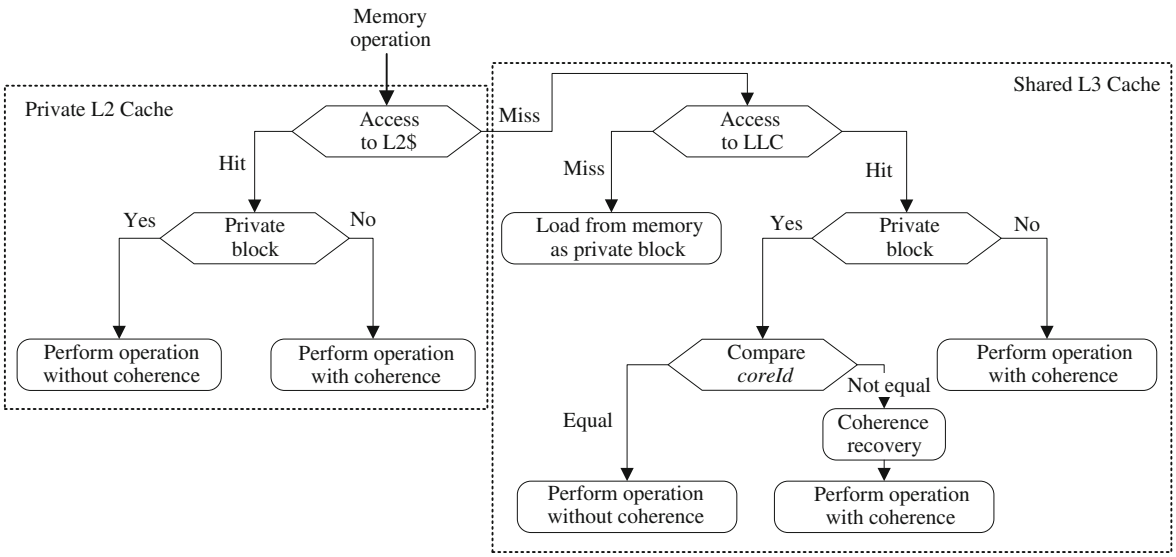

Figure 4 Block diagram of memory access of the proposed scheme.

creates a private block can access the private cache block without passing through the coherence protocol procedure. Since the block is private, it is not necessary to store the coherence information for that block in the directory. When private L2 cache incurs a memory reference (we will not discuss L1 cache since we assume it is write-through), as shown in the left part of Figure 4, it issues a lookup process to search the block. If no block matches, it means that a cache miss occurred, the memory block referenced is going to be loaded for the first time and assumed to be a private block, and L2 cache will issue a non-coherent request to L3 cache. On the other hand, if block match occurs, the *isShared* flag of the matching cache block will be checked.

If the *isShared* bit is not set (0 in value), L2 cache performs the access as the cache block is private to the core, that is, by bypassing the coherence maintenance. However, if the *isShared* bit is set (1 in value), L2 cache needs to perform the access according to the coherence state of that cache block and issues a coherent request.

In our MESI baseline architecture shown in Figure 3, if a private L2 cache block in MESI state 'S' (i.e. the block is shared by other cores and copies present in other cores private cache) is with the *isShared* bit set, on a write request, the L2 cache will send a coherent request to the directory cache to get the write permission. Otherwise, if the *isShared* bit is not set, indicating that the cache block is private, on a write request, the L2 cache will perform the write regardless of what the MESI state is.

3.4 Detection of shared cache blocks

Unlike the OS-aided approach in prior works, we used the *coreId* flag in shared L3 cache to distinguish between private cache blocks and shared cache blocks. When a L2 non-coherent request arrives at the L3 cache, if not hit, L3 cache will load the block from memory as a private block, unset the *isShared* bit of the corresponding cache block, and fill the *coreId* flag bits with the request core Id (shown in Algorithm 1).

If L3 cache hits and the *isShared* flag bit of the block is not set, a comparison between the *coreID* of the cache block and the ID of the requestor core will be performed. If they are equal, it means that the cache block is still referenced by only one core and remains private. Otherwise, the cache block is being assessed by more than one cores and becomes shared. On transition from private to shared in our approach, a coherence recovery action gets triggered that creates an entry for that block in the directory cache, restores the coherence state of that private block, and converts its state into shared. From that point on, the block is considered to be shared, and all the following accesses to that block will be performed according to the coherence protocol.

Algorithm 1 Use $coreId$ to detect shared cache blocks

```
1: if cache miss then
2:     Load the block from main memory
3:     isShared = unset
4:     coreId = requestor'sId
5:     #Will not allocate an entry in directory cache
6: else
7:     if isShared == unset then
8:         if coreId! = requestor'sId then
9:             Trigger coherence recovery mechanism
10:            isShared = set
11:        end if
12:    end if
13: end if
```

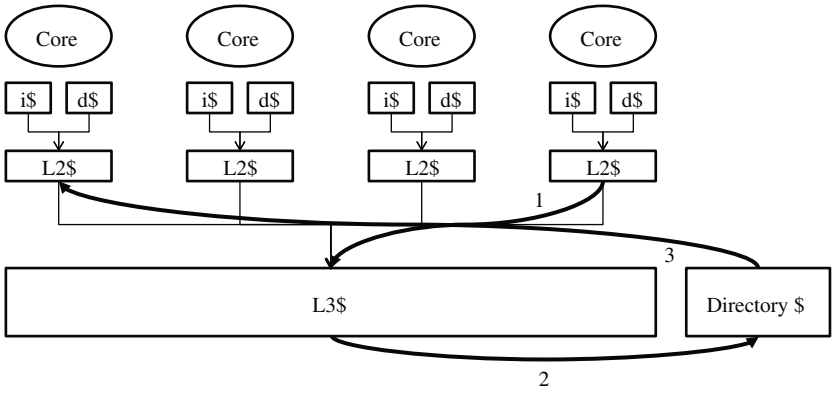

Figure 5 Coherence recovery mechanism.

As an example, in our MESI baseline architecture shown in Figure 3, after core C1 accessing block A, block A is resident in L3 cache with its $coreId$ = C1 and $isShared = 0$. When another core, say C3, issues an access request to block A. The request hits in L3 cache. The L3 cache checks the $isShared$ bit of block A and finds that $isShared = 0$ which indicates the block is private. Then it compares the $coreId$ bits of block A (C1 in value) with the requestor core Id (C3 in value). Since those two core identifiers are unequal, the L3 cache detects that block A is accessed by two different cores, C1 and C3, and should become shared. So it triggers a coherence recovery mechanism to record block A in the directory cache, creating an entry and updating the coherence state.

The right part of Figure 4 illustrates the actions performed on a memory access to shared L3 cache.

3.5 Coherence recovery mechanism

The entry with coherence information in the directory cache is the main difference between private and shared cache blocks in our scheme. A private cache block is not tracked in the directory cache, that is, no entry is allocated for recording the coherence state. However, a shared one is tracked. Therefore, when a private cache block becomes shared, the coherence recovery needs to be performed to maintain coherence of the shared block.

A coherence recovery process is illustrated Figure 5. When a request from a different core arrives to a private cache block and transfers the state of the block from private to shared ('1'), the shared L3 cache sends a request containing the IDs of the two cores: the one initially accessing the private block and the one issuing the new request, to the directory cache ('2'). The directory cache receives this request and creates an entry for that block for tracking its coherence states and the cores sharing the blocks. After creating the entry, the directory cache sends a coherence recovery request to the corresponding private L2 cache ('3'). Upon receiving the coherence recovery request, the private L2 cache controller sets the $isShared$ flag of the accessed cache block and recovers its coherence state.

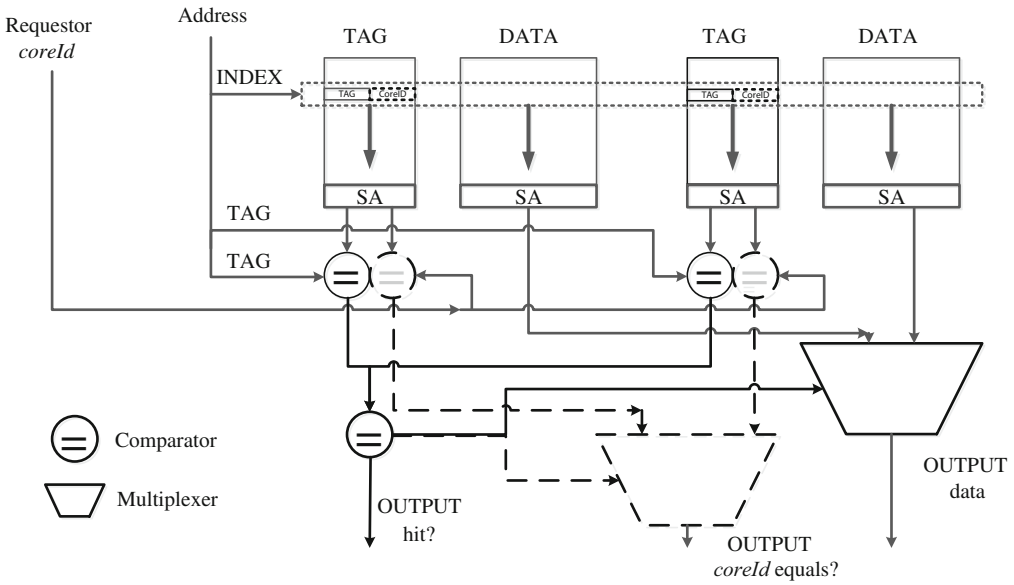

Figure 6 *coreId* compare logic in L3 cache controller.

3.6 Implementation issues

As described before we modified the original cache flag structure to implement our approach. We added a flag *isShared* (1 bit) to each block of L2 cache and L3 cache to indicate the sharing status of the block, and a flag *coreId* ($\log_2 N$ bits) to each block in L3 cache to identify the core which initial loads the block from the memory into L3 cache. We also added a comparator to the L3 cache controller to support *coreId* comparison. Figure 6 shows part of the modified L3 cache controller. It is an implementation for a 2-way associative cache with tag and data arrays. The part with solid line shows the original cache lookup implementation. When an access request arrives, the address in the request is decoded into three parts: INDEX, TAG, and OFFSET. The INDEX is used to find the set. Once the corresponding set is located, all the tags and data inside the set are activated and amplified through the Sense Amplifiers (SA). The TAG is compared simultaneously with the flags of the two blocks in that set by two comparators. Then the output of the TAG comparison results determines whether the cache access is a hit or not. If the access is a hit, one of the blocks (the hit block) of the set is gated to the output of the multiplexer. OFFSET is used to select the corresponding data within the cache block, which is not shown in the figure.

The part with dotted line in Figure 6 shows the newly supplemented hardware for implementation of *coreId* comparison operation. We use two separate comparators to perform *coreId* comparison for the two blocks in the set simultaneously. A multiplexer is used to select the data from one of blocks in the set. Note that *coreId* comparison is carried out in parallel with the original TAG comparison, thus no additional latency introduced. From the above discussion, we know that our hardware coherence bypass technique does not incur any extra latency to cache access and can be implemented with ease.

Our implementation scheme does introduce extra hardware cost, which comes from both storage space for extra *isShared* and *coreId* flags and the comparison logic added to the L3 cache controller. The major hardware cost will be discussed in Subsection 4.6.

4 Evaluation

We have evaluated our proposed scheme and its implementation by simulation. In this section, we present the methodology of the evaluation and analyze the simulation results.

Table 1 System parameters

Component	Parameters
CMP	32 core
Core frequency	2.0 GHz
Core block size	64 bytes
Instruction cache	32 KB, 2-way, 2 cycles, 32 MSHRs
Data cache	32 KB, 4-way, 2 cycles, 64 MSHRs
Private L2 cache	512 KB, 8-way, 9 cycles, 64 MSHRs
Shared L3 cache	8 MB, 16-way, 15 cycles, 128 MSHRs
Directory cache	256 KB, 4-way
Coherence protocol	MESI
Memory access latency	80 ns

Table 2 Benchmarks and input sizes

	Benchmarks	Input size
SPLASH-2 (9)	Barnes	16 k particles
	Cholesky	Input file tk23.O
	FFT	256 k points
	FMM	16 k particles
	LU	512×512 matrix, 16×16 blocks
	Ocean	258×258 ocean
	Radiosity	-batch -room
	Water-nsquared	512 molecules
	Water-spatial	512 molecules
PARSEC (4)	Blackscholes	in_16K.txt
	Fluidanimate	in_100K.fluid
	Streamcluster	simmedium
	Swaptions	32 swaptions, 10 000 simulations

4.1 Methodology

The cycle-accurate simulator SESC[1)], which is able to model a wide set of architectures, is adopted in simulation of our proposed approach. The target system on which we model and simulate the proposed scheme is a directory-based coherent multi-core processor. The main parameters of the target system are shown in Table 1. Our proposed approach is implemented and evaluated upon this system. A variety of parallel workloads are selected from two benchmark suites, (SPLASH-2 [17] and PARSEC [18]), as the workload to drive the simulation models. The benchmarks and the size of their parameters are listed in Table 2.

4.2 Fine-grain detection of the private blocks

Our approach is based on the fact that a significant amount of the memory blocks accessed during parallel program execution are private (see Figure 1). The more private blocks detected, the more coherence overhead can be avoided and the less space in the directory cache would be required to maintain the coherence for the accessed blocks. Some prior works have detected private blocks at a page granularity (coarse-grain). With the coarse-grain approach, a page containing both private and shared blocks or containing only private blocks read by multiple cores will be considered as shared; while all blocks within that kind of page will be treated as shared blocks. Consequently, the coarse-grain approach can only detect a small portion of private blocks. In the extreme case when a page contains only one shared

1) SESC: http://sesc.sourceforge.net.

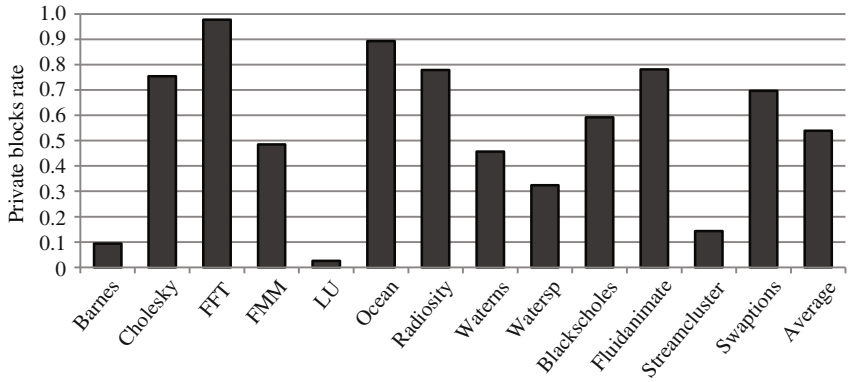

Figure 7 Fraction of detected private blocks.

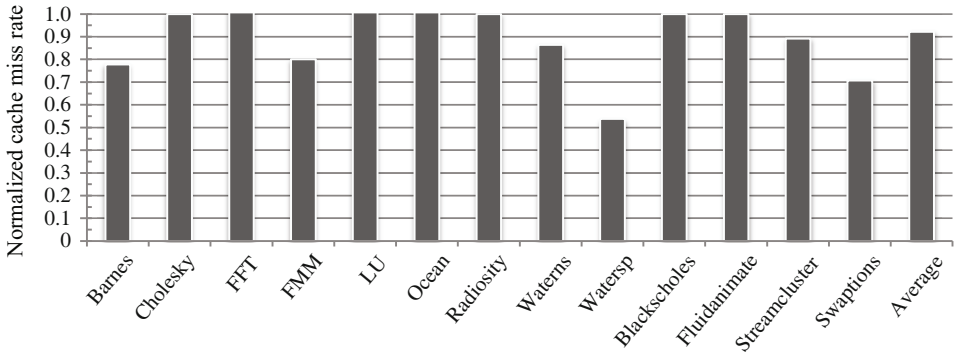

Figure 8 Normalized L2 cache miss rate (the miss rate of the baseline system is 1).

block, the entire page will be treated as shared page. But things may get worse, if one page is detected as shared, all the blocks in this page need to be checked for coherence. These coherence actions will cause a traffic bursty both in the network on chip (NoC) and directory cache. Our proposal adopts a different approach. It uses special hardware to support detection of the private blocks at block granularity. Theoretically, all private memory blocks can be identified. Figure 7 shows the fraction of the memory blocks detected by our approach to be private during benchmarks program execution. Comparing with the data in Figure 1, we can find that our approach can identify all private blocks, that is, about 54% on average, which is impossible for other coarse-grain approaches. Hence, our fine-grain approach is more efficient in identifying private blocks.

4.3 L2 cache misses

As discussed before, in the cache coherence scheme that uses the directory cache, if a conflict (i.e., multiple cache blocks map to the same entry location in the directory cache) occurs in the directory cache, it will invalidate the existing directory entry and send coherence messages to the sharers of the block (the local caches which contain copy of the memory block) to invalidate the shared copies. In other words, the cached block in the core caches and private L2 caches is evicted because of entry conflict in the directory cache instead of conflict in the data cache. This certainly increases the cache miss rate.

With our approach, the directory cache does not need to track private blocks, which leaves more space for entries of the shared blocks and alleviates the contention for directory cache entries. Consequently, less entry evictions will occur because of conflict in the directory cache and fewer data blocks will be invalidated from the core local data cache and L2 cache. In this paper we only discuss L2 cache since we assume core caches are write-through. As a result, the L2 cache miss rate is reduced. Figure 8 shows the ratio of the overall L2 cache miss of our approach to the overall L2 cache miss of the baseline system. We can see that by using our hardware-supported coherence bypassing approach which does not track private blocks in the directory cache, about 8% of L2 cache misses can be avoided on average.

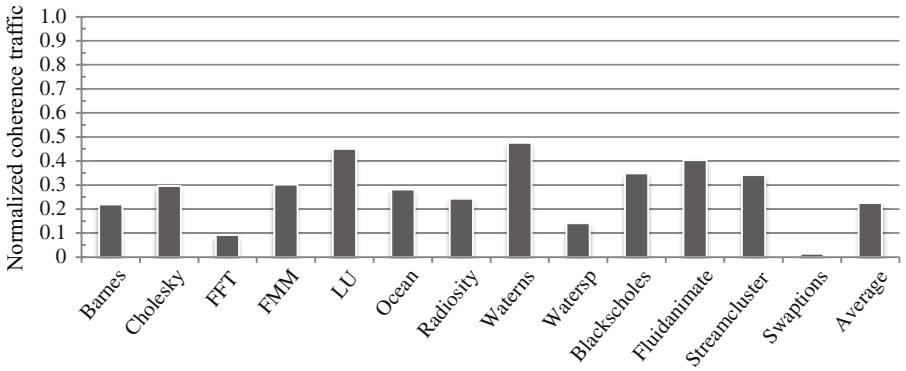

Figure 9 Normalized coherence traffic.

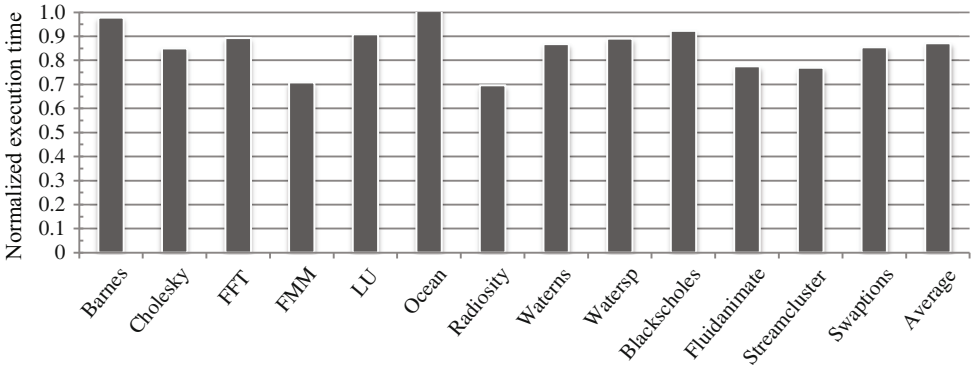

Figure 10 Normalized execution time with respect to the baseline system.

The coherence overhead is also measured. In fact, as our approach incurs less directory cache eviction and cache coherence maintenance, the traffic caused by coherence message transmission is also reduced. Figure 9 shows the normalized coherence traffic of our approach with respect to the baseline system. We can see from Figure 8 that the coherence traffic is drastically reduced, about 77% on average.

4.4 Application execution time

Reduction in cache miss rate and lower coherence protocol overhead (i.e., lower access latency because of coherence bypassing and lower coherence traffic) has very positive effect on application execution performance. Figure 10 shows the normalized execution time of our benchmark programs with respect to that on the baseline system. On average, the execution time of our approach is 13% shorter than the referenced baseline system, which proves that the proposed scheme can improve application performance significantly. We also use the Non-Parametric Test introduced in [19] to test our speedup, this method validate our speedup with a confidence of 0.95.

In addition, we evaluated our system under different system configurations. The parameters we adjusted include the capacity of L2 and L3 caches and the number of cores. In our experiments, the L2 size takes the value of 128 k, 256 k, and 512 k. The capacity of L3 cache is set to 512 k, 1 M, 2 M, 4 M, 8 M, 16 M, and 32 M. The number of cores is from 4 cores to 64 cores that doubles in every successive experiment. FFT is used as the benchmark. The experiment results are shown in Figure 11.

Figure 11 (a) and (b) show the execution time of FFT while changing the size of L2 and L3 cache, respectively. From the results we can tell that our approach outperforms the baseline system in all cache size settings with a constant gain in execution time, while changing L2/L3 cache size has little influence to FFTs execution time in both proposed and baseline systems. The reason is that FFT has a relatively small working set which can fit well to the smallest cache size setting and increase in the cache size does not bring much benefit in performance improvement. The gain obtained by our approach mainly

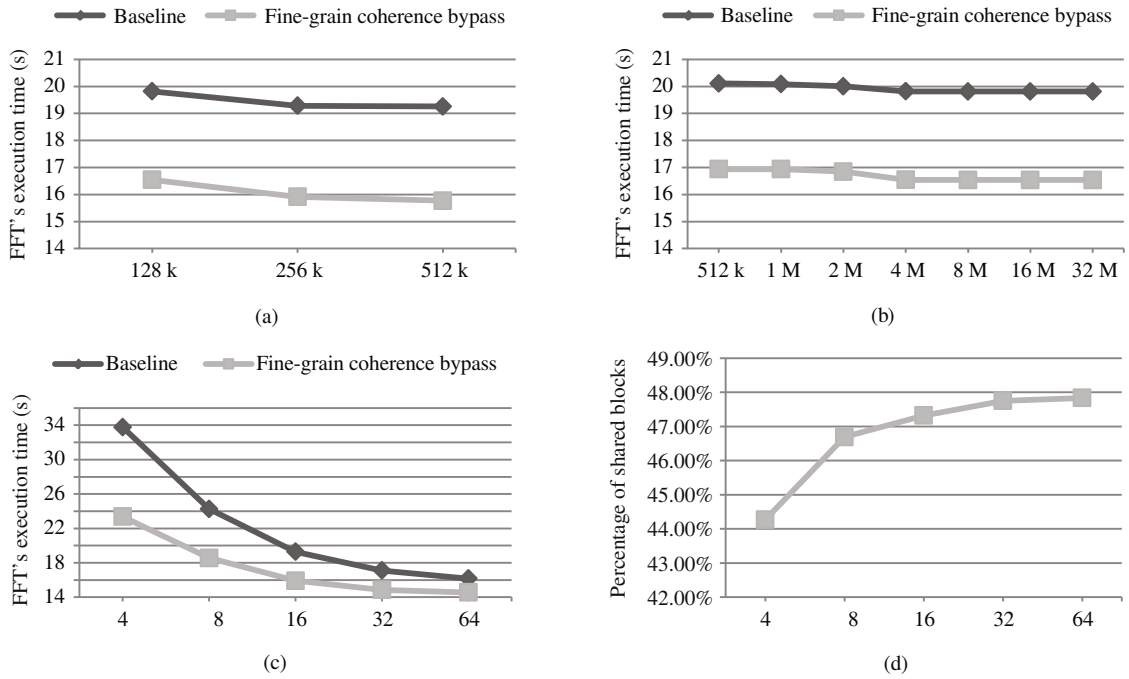

Figure 11 FFT's execution time versus the size of L2/L3 and the number of cores. (a) L2 size; (b) L3 size; (c) core number; (d) core number.

comes from coherence bypassing in accessing the privates blocks, which is constant because the number of private blocks does not change with the size of the cache.

Figure 11(c) shows the effect of changing the number of cores. In this case, the performance gain of our approach is no longer constant. When the number is small (4 cores for example), our approach gets a large gain (23.3 s versus 33.8 s). While increasing the number of cores, the performance gain decreases and eventually becomes stable (14.6 s versus 16.2 s at 64 cores). The reason for that phenomenon is a little bit subtle. It is to be noted that our criterion for classifying private and shared blocks is that a private block will only be accessed by a single core. When the system consists of a small number of cores, the working data set will be partitioned into a few subsets. A large portion of the working set will be allocated to and accessed by one core. The number of private blocks will be large because they are accessed by only one core according to our criterion. So more block accesses will bypass the coherence procedure and involve lower latency. When the number of cores increases, the working set will be partitioned into a larger number of subsets, some of the subsets will be accessed by more than one cores. This results in the increase of the number of shared blocks detected. Consequently, fewer blocks will be considered as private and the benefit obtained by coherence bypassing decreases. We can learn this effect very clearly from Figure 11 (c) and (d). Though the total execution time of FFT is decreasing because of higher computing power when there is increase in the number of cores, the performance gain of our approach diminishes for the reason explained above.

The directory is only needed to track the shared blocks, thus the directory cache size could be reduced while obtaining similar performance. The bars in Figure 12 labeled as DC, DC:2 and DC:4 represent three configurations with a full, a half and a fourth of the baseline directory cache size respectively. We can see from the figure that our mechanism allows us to reduce the size of the directory cache to half of the original size while maintaining the similar application execution time (on average) of the baseline system.

4.5 Energy consumption

Our mechanism has been able to reduce the energy consumption. The energy consumed can be split into two parts: the dynamic energy and the leakage energy. The dynamic energy is mainly consumed by

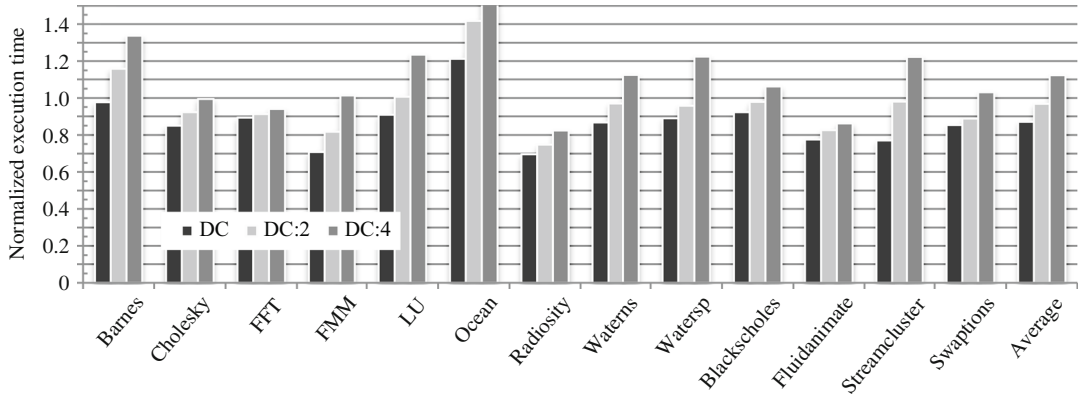

Figure 12 Execution time normalized to the baseline system. DC, DC:2 and DC:4 stand for directory caches with their size divided by 1, 2, and 4, respectively.

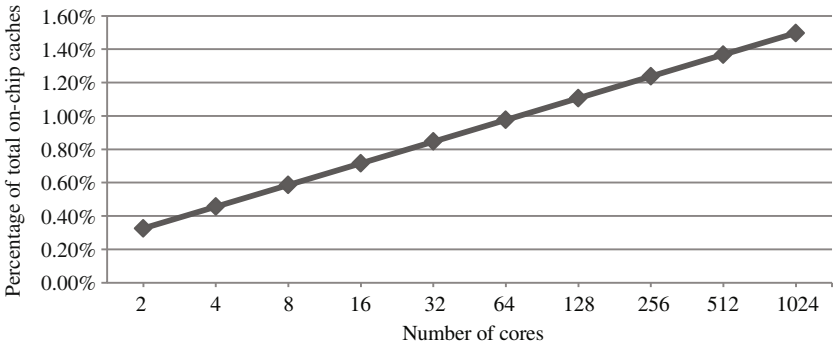

Figure 13 Storage cost for extra flags.

the cache access and on-chip network traffic. As discussed above, our mechanism could reduce the cache misses and network traffics, thus reduce the dynamic energy. The leakage energy is directly proportional to the reduction in execution time, which could be achieved through our mechanism simultaneously. With respect to directory cache, its leakage energy reduction depends on the application's execution time and its size. Thus, we can reduce its leakage energy consumption when the directory cache is only half of the size as in the baseline system.

4.6 Hardware cost

The hardware cost of our approach consists of two parts: the storage cost for storing the extra flags and the extra hardware logic in L3 cache controller. In our proposal the structures of the L2 and L3 caches are modified to include additional flags, that is, the *isShared* flag added to every L2 and L3 cache block and the *coreId* bits added to every L3 cache block. The total storage cost can calculated by the following formula (in bits):

$$\frac{S_{\mathrm{L2}} \times N + S_{\mathrm{LLC}}}{S_{\mathrm{block}}} \times 1 + \frac{S_{\mathrm{LLC}}}{S_{\mathrm{block}}} \times \log_2 N. \tag{1}$$

In the formula, N is the number of cores, S_{L2} and S_{LLC} is the size of private L2 cache and shared L3 cache respectively, while S_{block} represent the cache block size. By applying the formula to our 32 core baseline system, the storage cost for extra flags was found to be 128 KB, which is about 0.52% of the total caches (all private L2 caches and the shared L3, regardless of L1) capacity. For a system consisting 1024 cores, each core has a 256 KB private L2 cache, and the shared L3 cache is of 512 KB, the storage cost as per our approach is about 11.5 MB, which is only 1.5% of the total on-chip cache capacity and acceptable for modern many-core processor design. The storage cost of our approach is depicted in Figure 13. The storage cost can be further reduced by adopting other mechanism such as bloom filters in

implementation, or combining with the coherence state (i.e., use 'I' state to indicate a block is private), which will be our future research work.

The second part of cost for implementing our scheme is the extra hardware in the L3 cache controller for detecting the shared blocks. Using the implementation scheme described in Subsection 3.6, the hardware cost involves mainly comparators and multiplexers. For a 32-core system with 16-way L3 shared cache, we need sixteen 5-bit comparators and a 16-way multiplexer to implement the $coreId$ comparison logic, the hardware overhead is relatively small and acceptable compared to the original L3 cache controller implementation.

5 Conclusion

In order to improve the performance of multicore processors that adopt directory-based cache coherence, we propose a novel hardware-supported approach for fine-grain coherence bypassing. Our approach is based on the fact that a large percentage of memory blocks are accessed by only one core during their lifetime and can be treated as private blocks requiring no coherence maintenance. By dynamically distinguishing private and shared blocks, we are able to bypass the coherence procedure when accessing the private memory blocks, which not only reduces the access latency, but also lowers the cache miss rate. In our approach the directory cache no longer allocates entries for the private blocks for coherence purpose, so more entries of the directory cache can be allocated to the shared blocks. This also improves the scalability of the directory-based cache coherence protocol. The overall performance of the multicore processor adopting our approach can be improved significantly. Our experimental results show that, on average, the proposed approach can avoid coherence tracking for 54% of accessed memory blocks, reduce the coherence traffic overhead by 77%, removes 8% L2 cache misses, and shortens the execution time of benchmark programs by 13%.

A few issues could be addressed to extend our scheme further. First, the current proposal considers only unidirectional transition of the block state, that is, from private to shared. Once a block is found to be shared, it will remain that state even though later on it may be accessed by only one core again. If we could detect when a block returns to private from the shared state, coherence bypassing can be resumed. Second, the $isShared$ flag has some overlap in function with the flag in the coherence protocol like MESI. It might be merged with the original flag to result in a more integrated new coherence protocol. The storage space required for extra flags can be reduced. Third, mechanisms such as bloom filters can be introduced in implementation of our scheme to further reduce the hardware cost. The aforesaid topics will be taken up for our future studies.

Acknowledgements

This work was supported by National High-tech R&D Program of China (863) (Grant No. 2012AA010902), and National Natural Science Foundation of China (Grant Nos. 61073011, 61133004, 61202425). We thank the reviewers for their critical and expert comments that help us to improve the paper.

References

1 Hu W W, Wang J, Gao X, et al. Godson-3: a scalable multicore RISC processor with x86 emulation. IEEE Micro, 2009, 29: 17–29

2 Hu W W, Wang R, Chen Y J, et al. Godson-3B: a 1 GHz 40 W 8-core 128 GFlops processor in 65 nm CMOS. In: Proceedings of the 58th IEEE International Solid-State Circuits Conference (ISSCC'11), San Francisco, 2011. 76–78

3 Marty M R, Hill M D. Virtual hierarchies to support server consolidation. In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07). New York: ACM, 2007. 46–56

4 Gupta A, Weber W D, Mowry T. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In: Proceedings of International Conference on Parallel Processing (ICPP'90). New York: Springer, 1990. 312–321

5 Marty M R. Cache coherence techniques for multicore processors. Dissertation for the Doctoral Degree. Madison: University of Wisconsin-Madison, 2008

6 Hardavellas N, Ferdman M, Falsafi B, et al. Reactive NUCA: near-optimal block placement and replication in distributed caches. In: Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09). New York: ACM, 2009. 184–195

7 Hossain H, Dwarkadas S, Huang M C. POPS: coherence protocol optimization for both private and shared data. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'11), Galveston, 2011. 45–55

8 Zhao H Z, Shriraman A, Dwarkadas S, et al. SPATL: honey, I shrunk the coherence directory. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'11), Galveston, 2011. 33–44

9 Zebchuk J, Srinivasan V, Qureshi M K, et al. Tagless coherence directory. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09). New York: ACM, 2009. 423–434

10 Zhang Y R, Ding W, Liu J, et al. Optimizing data layouts for parallel computation on multicores. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'11), Galveston, 2011. 143–154

11 Cuesta B A, Ros A, Gmez M F, et al. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In: Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11). New York: ACM, 2011. 93–104

12 Cantin J F, Lipasti M H, Smith J E. Improving multiprocessor performance with coarse-grain coherence tracking. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05). New York: ACM, 2005. 246–257

13 Moshovos A. RegionScout: exploiting coarse grain sharing in snoop-based coherence. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05). New York: ACM, 2005. 234–245

14 Zebchuk J, Safi E, Moshovos A. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07). Washington: IEEE Computer Society, 2007. 314–327

15 Zeffer H, Radovi Z, Karlsson M, et al. TMA: a trap-based memory architecture. In: Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06). New York: ACM, 2006. 259–268

16 Zeffer H, Hagersten E. A case for low-complexity MP architectures. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07). New York: ACM, 2007. 10–16

17 Woo S C, Ohara M, Torrie E, et al. The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95). New York: ACM, 1995. 24–36

18 Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmarks suite: Characterization and architectural implications. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'08), Toronto, 2008. 72–81

19 Chen T S, Chen Y J, Guo Q, et al. Statistical performance comparisons of computers. In: Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture (HPCA'12). Washington: IEEE Computer Society, 2012. 1–12