J Supercomput (2018) 74:970–993
https://doi.org/10.1007/s11227-017-2174-8

Reducing the second-level cache conflict misses using a set folding technique

Ali Shatnawi[1] · **Mohammad Alsaedeen**[1]

Published online: 1 November 2017

Abstract The cache memory has a direct effect on the performance of a computer system. Instructions and data are fetched from a fast cache instead of a slow memory to save hundreds of cycles. Reducing the cache miss ratio will definitely improve the execution time of an application. In this work, we propose cache memory designs that reduce the number of conflict misses significantly. The proposed designs reduce the conflict misses in the last level multi-way set associative cache. Each set is divided into a group of subsets: the first is referred to as the exclusive subset, and the rest are the shared subsets. The exclusive is configured as a traditional cache where each block is mapped to the set whose index matches the block index. In addition to their standard cache indexing role, the shared subsets are configured to host blocks with different indices. A memory block can be mapped to one subset from the exclusive type or one of multiple subsets from the shared type. Since the proposed technique is based on combining multiple sets of the shared part to form a larger set, that is shared between memory blocks with different indices, we have chosen the name "set folding." The decision as to where to map a memory block depends on the number of misses encountered at each of the potential target sets. To evaluate the proposed design based on the overall hit rate, twenty-three benchmarks from SPEC CPU 2006 were simulated using the SuperESCalar simulator. The proposed designs require a few extra storage bits which adds a small overhead on the hardware complexity in comparison with the conventional cache. However, the proposed designs achieve lower miss rates for most of the benchmarks.

✉ Ali Shatnawi
ali@just.edu.jo

Mohammad Alsaedeen
mkalsaedeen12@cit.just.edu.jo

[1] Department of Computer Engineering, Faculty of Computer and Information Technology, Jordan University of Science and Technology, Irbid 22110, Jordan

Keywords Computer architecture · Cache memory · Cache optimization · Conflict misses

1 Introduction

1.1 Processor-memory gap

The performance of the microprocessor is increasing at a rate that is higher than that of the memory. It has been reported that the processor performance was improving at an annual rate of 60% before 2004 and has been improving at a rate of 20% since 2004. On the other hand, the main memory is improving at a rate of 9% [1]. This rapid development was according to Moore's law which states that the transistor density doubles every 18 months. New transistor technology is smaller and faster and has less power consumption [2]. This results in an increasing gap between the speeds of the processor and the main memory.

The problem of overall latency in any computer system arises from the off-chip location of the memory unit. In other words, the process of data transfer from memory to CPU and vice versa does not occur within the same chip which, essentially, reduces the computer performance [3,4]. To reduce the impact of this problem, computer architects use a mix of SRAM (static random access memory) which is fast and expensive and DRAM (dynamic random access memory) which is cheaper and slower. The combination of different memory types in different layers of the memory is normally referred to as the memory hierarchy. The first two to three layers of the hierarchy which constitute the system cache are typically manufactured using SRAM technology. The need for a cache is becoming inevitable. Using conventional cache may achieve reasonable improvement in terms of the average memory access time (AMAT), but in many cases the amount of improvement may be insufficient. Therefore, further research for finding new designs and structures that efficiently utilize the cache becomes urgently needed in many applications.

The use of memory hierarchy bridges the gap between a fast processor and a relatively slow memory. In many scenarios, it reduces the average memory access times from order of hundreds of cycles to order of tens of cycles. This was the main motivation for using the cache. The cache is designed to store the code and data segments that are most frequently needed by the processor [5,6].

To properly address the cache, the memory address is divided into three parts , the byte offset $b = log_2(blocksize)$, the set index which requires $m = log_2 \frac{cache\,size}{n * block\,size}$, where n is the associativity level, and the tag t.

The average memory access time (AMAT) is a function of the hit time, miss rate, and miss penalty. Theoretically, the reduction in any of these factors will decrease the AMAT, but in many cases the reduction in one may be at the expense of increasing the others. Hence, a trade-off is usually needed. Using a multi-level cache, in general, decreases the miss penalty and consequently decreases the $AMAT = Hit\ time + Miss\ rate * Miss\ penalty$. Adding more levels bridges the speed gap between the fast first-level cache and the memory [7].

1.2 Motivation

The large gap in miss rate between direct-mapped cache and set associative caches inspires a great motivation to make efforts on investigating and researching new schemes and designs to bridge this gap. The hit rate of a fully associative cache is much higher than that of a direct-mapped cache for all benchmarks. This attracts researchers to innovate new designs to bridge this gap. Dispersing data into cache and utilizing all the sets of the cache contributes to reducing the number of conflict misses. In the case when a set is exposed to a high access pressure, more conflict misses may occur. Using a less-accessed set to reduce the pressure on a heavily used one alleviates and reduces the rate of misses resulting from conflicts.

In this work, we propose a new cache structure that is different from the conventional cache. This design reduces the conflict misses of the conventional cache without the need for increasing its associativity. The proposed design depends on dividing the cache into two parts or subsets. The first subset is called the exclusive subset. The exclusive subset is designed to host the blocks that have matching index with the set index. The second subset is called the shared subset. A shared subset can host blocks with matching index, as well blocks with different indices. The choice of the set to host a block depends on the number of misses encountered by each of the two possible sets.

1.3 Paper organization

The rest of the paper is organized as follows. Section 2 presents some related work, Sect. 3 presents the proposed design, Sect. 4 presents the evaluation environment, Sect. 5 presents the experimental results, and Sect. 6 presents the paper conclusion.

2 Related work

The high rate cache conflict miss is one of the major causes of low computer performance. Many researchers have been working to alleviate the performance problem caused by cache misses. One of the great achievements in the cache design to alleviate the problem of conflict misses is the introduction of the victim cache [6]. The victim cache is a small fully associative cache added to the main cache to host some blocks after their eviction from the main cache. It extends the associativity of the sets that suffer from high rates of conflict misses. The main drawback of using a victim cache is the increase in the hit time due to the need to check the fully associative part of the cache in the case of a miss.

In skewed cache [8], each set is related to another one by inverting the most significant bit of the set index. Therefore, the tag stored in each block takes one bit more than the tag stored in the conventional scheme.

In [5], a technique called column-associative cache is proposed. It uses two hash functions: the first uses the set index as the hashing key, whereas the second complements the most significant bit of the set index before using it as the hashing key. That

is, the cache sets are organized in pairs. A set extends to the other set in the pair when a conflict miss occurs.

In [8–11], intelligent indexing functions to reduce the conflict misses are used. The main drawback of these designs is the high power consumption due to the high hardware complexity. In [12,13], the authors used software techniques based on page coloring and bin hopping to optimize the process of block mapping.

In [14], the author proposed a B-cache which attempts to reduce the number of conflict misses by distributing the accesses uniformly over the lines of the direct-mapped cache. To implement this, the author increased the decoder length and used programmable decoders. The index bits are divided into two categories, the most significant bits are programmable using programmable decoders (PD), and the least significant bits are non-programmable using non-programmable decoders (NPD). In the conventional direct-mapped cache, there is a single room for placing a block, but in a B-cache, there are multiple choices for placing a specific block. The B-cache utilizes some of the empty sets to reduce the miss rate at the expense of an increase in the power cost without a significant increase in the access time as compared to a conventional direct-mapped cache.

The proposed solution in [15] is similar to that in [14]. However, in [15], the number of bits for the cache index is increased. In [14], it is possible to use a large number of bits for indexing to accomplish the sought objective.

In [16], a V-way cache attempts to reduce the conflict misses by allowing the use of more tags than the available physical sets. Pointers are used to associate the tag with its actual set. A set balancing cache in [17] outperforms the V-way approach by performing a superior set balancing mechanism. Set balancing moves lines from stressed sets to some underutilized ones.

The two methods in [18,19] use XOR-based mapping schemes to obtain pseudo-random block placement. Having a random distribution of memory blocks over the cache sets statistically reduces the rate of conflict misses. The difficulty to predict the pattern of memory references in the general case makes the XOR-based mapping function hard to implement. Hence, it could work for some applications and not for others.

The prime modulo and prime displacement indexing functions achieve better dispersion with high cost due to their complex calculations [9]. The prime modulo and prime displacement render some cache sets unused. This waste is a result of using a prime number that is less than the number of sets as the modulus. The use of prime modulo reduces the rate of conflict misses, yet it may not be used in caches existing in the critical execution path like the first-level cache. It could, however, be used in large higher level caches.

The author of [8] proposed a skewed-associative cache. It is a two-way cache which uses a separate hashing function for each way. The hashing function is based on XORing two bits from the block address. This scheme cannot easily implement the LRU replacement policy because the blocks in a cache set do not share the same index.

In [20], the author showed that if certain bits are used in the indexing process, the miss rate can be reduced. To determine the index bits, a heuristic was proposed. The indexing function is determined for each application at design time to optimize

its performance at run time. This technique performs well only if all applications are known in advance. In fact, this was the intention of the authors, to optimize the design for a specific set of applications.

There are many other techniques developed to reduce the miss rate. Examples of these techniques include cache bypassing [21] , cache miss classification and isolation [22], reconfigurable associativity and dynamic cache partitioning [23–25], and page remapping and coloring at run time [26,27].

In [28,29], the use of thrashing-avoidance cache (TAC) increases the hit rate significantly when applied on C++-based applications. It is to noted that C++-based applications suffer from more cache conflict misses than C-based application. This is due to that fact that frequent procedure calls/returns in object-oriented programs reduces the effective locality of reference and increases the chances of jumping between blocks that have the same index. It has been shown in [30] that C++ based applications execute almost seven times more calls than traditional C programs. In TAC, the instructions are split into groups, each being associated with one of the cache banks. In data cache, the convention of the store-/load-dependent data cache (SLDC) is introduced. In SLDC, the address distance is used as the main criterion for distributing data over the different cache segments. Then, a group of related segments is virtually gathered in each bank of the cache in order to increase the locality of reference and consequently increase the hit rate.

In [15], a technique based on adaptive selection of cache indexing bits (ASCIB) is proposed. It dynamically alternates the bits used for set indexing in a way to reduce conflict misses. Choosing the appropriate indexing bits during run time can successfully change the working set during the execution of the application. These changes lead to a better dispersion of the working set over the available sets. Each iteration of ASCIB algorithm is divided into three phases: the bit-victimization phase, the bit-selection phase, and the idle phase. In bit-victimization phase, the algorithm chooses the worst bit of the set index bits that does not help in dispersing the working set fairly on the available sets. The bit is victimized based on its low entropy and high correlation. In the bit-selection phase, the algorithm selects the appropriate bit of the tag to replace the victimized bit. The selection of this bit helps disperse the working set fairly over the available sets. Finally, in the idle phase, the algorithm does not perform any action. It, however, results in power saving. In ASCIB, analyzing more bits results in avoiding more conflict misses. However, this increases the area size, the latency, and the power consumed.

The authors of [31] proposed a new technique called WS-DAM to dynamically increase the associativity of the sets suffering more conflict misses. Expanding the associativity of some sets does not mean increasing the cache size. This is attained by moving some of the ways from the lightly used sets to the heavily used ones. CMP-SVR shows, on average, 6.63% improvement in execution time (CPI) and 14.46% improvement in miss rate.

In [32], they proposed CMP-SVR technique to dynamically increase the associativity of most pressure sets without increasing the cache size. The last level cache (LLC) is divided into two parts: reserve storage (RT) and normal storage (NT). The sets are divided into fellow groups. In other words, each set has its own fellow and can use its reserve ways to increase its associativity during the execution. An extra tag array

(SA-TGS) is added to RT to facilitate the searching process with less expensive cost. The associativity of SA-TGS depends on the number of sets in a fellow group and the number of reserve ways per set. CMP-SVR shows about 8% improvement in cycles per instruction and 28% improvement in miss rate.

In [33], the author proposed new techniques to address the problem of conflict misses in multi-level cache. Two placement strategies were proposed: least XOR and full XOR for multi-level caches. These techniques reduce the opportunity of two addresses to conflict with each other at multiple places in multi-cache system. This leads to improving the global miss rate, which is more important indicator of performance than the local miss rate. These techniques show 10–20% improvement in L2 and L3 cache miss rates, without any extra hardware.

In [34], the authors proposed a new technique for reducing both conflict and capacity misses. This technique depends on changing the placement and eviction policies of the cache. The decision of placement is taken based on the criteria of the set saturation level (SSL) that measures the degree of a set ability to hold its working set. This technique is called Bimodal Set Balancing Cache and requires only less than 1% of storage overhead. Bimodal Set Balancing Cache shows about 16% improvement in L2 cache and 4.8% improvement in IPC.

The authors of [35] proposed some cache management techniques on the CMP platform. These techniques lead to evenly distribute memory accesses across the sets of the private caches as well as the shared caches. SSBC, PSBC, BP-NUCA, and BP-NUCA+ cache management schemes were proposed based on the single-core scheme SBC [17]. Adapting the SBC to shared caches (SSBC) led to system degradation rather than improvement. Adapting the SBC to private caches (PSBC) shows an improvement of about 2%. Furthermore, the modification of private cache-based techniques BP-NUCA and BP-NUCA+ shows insignificant improvement. In other words, the non-uniformity resulting from the distribution of memory accesses across cache sets of multiprogrammed workloads running on CMPs platforms did not result in significant improvement in the cache performance.

In [36], the authors proposed a new GPU cache indexing technique called full permutation (FUP) which uses multiple metrics to calculate the set index, such as the feature bits and the intra-warp concentration. FUP uses two-level XOR gates for the set index calculation. Experiments showed that FUP outperformed preceding techniques, such as BXOR [18] and pDisp [9].

In [37], the authors introduced the concept of array optimal padding for arbitrary tile sizes in a set associative cache. Padding for nested tiles and multi-level caches is also proposed. Experimental results showed an improvement in terms of reducing conflict misses due to the use of padding in multiple benchmarks.

In [38], a new technique, called pseudorandom interleaving cache (PRIC), to improve the cache indexing function, was developed. This technique is based on polynomial modulus mapping. It contributes to alleviating the associativity stalls and reducing conflict misses.

In [39], a technique for determining the sources of cache conflict misses is proposed. A cache simulator is used as a diagnosis tool to identify the main causes of conflict misses in the cache lines. The results obtained can help produce more efficient indexing functions.

3 The proposed design

3.1 Introduction

The main objective of the proposed cache design is to reduce the rate of conflict misses. The high penalty of conflict misses in the last level cache (LLC) makes the miss rate minimization a critical design objective for many researchers.

Cache designers attempt to satisfy one or more the following criteria: high hit rate, fast access time, small area, small storage overhead, and low power consumption. In general, optimal cache architectures combine the merits of the simple direct-mapped cache with the high hit rate characteristics of set-associated caches. That is, it will be very attractive to increase the associativity level of the cache without increasing its size. The proposed design is intended to extend the n-way set associative cache starting from n equal to two.

In this section, we present details about the proposed designs and their implementations, illustrative figures, the replacement policy, and the amount of memory overhead for each proposed design in comparison with the conventional, skewed, and victim caches.

3.2 Set folding architecture

The set folding architecture is produced by folding multiple sets of different indices in one super set that is shared by more than one set. Hence, we have chosen the name "set folding" for the proposed technique. We present two different architectures based on the proposed idea of set folding, namely single set folding and double set folding. We will further illustrate the architectures of the triple and quad set folding by figures only. The analysis will be based on the general case of set folding design.

3.2.1 Single set folding design

In the single set folding design of a n-way set associative cache, a set is divided into two subsets, each containing $\frac{n}{2}$ blocks, as shown in Fig. 1. The first subset is configured to host the blocks with indices that match the set index, whereas the second one is configured to host blocks with matching indices as well as blocks with the same block index except the most significant bit (MSB). For example, a second subset of index k may contain blocks with index k as well as those with index $(k + 2^{(m-1)})\ mod\ 2^m$, where m is the number of bits associated with the set index, as illustrated in Fig. 2.

The physical address in the proposed design has 32 bits, divided into three parts: byte offset with b bits, set index with m bits and tag with t bits. Thus, we have 2^m sets of n 2^b-byte blocks. For example, when $m = 6$, the second subset of set number 100011 can host blocks with index 100011 and/or with index 000011. In this paper, we will refer to the first cache subsets as the exclusive cache, and the second cache subsets as the shared cache. One extra bit is required for each block in the shared cache. This bit is added to the tag part of the block to distinguish the original set index from the extended block index.

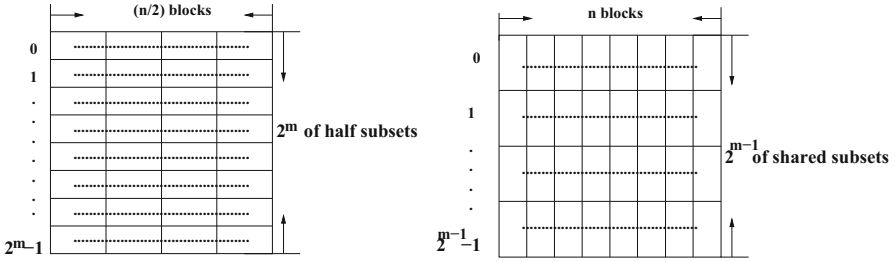

Fig. 1 Set folding cache

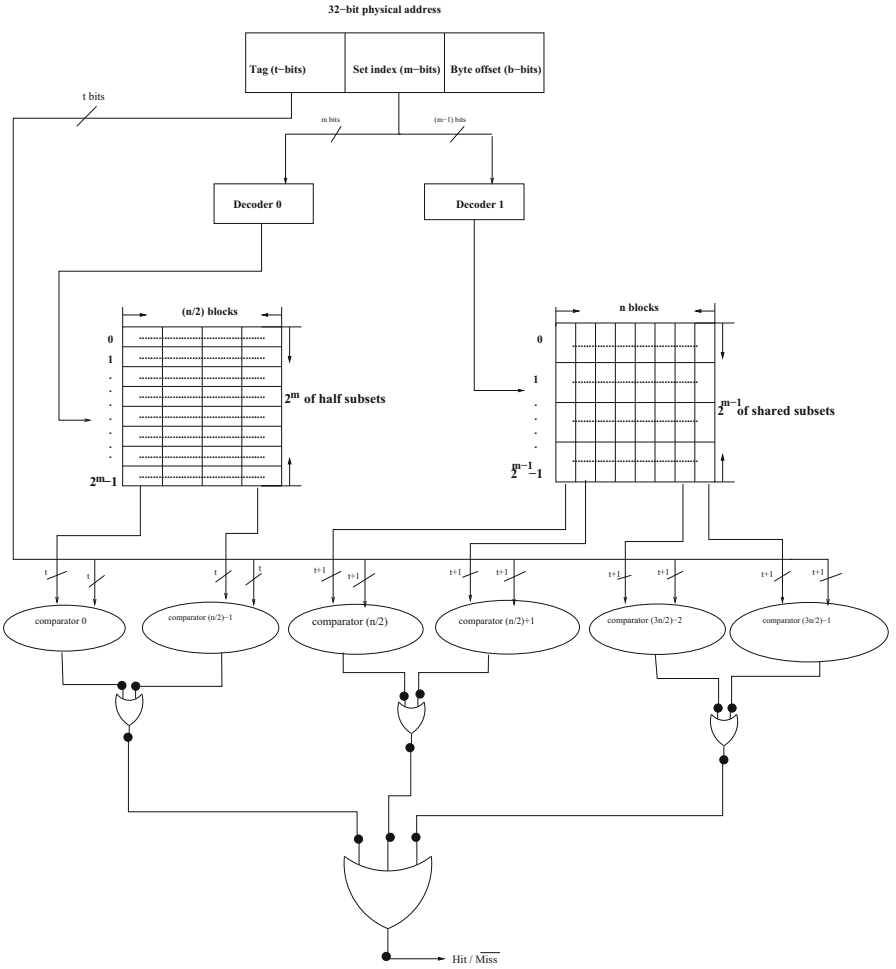

Fig. 2 Set folding design

The number of extra storage bits needed over a conventional cache is very little. For example, if we have a cache with t tag bits and b byte offset bits, one valid bit, one dirty bit, and $8 * 2^b$ data bits in each block, the number of bits needed by the exclusive subset of the cache is calculated as shown in Eq. 1.

$$stored\_bits\_in\_cache = \#ways \times \#sets \times \#bits\_in\_block \tag{1}$$

The number of data bits stored in the exclusive cache is calculated using Eq. 2.

$$\begin{aligned} &stored\_bits\_in\_the\_exclusive\_part \\ &= \frac{n}{2} \times 2^m \times 8 \times 2^b = 4 \times n \times 2^{m+b}. \end{aligned} \tag{2}$$

The memory overhead of the single set folding design is one bit per each extended block, which is equivalent to one bit per every two blocks. Hence, this overhead relative to the data size can be expressed as in Eq. 3.

$$memory\_overhead = \frac{1}{2 \times 8 \times 2^b} = 2^{-(b+4)} \tag{3}$$

For example, the memory overhead for a two-way 1 MB cache with 64-byte block size incurs an overhead of 1 KB which forms less than 0.1% of the cache size.

3.2.2 Double set folding design

In double set folding design, the cache is divided into three parts, as shown in Figs. 3 and 4. The first part (exclusive subset) remains the same as in set folding design. The exclusive subset represents half of the cache. However, each set of the exclusive subsets can receive a block that matches its set index. Therefore, each set contains half the number of blocks as a conventional cache. The second part of this design is Shared Subset 1; half the blocks are similar to the conventional cache, and the other half is designed to allow two possible indices in each set. Every two memory addresses having the same set index, except the MSBit, map to the same Shared Subset 1, as shown in Fig. 3. The third part is referred to as Shared Subset 2. It contains one-fourth the number of sets of the conventional cache, with blocks with 2 different bits in the block index map to the same set. In other words, every four memory addresses having the same set index except the two MSB of their set index can be mapped to the Shared Subset 2 at the same set. However, in this design, the set associativity extends from n-way to $2n$-way of hybrid associativity. That being said, in normal conventional cache, n-way hosts blocks from the same set index, but in this design, it hosts $2n$ blocks from different set indices which gives opportunity to significant reduction in the number of conflict misses. However, the stored tag in exclusive subset is the same as conventional, but the number of stored tag bits in shared subsets is one bit more than the tag stored in conventional cache. Therefore, the number of tag bits stored in the Shared Subset 2 has two bits more than the tag stored in conventional cache, as shown in Fig. 4. The number of comparators needed to operate this design is $2n$ compared

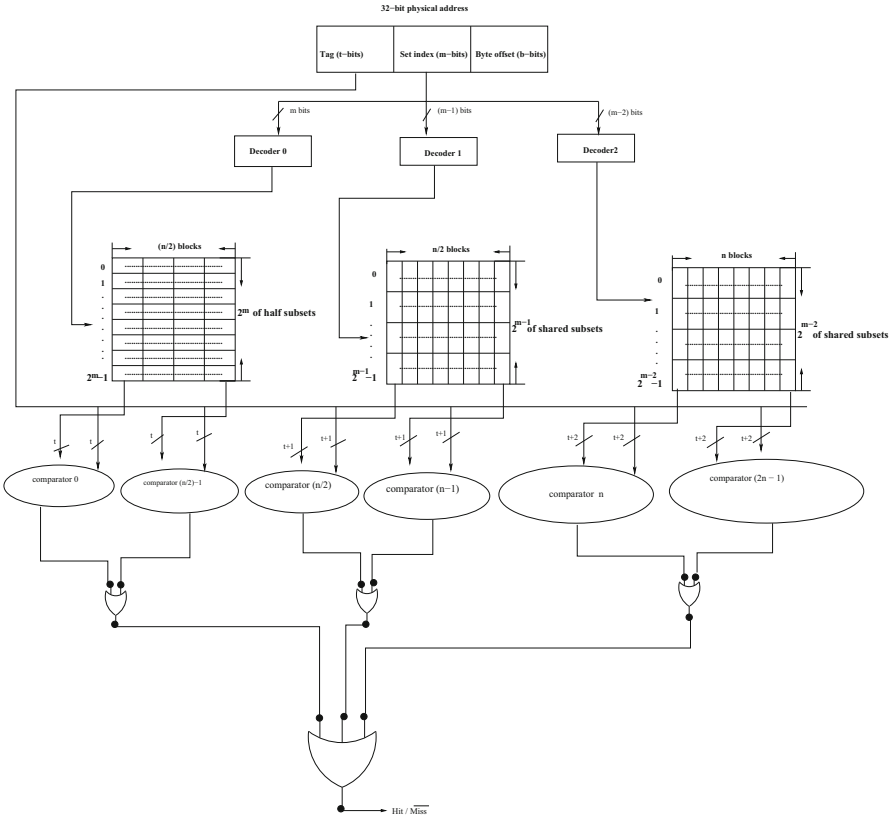

Fig. 3 Double set folding design

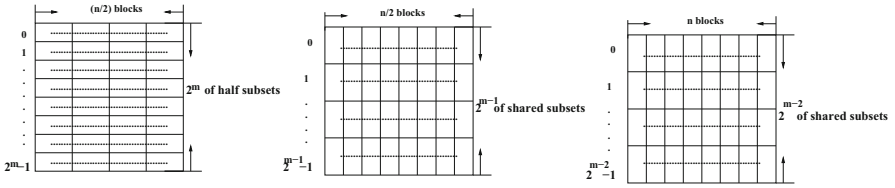

Fig. 4 Double set folding cache

to n in conventional cache, and these comparators are classified into three types based on the number of bits. The first type is used in exclusive subset, the number of this type of comparators is $\frac{n}{2}$, and each of them is a t-bit comparator. The second type is used in Shared Subset 1, its number is $\frac{n}{2}$, and each of them is a $(t+1)$-bit comparator. The third type is used in Shared Subset 2 using n comparators, and each of them is a $(t+2)$-bit comparator. In other words, the hardware overhead is extremely low in comparison with other schemes, such as Victim cache. In this design, the hardware overhead comprises n comparators with only a few extra bits in design. The memory overhead in comparison with the conventional cache can be computed as in Eq. 4.

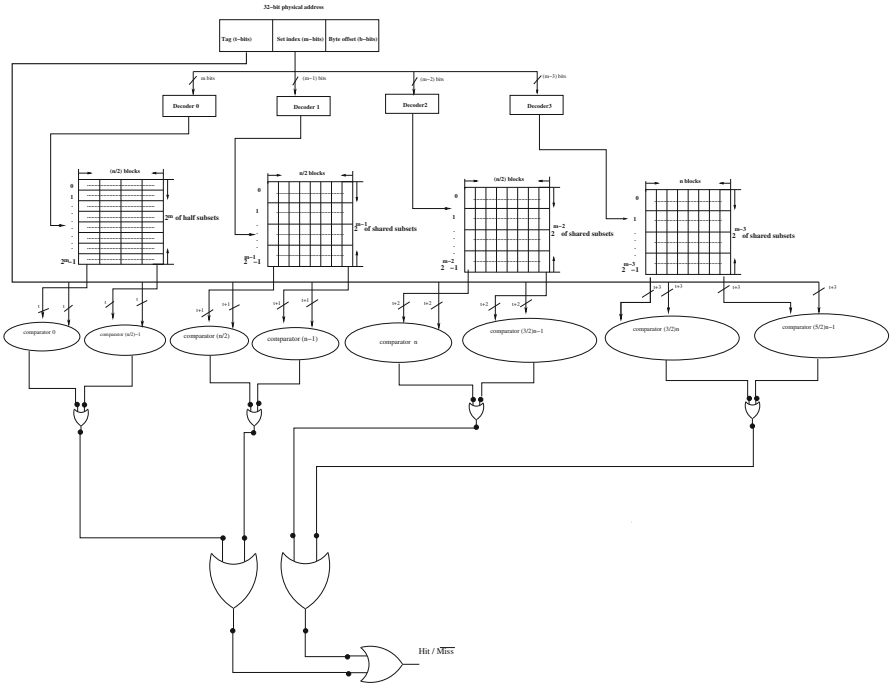

Fig. 5 Triple set folding design

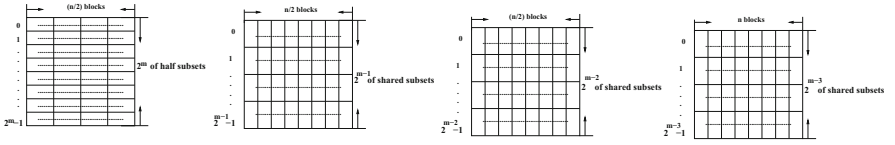

Fig. 6 Triple set folding cache

$$\begin{aligned} &memory\_overhead\_in\_double\_set\_folding\_design \\ &= \frac{1 \times 2^{m-1} \times n/2 + 2 \times 2^{m-2} \times n}{2^m \times n \times (8 \times 2^b)} = 3 \times 2^{-(b+5)} \end{aligned} \tag{4}$$

For example, in a 32-bit address with 64-byte blocks, the storage overhead will be less than 0.15%.

Similarly, the architectures of the triple and quad set folding designs are depicted in Figs. 5, 6, 7 and 8.

3.2.3 Analysis of the general set folding design

In general, let the variable u refer to the value of set folding factor. In other words, u is equal to 1 in the case of a single set folding, 2 in the case of double set folding, and so on. The number of extra bits needed to support the set folding design with set folding factor equal to u is given in Eq. 5.

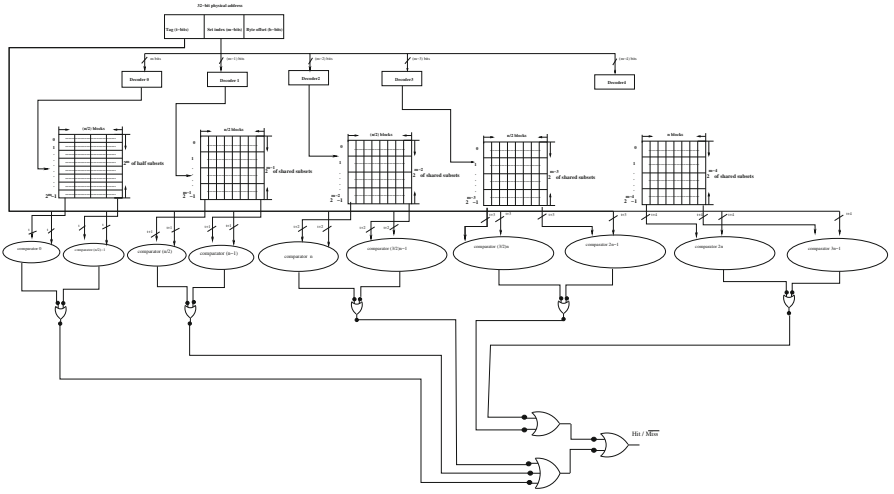

Fig. 7 Quad set folding design

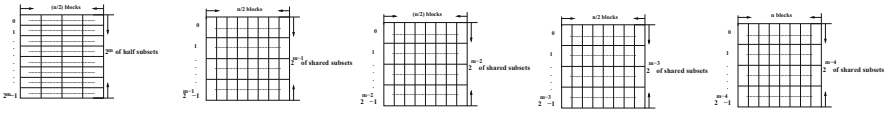

Fig. 8 Quad set folding cache

$$
\begin{aligned}
&extra\_bits\_in\_proposed\_design \\
&= \frac{n}{2} \times 2^{m-1} + 2 \times \frac{n}{2} \times 2^{m-2} + 3 \times \frac{n}{2} \times 2^{m-3} \\
&\quad + \cdots + (u-1) \times \frac{n}{2} \times 2^{m-(u-1)} + u \times n \times 2^{m-u} \\
&= \frac{n}{2} \sum_{i=1}^{u} i \times 2^{m-i} + \frac{n}{2} \times u \times 2^{m-u} \\
&= \frac{n}{2} \times 2^{m} \sum_{i=1}^{u} i \times \left(\frac{1}{2}\right)^{i} + \frac{n}{2} \times u \times 2^{m-u} \\
&= \frac{n}{2} \times 2^{m-1} \sum_{i=1}^{u} i \times \left(\frac{1}{2}\right)^{i-1} + \frac{n}{2} \times u \times 2^{m-u} \\
&= \frac{n}{2} \times 2^{m-1} \left(4 - (2u+4)\left(\frac{1}{2}\right)^{u}\right) + \frac{n}{2} \times u \times 2^{m-u} \\
&= n \times 2^{m}(1 - 2^{-u}) \qquad (5)
\end{aligned}
$$

Hence, the memory overhead with respect to the cache data size is given by Eq. 6.

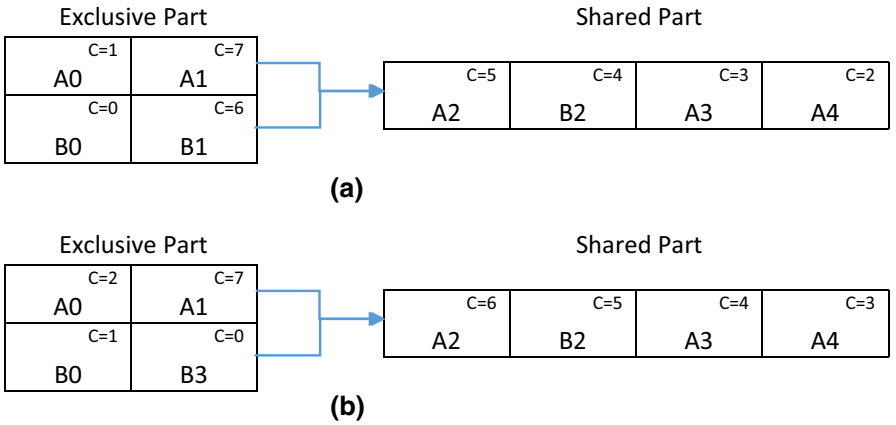

Fig. 9 An example illustrating the placement and replacement policies. **a** After the access sequence A0, B0, A1, B1, A2, B2, A3, A4, A0, B0, **b** after encountering an access form block B3

$$
\begin{aligned}
&memory\_overhead \\
&= \frac{extra\_bits}{total\_size\_in\_bits\_of\_conventional\_cache} \\
&= \frac{n \times 2^m (1 - 2^{-u})}{n \times 2^m \times (8 \times 2^b)} \\
&= (1 - 2^{-u}) \times 2^{-(b+3)}
\end{aligned} \tag{6}
$$

3.3 Cache placement and replacement

If there is a vacant block in the exclusive part of a set, a missed block with a matching index is placed in this vacant block. If all blocks in the exclusive part are occupied and there is a vacant block in the shared part, the missed block is placed in the shared part. Figure 9a illustrates the placement of two groups of blocks, A and B. The blocks of each group have the same set index, the two goups have distinct exclusive indexes, and they both have a common shared index. This example is based on a 4-way set associative cache with single set folding design. The replacement policy is based on the general LRU. The label C refers to the counter used to support the replacement structure. The policy considers the blocks in the exclusive part along with the shared part for replacement without touching the blocks of the exclusive part of other sets. As illustrated in Fig. 9b, when B3 is accessed, the LRU block from the exclusive part of B and the shared of both A and B is replaced, which happened to be B1 as it has the highest counter (A1 is not considered). Hence, B3 replaces B1 and all the counters are adjusted accordingly.

4 Evaluation environment

4.1 SESC simulator

Experiments to demonstrate the effectiveness of the proposed design have been carried out using the SESC microprocessor architectural simulator. This simulator is based on execution-driven simulation environment that supports a dynamic superscalar processor model in which the architecture is simulated cycle by cycle [29,40]. SESC simulates a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern dynamic superscalar processor necessary for accurate simulation. It has the ability to model different processors and memory architectures [41]. SPEC2006 benchmarks are used to demonstrate the effectiveness of the proposed design. All benchmarks enter rabbit mode (fast forwarding) for five billion instructions after that they are simulated for 1 billion committed instructions. All benchmarks are compiled to binary executable code based on the MIPS instruction set architecture. The output of the SESC simulator is verified by comparing with the output generated by the SPEC organization [42–44].

4.2 Benchmarks classification

As shown in Table 1, the given benchmarks are classified into two groups, A and B. Group A consists of the benchmarks whose hit rates, when a direct-mapped cache is used, are lower than those when using a fully associative scheme by more than 20%. Group B benchmarks are those whose hit rates in the case of a direct-mapped cache are within 20% of the those in the case of a fully associative cache. The benchmarks of Group A suffer from intensive cache address conflicts; hence, they would benefit from structures that alleviate conflict misses as a result of extending the effective level of associativity. It is to be noted that Group A consists of five floating point (FP) applications (bwaves, dealII, gromacs, soplex, and sphinx3) and four integer (INT) applications (gobmk, hmmer, omnetpp, and perlbench). On the other hand, the benchmarks of Group B are not conflict intensive and hence less likely to benefit from the proposed designs. Group B consists of six FP applications (calculix, lbm, milc, namd, povray, and zeusmp) and eight INT applications (astar, bzip2, gcc, h264ref, libquantum, mcf, sjeng, and xalan).

The hit rate of the libquantum benchmark is very close to zero as shown in Table 1. This application is known to suffer from a very low hit rate for small size caches as presented in [45]. In our study, however, we only used small size cache to demonstrate the effectiveness of the proposed technique in reducing conflict misses for most of the benchmarks.

4.3 SESC configuration

Table 2 shows the base configuration used to run the SESC simulator.

Table 1 SPEC 2006 applications with their L2 cache miss rates

Benchmark	Direct-mapped hit rate (%)	Fully associative hit rate (%)
Group A		
Bwaves	22.04	50.9
dealII	38.42	81.47
Gobmk	22.84	44.76
gromacs	17.15	39.96
Hmmer	42.24	78.3
omnetpp	12.99	36.41
perlbench	18.46	40.34
Soplex	10.2	41.54
sphinx3	2.61	53.36
Group B		
Astar	78.3	94.3
bzip2	32.17	50.68
calculiux	35.11	51.97
gcc	20.32	38.53
h264ref	38.84	43.65
lbm	3.85	20.72
libquantum	0	0
mcf	4.34	19.07
milc	9.08	23.88
namd	39.53	58.29
povray	29.92	48.88
sjeng	32.96	51.01
xalan	34.84	44.58
zeusmp	23.07	32.02

5 Experimental results

We used twenty-three benchmarks from the SPEC CPU2006, listed in Table 1, to compare the proposed designs with the the conventional cache. Each benchmark contains one billion instructions. The memory address is 32-bit wide. Simulation results have shown good improvement in terms of the hit rate for all the twenty-three benchmarks as will be presented. The impact of the level of associativity on the hit rate for all used benchmarks before applying the proposed technique is depicted in the Figs. 10, 11 and 12.

All speedups in this paper are calculated relative to the conventional cache with the same associativity level. The results can be summarized as follows, which are also depicted in Figs. 13, 14, 15, 16, 17, 18, 19, 20, 21 and 21.

1. *Two-way single set folding scheme* The average speedup for all benchmarks is 22.7% with the peak speed being 65% for the lb benchmark. It has also been noted

Table 2 Parameters of the simulated architecture (base configuration)

Processor	
Number of CPU's	1
Clock rate	3 GHz
Number of architecture bits	32
Out of order	True
Fetch width	4
Issue width	4
Retire width	5
Instruction queue size	24
Functional unit	64 INT, 64 FP, 1 LD, 1 ST
Max pending	LD/ST 32 LD, 32 ST
ROB size	128
Number of registers	128 INT registers, 64 FP registers
Memory	
L1 data	8 KB, 1 way, 64-B line, 2 cycles HT
L1 instruction	8 KB, 1 way, 64-B line, 1 cycle HT
L2 unified	16 KB, X way, 64-B line, 10 cycles HT
Page size	4096
Memory latency	250 ns
Table look-aside buffer	
Instruction TLB	512 B, 64 way, 8-B line, 3 ports, LRU
Data TLB	512 B, 64 way, 8-B line, 3 ports, LRU
Prediction table	
16 KB size, hybrid, 2K entries, 2 way, 1 cycle HT	

that no benchmark was subjected to any slowdown. Figures 13, 14, 15 depict the performance of this design.

2. *Two-way double set folding scheme* The average speedup for all benchmarks is 32.5% with the peak speed being 137% for the dealII benchmark. Figures 13, 14, 15 depict the performance of this design.
3. *Two-way triple set folding scheme* The average speedup for all benchmarks is 24.9% with the peak speed being 80.2% for dealII benchmark. Figures 13, 14, 15 depict the performance of this design.
4. *Two-way quad set folding scheme* The average speedup for all benchmarks is 10.9% with the peak speed being 54.7% for lbm benchmark. Figures 13, 14, and 15 depict the performance of this design.
5. *Four-way single set folding scheme* The average speedup for all benchmarks is 13.1% with the peak speed being 307.2% for the milc benchmark. Figures 16, 17, and 18 depict the performance of this design.
6. *Four-way double set folding scheme* The average speedup for all benchmarks is − 10.8% with the peak speed being 8.4% for the sphinx benchmark. Figures 16, 17, and 18 depict the performance of this design.

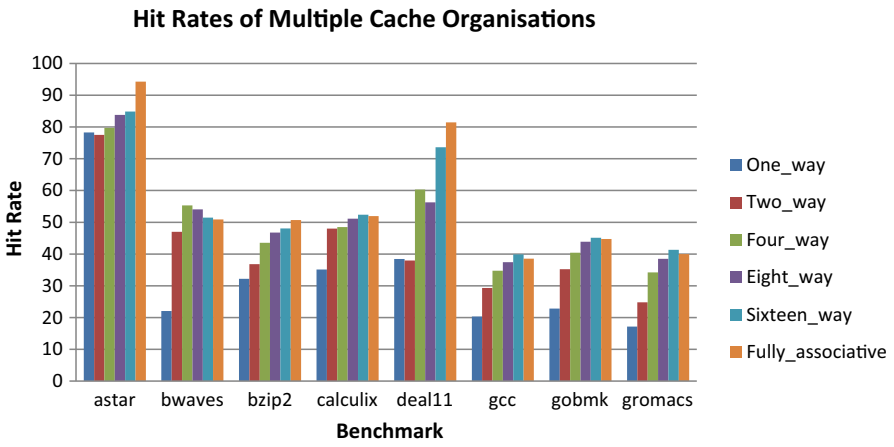

Fig. 10 Hit rates of multiple cache organization (part1)

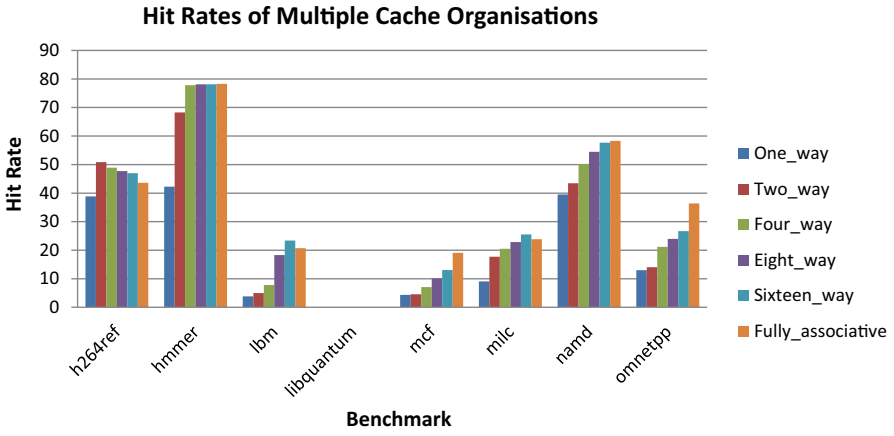

Fig. 11 Hit rates of multiple cache organization (part2)

7. *Four-way triple set folding scheme* The average speedup for all benchmarks is 4.3% with the peak speed being 30.2% for the dealII benchmark. Figures 16, 17, and 18 depict the performance of this design.
8. *Four-way quad set folding scheme* The average speedup for all benchmarks is − 5.9% with the peak speed being 8.1% for the sphinx3 benchmark. Figures 16, 17, and 18 depict the performance of this design.
9. *Eight-way single set folding scheme* The average speedup for all benchmarks is 3.5% with the peak speed being 270.4% for the milc benchmark. Figures 19, 20, and 21 depict the performance of this design.
10. *Eight-way double set folding scheme* The average speedup for all benchmarks is − 15.5% with the peak speed being 8.7% for the dealII benchmark. Figures 19, 20, and 21 depict the performance of this design.

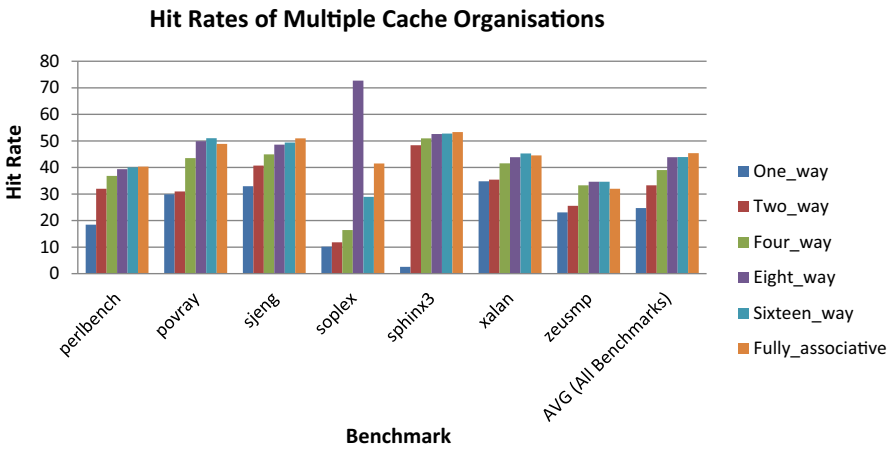

Fig. 12 Hit rates of multiple cache organization (part3)

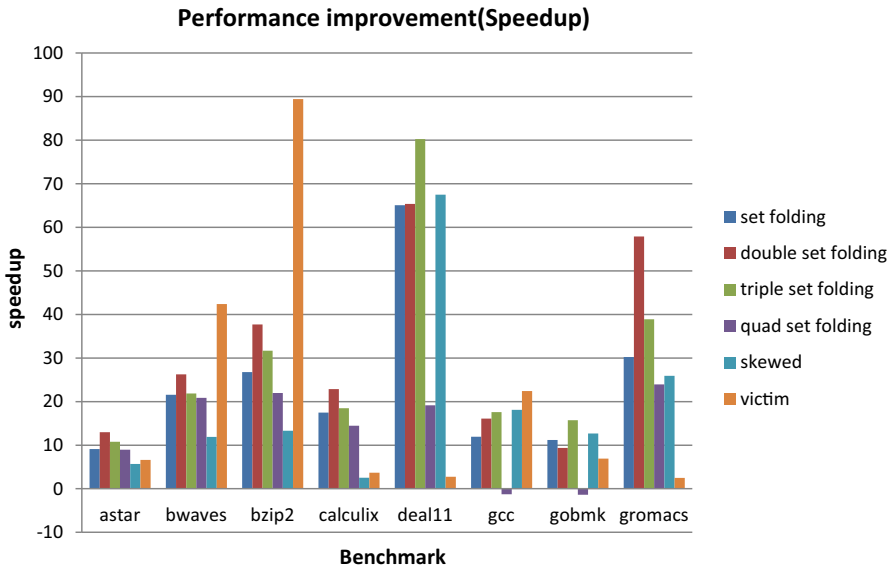

Fig. 13 Performance improvement results of our proposed designs (two-way) (part1)

11. Eight-way triple set folding scheme: The average speedup for all benchmarks is − 6.5% with the peak speed being 45.3% for the dealII benchmark. Figures 19, 20, and 21 depict the performance of this design.
12. *Eight-way quad set folding scheme* The average speedup for all benchmarks is − 15.3% with the peak speed being 71% for the astar benchmark. Figures 19, 20, and 21 depict the performance of this design.

Based on the above analysis, the following observations about the performance in terms of the hit ratio are presented:

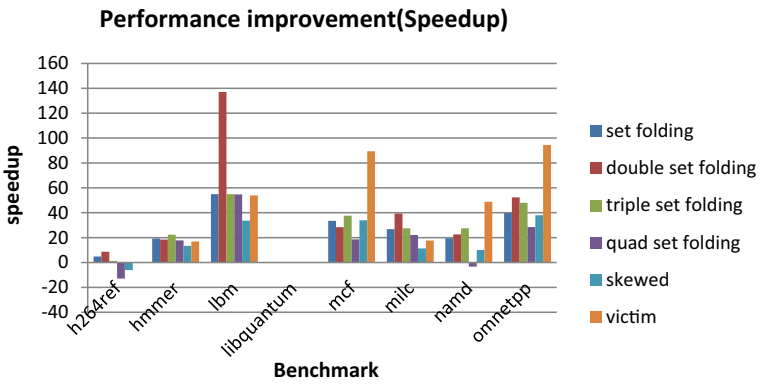

Fig. 14 Performance improvement results of our proposed designs (two-way) (part2)

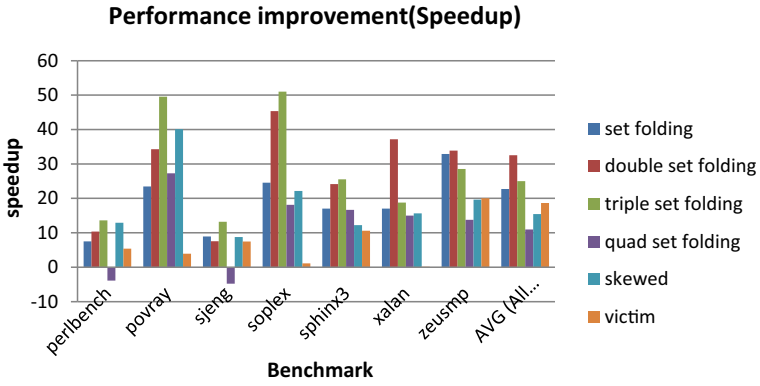

Fig. 15 Performance improvement results of our proposed designs (two-way) (part3)

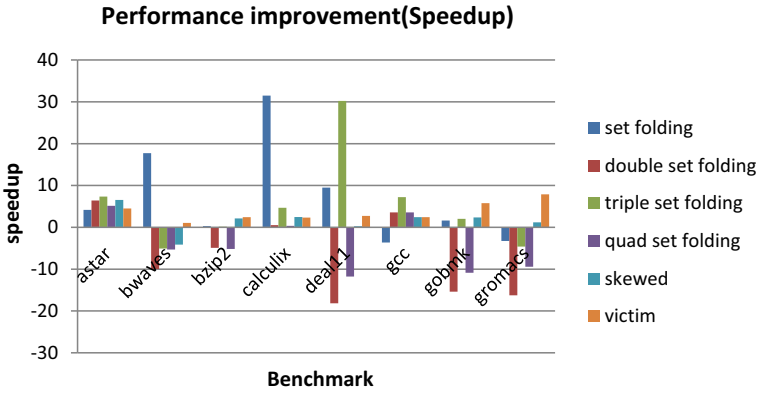

Fig. 16 Performance improvement results of our proposed designs (four-way) (part1)

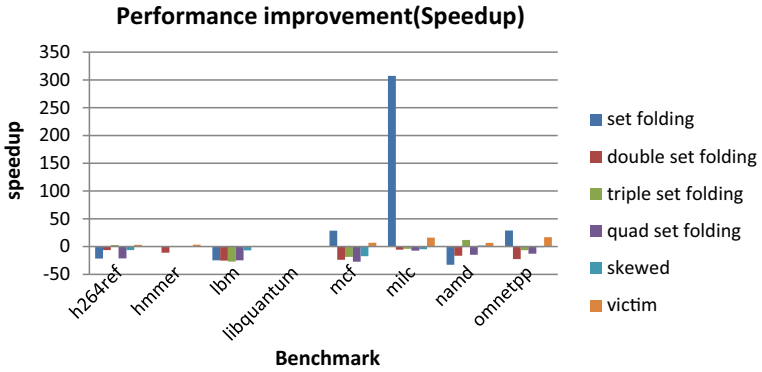

Fig. 17 Performance improvement results of our proposed designs (four-way) (part2)

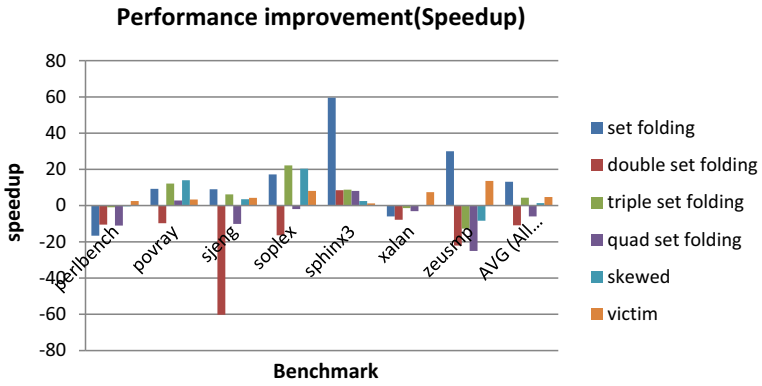

Fig. 18 Performance improvement results of our proposed designs (four-way) (part3)

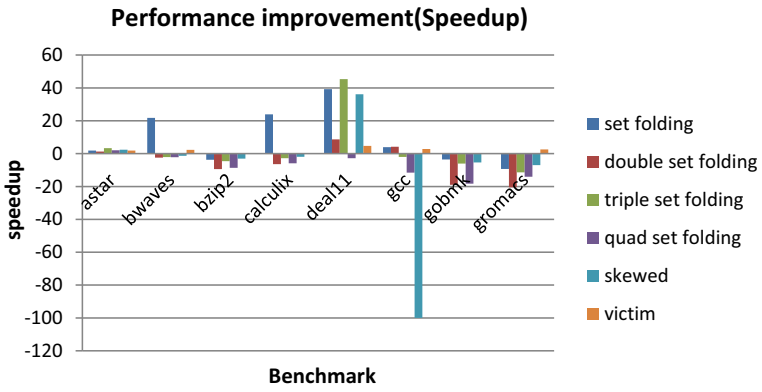

Fig. 19 Performance improvement results of our proposed designs (eight-way) (part1)

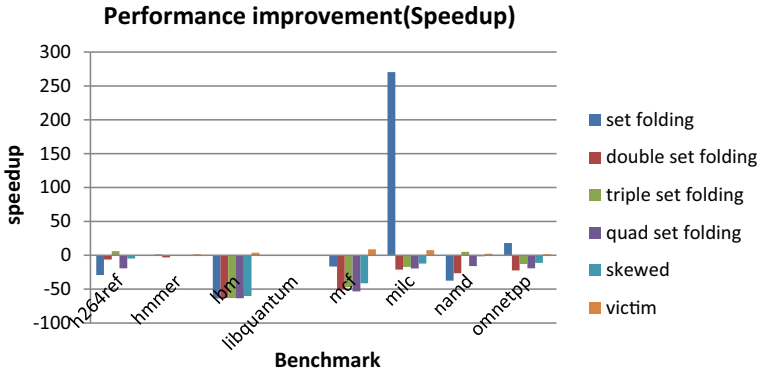

Fig. 20 Performance improvement results of our proposed designs (eight-way) (part2)

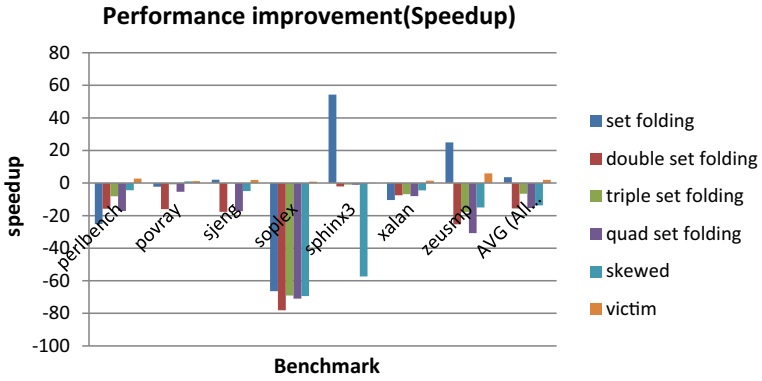

Fig. 21 Performance improvement results of our proposed designs (eight-way) (part3)

1. The two-way double set folding outperforms the two-way single set folding by about 43%.
2. The two-way triple set folding outperforms the two-way single set folding by about 9.6% but, obviously, performs less than the two-way double set folding.
3. Neither of the two-way quad set folding, nor the two-way eight set folding performs as well as the single set folding.
4. For the eight-way, the single set folding outperforms all other folding schemes.

The improvement made by the single set folding over the conventional two-way cache is due to the extension of the effective cache associativity. That is, the two-way associativity is normally inadequate for attaining high hit rate. The four proposed designs when applied to the two-way set associative cache remove significant proportion of conflict misses, thus improving the hit rate. However, the double set folding has achieved the best performance among all other folding levels.

6 Conclusion

In this work, we have proposed cache memory designs that reduce the number of conflict misses significantly.

We have presented a new cache structure that is different from the conventional cache. This design reduces the conflict misses that occur in the conventional caches with a little increase in hardware complexity without the need to increase the cache associativity. We have referred to the proposed design as "set folding" technique. The principle of set folding technique is to divide each set of the given cache into two subsets. The first subset is called the exclusive subset. The exclusive subset can host blocks that have a matching index with the set index. The second subset is called the shared subset, which hosts blocks with a matching index, as well blocks with a different index. Four designs have been derived from the set folding technique, namely single set folding technique, double set folding technique, triple set folding technique, and quad set folding. The single set folding technique has one shared subset and one exclusive subset, the double set folding technique has one exclusive subset and two shared subsets, the triple set folding technique has one exclusive subset and three shared subsets, and the quad set folding technique has one exclusive subset and four shared subsets.

To evaluate the proposed design based on the overall hit rate, twenty-three benchmarks from SPEC CPU 2006 were simulated on SuperEScalar simulator (SESC). The speedup results are summarized as follows. Using a two-way cache, the best folding level is the double set folding, which achieves 32.5% improvement over the conventional two-way cache. For the four-way cache, the best improvement is achieved when the single set folding technique is used, 13.1% improvement. An improvement of 3.5% is achieved when single set folding technique used for the eight-way cache structure. Hence, it can be concluded that the double set folding is the best design among the four designs to alleviate the problem of conflict misses. It can be concluded that the proposed design effectively increases the hit rate by increasing the effective cache set associativity without increasing the cache size. On anther front, the proposed designs do not produce significant power overhead as the extra hardware is extremely small in comparison with the entire cache size.

References

1. Jl H, Da P (2007) Computer architecture: a quantitative approach, vol 4. Kaufmann Publishers, San Francisco
2. Moore GE et al (1998) Cramming more components onto integrated circuits. Proc IEEE 86(1):82–85
3. Kagi A, Goodman JR, Burger D (1996) Memory bandwidth limitations of future microprocessors. In: Computer Architecture, 23rd Annual International Symposium on IEEE 1996, pp 78–78
4. Patt YN, Patel SJ, Evers M, Friendly DH, Stark J (1997) One billion transistors, one uniprocessor, one chip. Computer 30(9):51–57
5. Agarwal A, Pudar SD (1993) Column-associative caches: a technique for reducing the miss rate of direct-mapped caches, vol 21 (2). ASM
6. Jouppi NP (1990) Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: Computer Architecture, Proceedings of the 17th Annual International Symposium on IEEE 1990, pp 364–373
7. Smith AJ (1982) Cache memories. ACM Comput Surv (CSUR) 14(3):473–530

8. Seznec A (1993) A case for two-way skewed-associative caches. In: ACM SIGARCH computer architecture news, vol 21(2). ACM, pp 169–178
9. Kharbutli M, Irwin K, Solihin Y, Lee J (2004) Using prime numbers for cache indexing to eliminate conflict misses. In: Software, IEE Proceedings of IEEE, pp 288–299
10. Bodin F, Seznec A (1997) Skewed-associativity improves performance and enhances predictability. IEEE Trans Comput 46(5):530–544
11. Bodin F, Seznec A (1995) Skewed associativity enhances performance predictability. In: Proceedings of the 22nd annual international symposium on computer architecture, 22–24 June 1995. S. Margherita Ligure, Italy, pp 265–274
12. Bugnion E, Anderson JM, Mowry TC, Rosenblum M, Lam MS (1996) Compiler-directed page coloring for multiprocessors. In: ACM SIGPLAN notices, vol 31(9). ACM, pp 244–255
13. Kessler RE, Hill MD (1992) Page placement algorithms for large real-indexed caches. ACM Trans Comput Syst (TOCS) 10(4):338–359
14. Zhang C (2006) Balanced cache: reducing conflict misses of direct-mapped caches. ACM SIGARCH Compu Archit News 34(2):155–166
15. Ros A, Xekalakis P, Cintra M, Acacio ME, Garcia JM (2015) Adaptive selection of cache indexing bits for removing conflict misses. IEEE Trans Comput 64(6):1534–1547
16. Qureshi MK, Thompson D, Patt YN (2005) The v-way cache: demand-based associativity via global replacement. In: Computer Architecture, ISCA'05. Proceedings of the 32nd International Symposium on IEEE 2005, pp 544–555
17. Rolán D, Fraguela BB, Doallo R (2009) Adaptive line placement with the set balancing cache. In: Microarchitecture, MICRO-42. 42nd Annual IEEE/ACM International Symposium on IEEE 2009, pp 529–540
18. González A, Valero M, Topham N, Parcerisa JM (1997) Eliminating cache conflict misses through xor-based placement functions. In: Proceedings of the 11th International Conference on Supercomputing. ACM, pp 76–83
19. Rau BR (1991) Pseudo-randomly interleaved memory. In: ACM SIGARCH computer architecture news, vol 19(3). ACM, pp 74–83
20. Givargis T (2003) Improved indexing for cache miss reduction in embedded systems. In: Design Automation Conference, Proceedings of IEEE 2003, pp 875–880
21. Johnson TL, Connors DA, Merten MC, Hwu W-M (1999) Run-time cache bypassing. IEEE Trans Comput 48(12):1338–1354
22. Collins JD, Tullsen DM (2001) Runtime identification of cache conflict misses: the adaptive miss buffer. ACM Trans Comput Syst (TOCS) 19(4):413–439
23. Balasubramonian R, Albonesi D, Buyuktosunoglu A, Dwarkadas S (2000) Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture. ACM, pp 245–257
24. Chiou D, Jain P, Devadas S, Rudolph L (2000) Cache partitioning via columnization. In: Proceedings of Design Automation Conference. minus 0.4emCiteseer
25. Ranganathan P, Adve S, Jouppi NP (2000) Reconfigurable caches and their application to media processing, vol 28(2). ACM
26. Bershad BN, Lee D, Romer TH, Chen JB (1994) Avoiding conflict misses dynamically in large direct-mapped caches. In: ACM SIGPLAN notices, vol 29(11). ACM, pp. 158–170
27. Sherwood T, Calder B, Emer J (1999) Reducing cache misses using hardware and software page placement. In: Proceedings of the 13th International Conference on Supercomputing. ACM, pp 155–164
28. Chu Y, Ito MR (2000) The 2-way thrashing-avoidance cache (tac): an efficient instruction cache scheme for object-oriented languages. In: Computer Design, Proceedings of 2000 International Conference on IEEE, pp 93–98
29. Chu Y, Ito M (2001) An efficient instruction cache scheme for object-oriented languages. In: Performance, Computing, and Communications, IEEE International Conference on IEEE, pp 329–336
30. Calder B, Grunwald D, Zorn B (1994) Quantifying behavioral differences between c and c++ programs. J Program Lang 2(4):313–351
31. Das S, Kapoor HK (2015) Dynamic associativity management using utility based way-sharing. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, pp 1919–1924
32. Das S, Kapoor HK (2013) Dynamic associativity management using fellow sets. In: Electronic System Design (ISED), International Symposium on IEEE 2013, pp 133–137

33. Salwan H (2013) Global conflict avoidance using block placement strategies in multi-level caches. In: Information and Communication Technologies (ICT), 2013 IEEE Conference on IEEE, pp 1221–1226
34. Rolán D, Fraguela BB, Doallo R (2010) Reducing capacity and conflict misses using set saturation levels. In: High Performance Computing (HiPC), 2010 International Conference on IEEE, pp 1–9
35. Jia X, Jiang J, Ni X, Zhao T, Qi S, Fu G, Zhang M (2011) Understanding how non-uniform distribution of memory accesses on cache sets affects the system performance of chip multiprocessors. In: Parallel and Distributed Processing with Applications Workshops (ISPAW), Ninth IEEE International Symposium on IEEE 2011, pp 266–272
36. Wang B, Liu Z, Wang X, Yu W (2015) Eliminating intra-warp conflict misses in GPU. In: Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition. EDA Consortium, pp 689–694
37. Hong C, Bao W, Cohen A, Krishnamoorthy S, Pouchet L-N, Rastello F, Ramanujam J, Sadayappan P (2016) Effective padding of multidimensional arrays to avoid cache conflict misses. In: ACM SIGPLAN notices, vol 51(6). ACM, pp 129–144
38. Khairy M, Zahran M, Wassal A (2017) Sacat: streaming-aware conflict-avoiding thrashing-resistant gpgpu cache management scheme. IEEE Trans Parallel Distrib Syst 28(6):1740–1753
39. Sato Y, Endo T (2017) An accurate simulator of cache-line conflicts to exploit the underlying cache performance. In: European Conference on Parallel Processing. Springer, pp 119–133
40. Austin T, Larson E, Ernst D (2002) Simplescalar: an infrastructure for computer system modeling. Computer 35(2):59–67
41. Ortego PM, Sack P (2004) Sesc: Superescalar simulator. In: 17th Euro Micro Conference on Real Time Systems (ECRTS05), pp 1–4
42. Spec cpu benchmarks. http://www.spec.org/benchmarks.html
43. Spec cpu2000. http://www.spec.org/cpu2000
44. Spec cpu2006. http://www.spec.org/cpu2006
45. Beckmann N, Sanchez D (2015) Talus: a simple way to remove cliffs in cache performance. In: High Performance Computer Architecture (HPCA), IEEE 21st International Symposium on IEEE 2015, pp 64–75