BFDir: A Space-Efficient Coherence Directory Based on Bloom Filter

Jicheng Chen, Yaqian Zhao(✉), Hongzhi Shi, and Yihan Li

State Key Laboratory of High-End Server and Storage Technology,
Inspur Group Company Limited, Beijing, China
{chenjch,zhaoyaqian,shihzh,liyihan}@inspur.com

Abstract. Directory-based coherence is widely used in modern CMP systems. As the number of cores increases, it is increasingly deemed as the only candidate for on-chip cache coherence maintaining. However, limitations of traditional coherence directory pose serious challenges to deal with the ever-increasing size of the system. The hardware overhead and redundant message broadcasting problems dramatically degrade the scalability and performance of the system. In this paper, a space-efficient coherence directory BFDir is proposed. The directory dramatically reduces the directory size as the share list is shortened by Bloom filter. Also, it does not incur message broadcasting as that in limited directories. The evaluation results show, for 32-core CMP systems, compared to full-map directory, 59% overhead of share list can be avoided at the expense of 2.77% performance loss on average; compared to 16-bit coarse directory, 22% overhead of share list can be avoided at the expense of 0.16% average performance loss on average; compared to 8-bit coarse directory, 48% invalid messages are saved and the performance is improved by 2.31%.

Keywords: CMP · Cache coherence directory · Bloom filter · Space-efficient

1 Introduction

CMP processors, which contain multiple processing cores on a single chip have become popular design choice for high-performance processors. Currently, 8-core CMPs are commonplace on the mobile processor and server processor markets. Cores integrated in a single processor will continue to increase with the development of semiconductor technology, and tens or maybe hundreds of cores may be integrated in future CMP processors. In CMP systems, the shared memory programming paradigm is the key component to exploit the performance. To support the paradigm, the cache coherence across all cores in the system has to be maintained with cache coherence protocol (abbreviated as CC protocol).

Basically, all CC protocols fall into two categories: snooping and directory-based protocols. The snooping protocol is often used in CMP systems with few cores (such as IBM Power6 [1]) as it does not scale well with the increase of core number. The directory-based protocols use a hardware structure (called directory) to record coherence

© Springer Nature Singapore Pte Ltd. 2016
W. Xu et al. (Eds.): NCCET 2016, CCIS 666, pp. 81–93, 2016.
DOI: 10.1007/978-981-10-3159-5_8

information to avoid the message broadcasting in snooping protocols, but the hardware cost of the directory grows as the square of the number of cores.

In this paper, we propose a space-efficient coherence directory BFDir to reduce the hardware overhead of the directory structure and avoid the message broad-casting in snooping protocol. In the directory implementation, each core of the system is mapped to multiple positions of the share list by several hash functions. BFDir has two advantages. Firstly, mapping one core to multiple positions in the share list reduces the size of the share list and thus saves the directory space significantly. Secondly, as the share list records all potential sharers of a memory block, it will never overflow and no broadcasting thus exists, which will benefit the performance of the system.

2 Related Work

The basic idea of directory protocol is to establish a directory that maintains a global view of the coherence state of each block. Each entry of the directory has a share list to record which core has a valid copy of a memory block. Full-map directory is a typical implementation of coherence directory, in which each entry is allocated to each memory block in the memory and each bit of the share list is assigned to one core of the system [2]. It is simple, but its share list is redundant [3]. To reduce the length of share list, many directory designs are proposed, such as limited directory [4], linked directory [5–7] and coarse directory [8].

A limited directory only records fixed number of sharer pointers (for example 5 sharers). Its drawback is pointer overflow. When the number of sharers of a memory block exceeds the pointer number, extra operations (such as broadcasting messages to all other cores) have to be executed. Moreover, as the number of cores grows, the share list overflows tend to occur more frequently, which degrades the performance significantly. In linked directory, share information is distributed to a number of small-scale local directory. It compresses the share list, but increases the implementation complexity. Coarse directory reduces the size of share list by a set-associative structure. However, it yields redundant message broadcasting, since each bit of the share list represents a set tag.

During the last few years, some proposals that compressed tagless or sharing patterns have been presented to optimize the design of directory. Jason et al. proposed the Tagless Coherence Directory (TL), a scalable directory structure based on a grid of Bloom filters [9]. SPACE [10] stores the sharing patterns table instead of share list. SPATL [11] combines Tagless and SPACE. Lei et al. presented an area-efficient coherence directory which uses hybrid representation of share list [12]. These methods has made more changes to the directory structure and cache coherence protocol, although they significantly reduce the directory overhead.

Compared with these existing works, our proposal has several advantages. Firstly, only tiny modifications are made to the directory structure, which facilitates the management of the directory and reduces the overhead. Secondly, it is easy to match the coherence protocol with the directory. Lastly, our directory can scale well as the system grows to include more cores and processors.

3 The Methodology

In the section, we will first describe the idea of BFDir, and then explain why it is a good candidate to implement share list of coherent directory. Lastly, we describe the directory design and the protocol operations related to the directory.

3.1 Bloom filter

The bloom filter is a space-efficient, probabilistic data structure, designed to test whether an element is a member of a set [13]. It is implemented with a binary-array, and multiple hash functions as shown in Fig. 1. Each of the functions hashes some element of the target set to one position of the array. An element is determined to be a member of the set only when all the positions where it maps in the array are set.

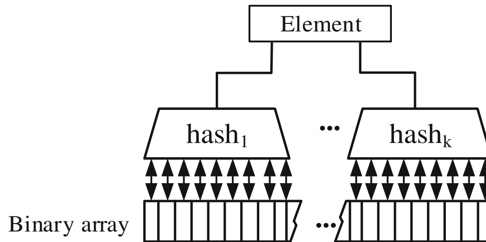


Fig. 1. A sample implementation of bloom filter.

Bloom filter is a good candidate to implement an efficient coherence directory due to two facts:

– Firstly, it is a space-efficient data structure. It is possible to map *n* cores (processors) to *m*-bit binary array, where n can be much larger than m. For example, if *n* equals 32 and the number of hash functions is 2, 10-bit array is enough to record the sharers ($C_{10}^{2} = 45 > 32$).
– Secondly, using bloom filter can avoid the overflow of the share list, and thus get rid of the time-consuming message broadcasting.

The disadvantage of bloom filter is mapping conflict, which can result in false sharing and error removing.

– False sharing occurs when an element is not a member of the set but the query returns true. Figure 2 shows a simple example of false sharing in Bloom filter, where the elements C0, C1 and C2 are mapped to position (0, 4), (2, 4) and (0, 2) respectively. When C1 and C2 are sharers, the position 0, 2, 4 are set to 1. The query for sharers will return C0, C1 and C2, while C0 is not a real sharer. Fortunately, the false sharing only affects the performance but not the correctness of the protocol. Hence, in typical coherence protocol implementations, such as the classical MESI protocol, false sharing is allowed. Moreover, the false sharing can be

reduced and the side effect can be mitigated by sophisticated design of the hash function and choosing a proper size of the array.

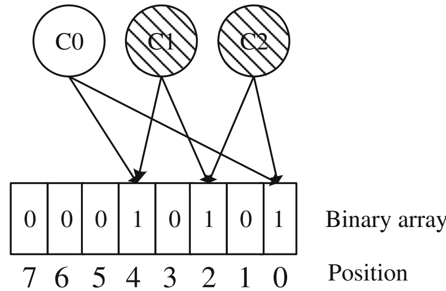


Fig. 2. An example of false positive in bloom filter. E0, E1 and E2 are elements, and E1 and E2 are in the set and shaded.

– Error removing is another challenge of bloom filter. It is safe to clear the bit and remove the element only when all elements in the set don't map to the same position. To safely remove an element, counting bloom filter [14] is proposed, which replaces each bit of the binary array with a counter. Compared with classical bloom filter, counting bloom filter significantly increases the size of the binary array. If 2-bit counters are used, the size of binary arrays is doubled. The added bits could be better utilized to increase the size of the binary arrays which reduce the mapping conflicts.

3.2 Directory Design Based on Bloom Filter

The coherence directory based on bloom filter (BFDir) contains two state bits and the share list based on bloom filter, as shown in Fig. 3. The share list provides a space-efficient structure which uses multiple different hash functions to map one core to several bits of the share list. In the BFDir, the design of the hash functions is critical and it can be implemented in the following three ways.

– With hard-wired logic. The function of the logic is to produce k numbers smaller than m, where k and m are the number of the hash functions and size of the share list respectively. Every time a core becomes a sharer, feed it to the logic and generate k numbers which are used to index into the share list and set the corresponding bits to 1. The hard-wired logic implementation is quite efficient and the latency is small. The drawback is that the logic cannot be programmed and it is impossible to change the mapping to fit the need of the system.
– As a lookup table. Each entry of the table is related to a core and k bits of it are set to 1. Every time a core becomes a sharer, the table is looked up and the entry related to the core is located and the k positions indicated by the entry are set to 1. The drawback of this method is that the hardware cost to implement a lookup table is high. Also, the lookup process can be time-consuming and the latency is large. The advantage is that the mapping can be changed.

– With registers. It is possible to use one or several registers to store the positions where each core is mapped. Every time a core becomes a sharer, the register or register block related to the core is used to set the share list. This implementation can be very time-efficient and flexible. The mapping can be easily changed by writing new values to the registers. The limitation is that enough registers must be provided to record the mapping, which may not be a problem as the FPGA chip used to implement the directory controller usually have plenty of register resources.

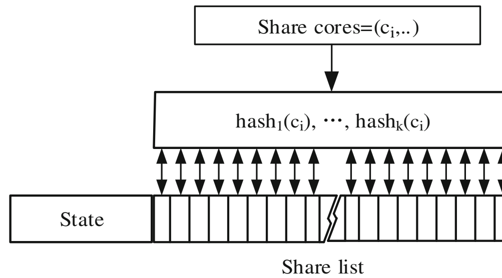


Fig. 3. The coherence directory based on bloom filter.

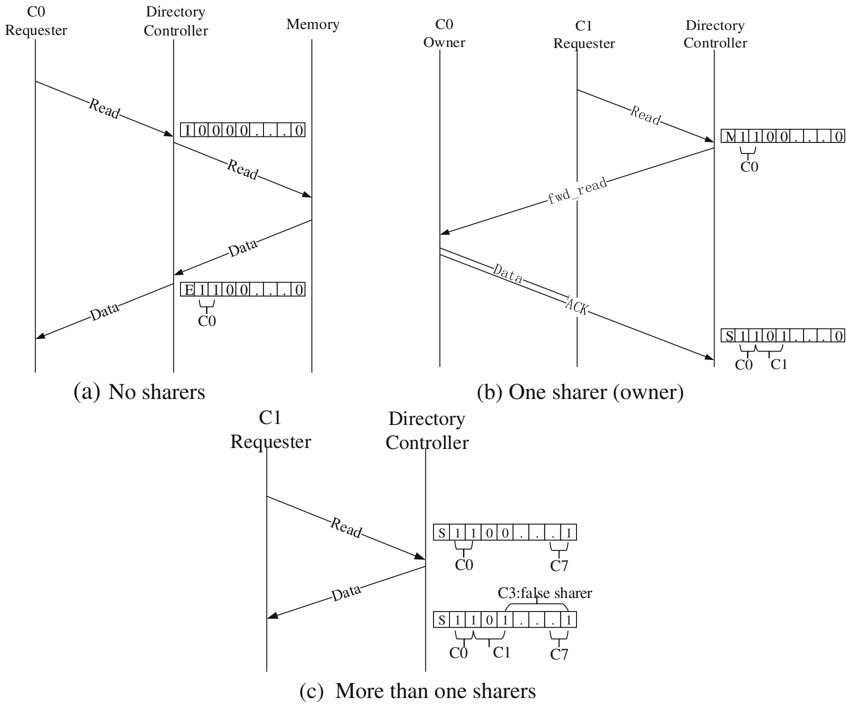
The size of the share list is also important. If it is too small, the probability of false sharing is high, which degrades the performance; if it is too large, the hardware cost of the directory structure increases. Suppose there are n cores in the system, the share list is m-bit long and k hash functions are used in the Bloom filter implementation. In such system, the condition $C_m^k \geq n$ must be satisfied; otherwise, more than one cores have identical mapping.

3.3 Protocol Operations

As a BFDir only records which cores probably share the requested block, share list may have many false sharers. To handle them, we use a lazy way, in which the share list is left unchanged until one core asks for an exclusive permission. The cost to lazily update the share list is that redundant messages needs to be sent, which may increase traffic in the interconnection network and yield negative impact on the performance. However, our evaluation shows that the performance loss is acceptable.

- Read Requests
  When the directory controller receives a read request, the directory controller corresponding to the block can be in one of three states:
  – The share list records no sharers for the requested block. This is to say, no core holds a valid copy of the block. The directory controller forwards the request to memory. The requester is added to the share list after receiving data.
  – The share list records only one sharer for the requested block. That is to say, the block is in E/M state and only one core (termed as owner) holds a valid copy. The directory controller forwards the request to the owner. The requester is added to the share list after receiving data from owner.

– The share list records more than one sharers for the requested block. The directory controller returns the valid copy to the requester and adds the requester to the share list. As shown in Fig. 4(c), there may be false sharer as the new sharer added.



(a) No sharers

(b) One sharer (owner)

(c) More than one sharers

Fig. 4. The change of directory in read processing

- Write Requests

  When a core (the writer) needs to write a block, it issues a request to the directory. Once the write request arrives the directory controller, there are three cases:

  – The share list records no sharers for the block. The directory controller asks the memory to provide the data directly to the writer and records the writer in the share list.

  – The share list records exactly one sharer (the owner) for the block. The directory controller forwards the request to the owner. The owner provides the data to the writer and invalidate the local copy. After the writer received data, the directory controller deletes the owner from share list and adds the writer to the share list.

  – The share list records more than one sharers for the block. The directory controller returns the data to the writer and sends invalid messages to all sharers. After the writer received data, the directory controller deletes all sharers and adds the writer to the share list.

As shown in Fig. 5(c), error removing never happens in BFDir, since all sharers are deleted in each writing process. So that classical bloom filter rather than counting bloom

filter is used to design the share list. However, false sharing may exist in BFDir, and leads to redundant invalid messages. To reduce the occurrence of false sharing, the probability that each bit of the share list is mapped to 1 should be equal. Therefore, the hash functions in BFDir should satisfy the following conditions:

(1) $C_m^k \geq n$

(2) $P\left(hash_j^l(c_i) = 1\right) = \dfrac{n \times k}{m},\ i = 1, 2, \ldots, n; j = 1, 2, \ldots, k; l = 1, 2, \ldots, m$

Where n is the number of cores, k is the number of hash functions, m is the length of share list, much smaller than n. Since there is a power increase of C_m^k with m, hash function space grows larger as n increases, with the result that the probability of false sharing will be reduced, and the system performance will be improved.

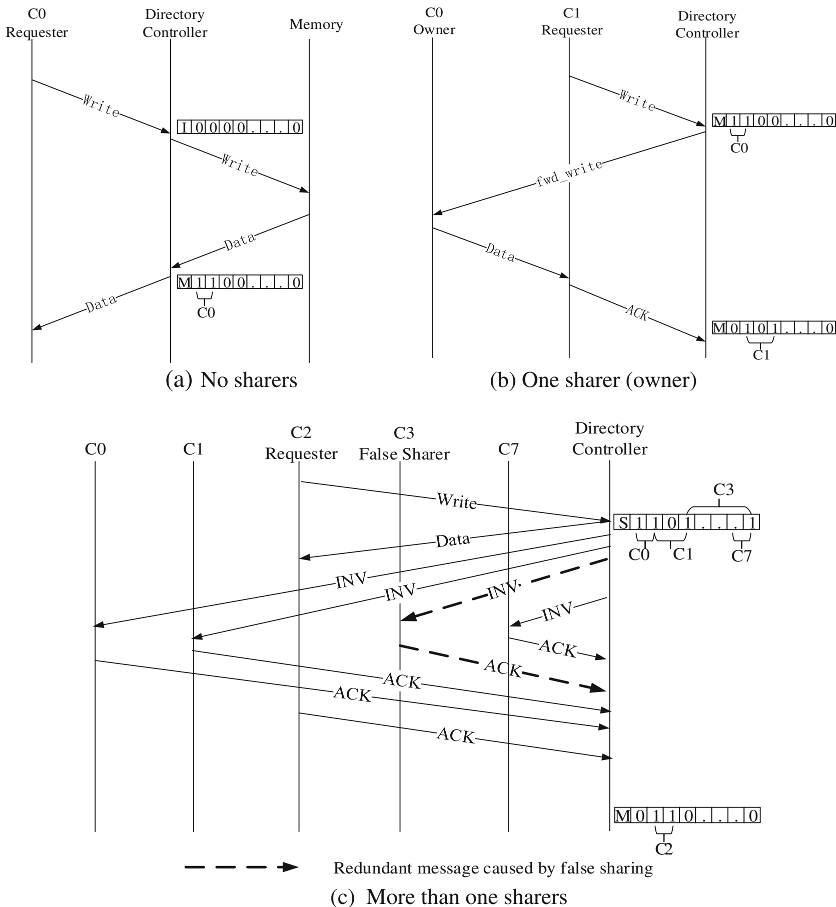


Fig. 5. The change of directory in write processing

4 Evaluation

In this section, we evaluate the performance of BFDir with typical parallel benchmarks. The performance is compared with those collected in a full-map directory and a coarse directory implementation respectively.

4.1 Experimental Setup

The quantitative evaluation is performed on an extensively modified version of gem5 simulation toolkit [15]. In the paper, we modified the Ruby model to support coherence protocol based on BFDir implementation. The structure of the CMP system used in the evaluation is depicted in Fig. 6, in which all cores have private L1 data/instruction caches and shared L2 cache.

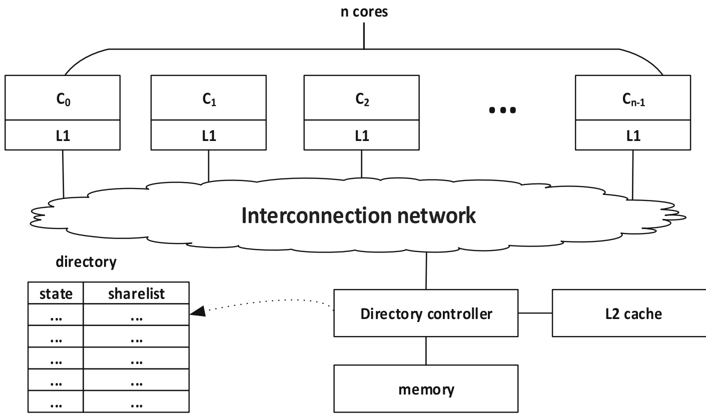


Fig. 6. The structure of the CMP system.

Overall configurations of the simulator are listed in Table 1. Other configurations not mentioned in the table use the default values provided by the simulator, such as the size of the router buffer. We choose a directory based MESI protocol to evaluate BFDir design because it is commonly used in many commercial processors, such as the Xeon MP series.

In coarse directory, the length of the share list are set to 8-bit and 16-bit, respectively. In BFDir, the length of the share list are set to 12-bit. The hash functions in BFDir are designed as follows: (1) 32 options are selected randomly from total $C_{12}^{2} = 66$ mapping options; (2) each bit maps some set element to one of the 12 array positions with a uniform random distribution. It is a fair solution but can be improved by more sophisticated design.

Table 1. Simulator configurations in the evaluation

Item	Value	Item	Value
Number of CPUs	32	Cache line size	64 KB
ISA	Alpha	L1 data cache size	64 KB
CPU model	Timing	L1 instruction cache size	32 KB
Simulation mode	FS mode	Associativity of L1 cache	4 ways
Frequency of CPU	1 GHz	L1 access latency	1 cycle
Topology of network	Crossbar	L2 Cache size	256 KB
CC protocol	MESI	Associativity of L2 Cache	8 ways
Memory size	4 GB	L2 access latency	8 cycles
Memory access latency	100 cycles	Cache replace Policy	PSEUDO LRU

4.2 Benchmarks

The benchmarks used in the evaluation is from SPLASH-2 [16] and PARSEC [17] benchmark suite, which are probably the most commonly used suite for scientific studies of parallel machines with shared memory. All the benchmarks are compiled to alpha ISA with gcc cross-compiler. And O3 optimization option is used in the compilation.

The simulation ticks (referred as *ST*) and number of invalid messages for memory blocks in NP/I (Not Present or Invalid) state (refer to as N_{inv}) of the four directory implementations are collected for all the benchmarks respectively. The performance loss of directory *A* is evaluated as

$$P_l = \frac{ST_A}{ST_{FM}} - 1 \tag{1}$$

Table 2. Benchmarks

Benchmark		Description	Problem Size
SPLASH-2	FFT	Performs 1D fast Fourier transform using six-step FFT method	262, 144 data points
	radix	An integer radix sort	n524288
	barnes	Imlements the Barnes-Hut method to simulate the interaction of a system of bodies.	16348
	ocean_ns	Studies the role of eddy and boundary currents in influencing large-scale ocean movements (contiguos partitions)	130 * 130
	ocean_sq	Studies the role of eddy and doundary currents in influencing large-scale ocean movements (non_contiguos partitions)	130 * 130
	Cholesky	Performs blocked Cholesky Factorization on a sparse matrix	Tk15.o
PARSEC	Bodytrack	Body tracking of a person	1000
	ferret	Content similarity search server	Small size

Where ST_A and ST_{FM} denote the simulation ticks when directory A and full-map directory are adopted respectively. The relative number of invalid messages of directory A is defined as

$$RN_{inv-A} = N_{inv-A} \big/ N_{inv-FM} \tag{2}$$

Where $N_{inv\text{-}A}$ and $N_{inv\text{-}FM}$ denote Ninv when directory A and full-map directory are adopted respectively (Table 2).

4.3 Performance Results

From Table 3, it can be seen that there are 64 M memory blocks. Therefore, the total size of directory is 272 MB in full-map directory, 80 MB in 8-bit coarse directory, 144 MB in 16-bit coarse directory, and 122 MB in BFDir. Compared to full-mapped directory, BFDir saves 160 MB directory space, which accounts for 59% of the total overhead of the full-map directory structure.

Table 3. Comparison of the size of directory

	Share list (bits)	Directory entry (bits)	Total size of directory (MB)
Full-map directory	32	34	272
8-bit Coarse directory	8	10	80
16-bit Coarse directory	16	18	144
BFDir	12	14	112

As shown in Fig. 7, the performance loss of BFDir compared to full-map directory is mostly within 4% in the 32-core system. For ferret, the performance loss of BFDir is negligible, only 0.907%, less than that of 16-bit coarse directory. While for ocean_ns, the performance loss is relative large, about 4.599%. This is due to the fact that during the execution, there are many data sharings among cores and the Write-after-Reads are frequent which incurs a lot of redundant invalid messages and increases network congestion and enlarges the latencies of data accesses. So the performance loss of coarse directory is also large. From Fig. 7, it is also clear that, the performance loss varies for different programs, such as cholesky and ocean_sq. This is due to the fact that, each program has its own data access pattern, which mostly relies on the characteristics of the algorithm the program implements and how the algorithm is programmed. Overall, BFDir with 12 bits share list is comparable to coarse directory with 16 bits share list in performance. This means that BFDir can avoid more unnecessary message broadcasting than coarse directory.

Figure 8 shows the relative number of invalid messages during the execution of the benchmarks. It is noted that, Fig. 8 depicts the relative number of invalid messages for blocks in NP/I state only, which are rare in full-map directory, so the ratio seems quite significant. Compared to full-map directory, both BFDir and coarse directory incur redundant invalid messages. For BFDir, the number of relative invalid messages is below 9, except for cholesky; for 16-bit coarse directory, the number of relative invalid messages is below 8, except for cholesky; while for 8-bit coarse directory, the number is much larger,

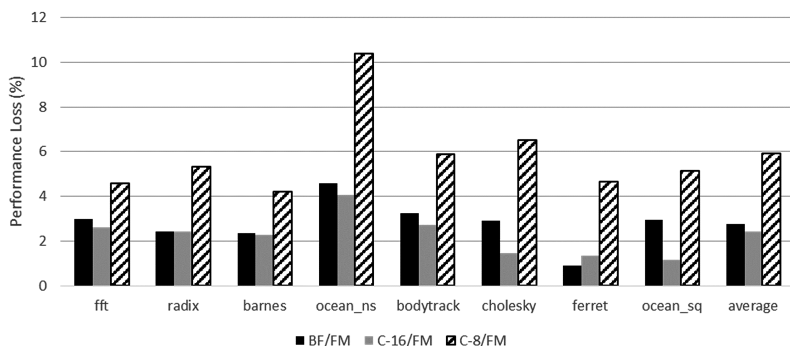


Fig. 7. Comparison of performance loss to full-map directory. BF, C-16, C-8 and FM represent BFDir, 16-bit coarse directory, 8-bit coarse directory and full-map directory respectively.

even larger than 19. Compared to 8-bit coarse directory, our BFDir reduces 46% invalid messages with 41% improvement on the size of directory, while 16-bit coarse directory reduces 55% invalid messages with 83% improvement on the size of directory. Using BFDir implementation can dramatically reduce blind message broadcasting. This is the reason why BFDir achieves better performance, as shown in Fig. 7.

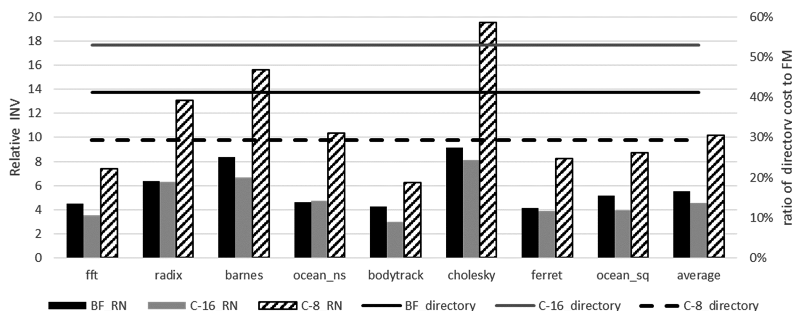


Fig. 8. Comparison of relative invalid messages and directory cost to full-map directory

By comparing the results in Figs. 7 and 8, it is shown that reducing the number of redundant messages broadcasting can benefit to improve system performance. However, besides the number of invalid messages for blocks in NP/I state, there are many factors affecting performance, such as the characteristics of the algorithm the program implements. Therefore, performance loss and relative number of invalid messages may have different changes, as presented by ferret.

On average, our BFDir nearly half the number of redundant invalid messages and performance loss in 8-bit coarse directory, and slightly increase the number of redundant invalid messages and performance loss in 16-bit coarse directory. The results confirm the motivation for pursuing better performance by reducing redundant messages.

5 Conclusion and Future Work

The hardware overhead of the full-map directory are very expensive, which makes it impossible to scale to large-scale systems. And other compressed directories incur lots of message broadcasting when the share list overflows, which significantly degrades the performance. In this paper, we propose BFDir, a space-efficient coherence directory design based on bloom filter. BFDir uses bloom filter to map cores or processors in the system to multiple bits of the share list and shorten its size. The results show that BFDir can reduce the hardware overhead of the full-map directory structure by at least 59% for 32-core CMP system at the expanse of average performance loss less than 2.77%, and reduce the hardware overhead of the 16-bit coarse directory by at least 22% at the expanse of performance loss less than 0.16%. Also, it avoids 46% redundant invalid messages and improves 2.31% performance compared to 8-bit coarse directory.

There are several work to do in the future. Firstly, the hash function should be designed more sophisticated. Secondly, more experiments will be carried out to observe the performance of BFDir in terms of scalability, area, and power consumption. Thirdly, current evaluations are carried on systems that use MESI protocol, other coherence protocol such as MOESI will also be studied.

References

1. Le, H.Q.: IBM POWER6 microarchitecture. IBM J. Res. Dev. **51**(6), 639–662 (2007)
2. Chaiken, D., Fields, C., Kurihara, K., et al.: Directory-based cache coherence in large scale multiprocessors. Computer **23**(6), 49–58 (1990)
3. Han, L., An, J., Gao, D., et al.: A survey on cache coherence for tiled many-core processor. In: 2012 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC), Hong Kong, pp. 114–118 (2012)
4. Agarwal, A., Simoni, R., Hennessy, J., et al.: An evaluation of directory schemes for cache coherence. ACM SIGARCH Comput. Archit. News **16**(2), 280–298 (1988). IEEE Computer Society Press
5. Thakkar, S., Dubois, M., Laundrie, A.T., et al.: Scalable shared-memory multiprocessor architectures. Computer **23**(6), 71–74 (1990)
6. Thapar, M., Delagi, B., Flynn, M.: Linked list cache coherence for scalable shared memory multiprocessors. In: Proceedings of 1993 Seventh International Parallel Processing Symposium, Washington, DC, USA, pp. 34–43. IEEE Computer Society (1993)
7. Alnaes, K., Kristiansen, E.H., Gustavson, D.B., et al.: Scalable coherent interface. In: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering (CompEuro 1990), pp. 446–453. IEEE (1990)
8. Gupta, A., Weber, W., Mowry, T.: Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In: Scalable Shared Memory Multiprocessors, pp. 312–321 (1995)
9. Zebchuk, J., Qureshi, M.K., Srinivasan, V., et al.: A tagless coherence directory. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42), pp. 423–434. ACM, New York (2009)
10. Zhao, H., Shriraman, A., Dwarkadas, H.: SPACE: sharing pattern-based directory coherence for multicore scalability. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 135–146 (2010)

11. Zhao, H., Shriraman, A., Dwarkadas, S., et al.: SPATL: honey, I shrunk the coherence directory. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 33–44 (2011)
12. Fang, L., Liu, P., Hu, Q., et al.: Building expressive, area-efficient coherence directories. International Conference on Parallel Architectures and Compilation Techniques, pp. 299–308. IEEE (2013)
13. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
14. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. **8**(3), 281–293 (2000)
15. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., et al.: The gem5 simulator. ACM SIGARCH Computer Arch. News **39**, 1–7 (2011)
16. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of the 22nd International Symposium on Computer Architecture, vol. 23, no. 2, pp. 24–36 (1995)
17. Bagrodia, R., Ameyer, R., Takai, M.: Parsec: a parallel simulation environment for complex systems. IEEE Comput. **31**(10), 77–85 (1998)