Computers and Electrical Engineering 38 (2012) 243–257

Contents lists available at SciVerse ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

A comparative analysis of performance improvement schemes for cache memories ☆

Krishna Kavi *, Izuchukwu Nwachukwu, Ademola Fawibe

The University of North Texas, Denton, TX 76203, USA

A R T I C L E   I N F O

Article history:
Received 9 February 2011
Received in revised form 19 December 2011
Accepted 19 December 2011
Available online 10 January 2012

A B S T R A C T

There have been numerous techniques proposed in the literature that aim to improve the performance of cache memories by reducing cache conflicts. These techniques were proposed over the past decade and each proposal independently claimed to reduce conflict misses. However, because the published results used different benchmarks and different experimental setups, it is not easy to compare them. *In this paper we report a side-by-side comparison of these techniques*. We also evaluate the suitability of some of these techniques for caches with higher set associativities. In addition to evaluating techniques for their impact on cache misses and average memory access times, we also evaluate the techniques for their ability in reducing the non-uniformity of cache accesses.

The conclusion of our work is that, each application may benefit from a different technique and no single scheme works universally well for all applications. We also observe that, for the majority of applications, XORing (XOR) and Odd-multiplier indexing schemes perform reasonably well. Among programmable associativity techniques, B-cache performs better than column-associative and adaptive-caches, but column-associative caches require very minimal extensions to hardware. Uniformity of cache accesses is improved most by B-cache technique while column-associative cache also improves cache access uniformities.

Based on the observation that different techniques benefit different applications, we explored the use of multiple, programmable addressing mechanisms, each addressing scheme designed for a specific application. We include some preliminary data using multiple addressing schemes.

1. Introduction

The primary contribution of the paper is a comprehensive comparative evaluation of several addressing schemes that aim to reduce conflict misses in cache memories. The techniques studied in this paper were proposed over the past decade and each proposal independently claimed to reduce conflict misses. However, because the published results used different benchmarks and different experimental setups, it is not easy to compare them. *In this paper, we report a side-by-side comparison of these techniques*. Many of these techniques focused on direct-mapped caches. In this paper, we also explore the suitability of these techniques for caches with higher set-associativities.

We introduce new indexing schemes that are a combination of other methods. In addition, we also evaluate the uniformity of memory accesses achieved by different techniques. We use statistical measures including Kurtosis and Skewness of the distribution of accesses to different cache sets for this purpose.

☆ Reviews processed and approved for publication by Editor-in-Chief Dr. Manu Malek.
* Corresponding author.
E-mail addresses: krishna.kavi@unt.edu, kavi@cse.unt.edu (K. Kavi), iun0001@unt.edu (I. Nwachukwu), ademolafawibe@unt.edu (A. Fawibe).

0045-7906/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2011.12.008

Our reason for this study is to see if cache memories can be designed with multiple, programmable addressing mechanisms, each addressing scheme designed for a specific application. We are currently conducting experiments to evaluate multiple programmable address decoders with cache memories. In this paper, we have included some preliminary data from such studies.

The performance of processors is limited by the memory access speeds. Cache memories can hide some of the long access times to main memories. Because of the limited sizes of caches, it is important to minimize cache misses, particularly misses due to conflicts. In conventional direct mapped caches the next data item whose memory address index bits map to that line evicts data that currently resides in that cache line. Associative caches mitigate such evictions by employing several cache lines for each set. However, on a cache access to a set-associative cache the tags of all the lines in a given set are checked to see if the block exists in cache. This operation increases access times and/or energy consumed by caches.

In addition to cache conflict misses in direct mapped caches, several researchers (for example, [1–4]) have reported that not all cache sets (or lines) are accessed equally and the heavily accessed sets lead to most of the conflict misses and thus to poor performance. Consider for example Fig. 1(a), which shows the misses per set of a directly-mapped L-1 data cache for the SPEC 2006 Gromacs benchmark (x-axis corresponds to cache line number and y-axis corresponds to the number of cache misses). Spreading the cache accesses more uniformly across all cache sets may reduce the conflict misses. Fully associative caches will improve cache uniformity since a data item can be placed anywhere in the cache. Fig. 1(b) shows how the uniformity of accesses (for Gromacs) improves when a 16-way associative cache is used. Note that the range of misses (y-axis) in Fig. 1(b) is much smaller than that in Fig. 1(a). However, higher associativities require more complex hardware.

Previously we observed that the degree of non-uniformity depends on the application [2,5]. For example, MCF (a SPEC benchmark) exhibits reasonably uniform behavior. In some benchmarks, even if most accesses fall to a small number of sets, most of these accesses are hits, and thus causes no performance problems. We may need to apply different techniques depending on the severity of non-uniformity of accesses. Fortunately several solutions are available. The solutions fall into two groups: finding optimal cache-indexing schemes and dynamically remapping addresses to less utilized cache sets.

1.1. Optimal indexes

Mapping an address to a cache set relies on the use of a portion of the address. Consider an address space of 2^N bytes (i.e., N address bits), and a cache with 2^n lines of 2^b bytes (for a capacity of 2^{n+b} bytes). We will use m bits out of the N address bits to locate a set with k lines (k-way associative), where $m = \{n - \log_2(k)\}$; and use b additional bits to locate a byte, leaving $(N - m - b)$ bits as the tag. In traditional caches we use lower-end m bits for indexing, defining modulo 2^m hashing (Fig. 2).

In a more general view we can consider this process as finding a *hash function* that maps a given key (representing the specified address) to a bucket (a cache line or set) in which the data may (or may not) be found. Cache access uniformity may be improved by finding a "*perfect hash function*". The size of the bucket determines the set-associativity. Similar to the use of linked lists to resolve collisions in hashing, we can view cache associativity (or bucket size) as collision resolution, and not all buckets need to be of the same size. It should be noted that finding a perfect hash function (i.e., selecting address bits representing the hash function) is NP-complete [1]. In the Section 2, we will present several heuristics for computing cache indexes.

1.2. Dynamic relocation of addresses (or programmable associativity)

As stated in the previous section, higher associativities can lead to more uniform utilization of cache and reduce conflict misses. However, higher associativity also requires more complex hardware and may increase access times to cache. In

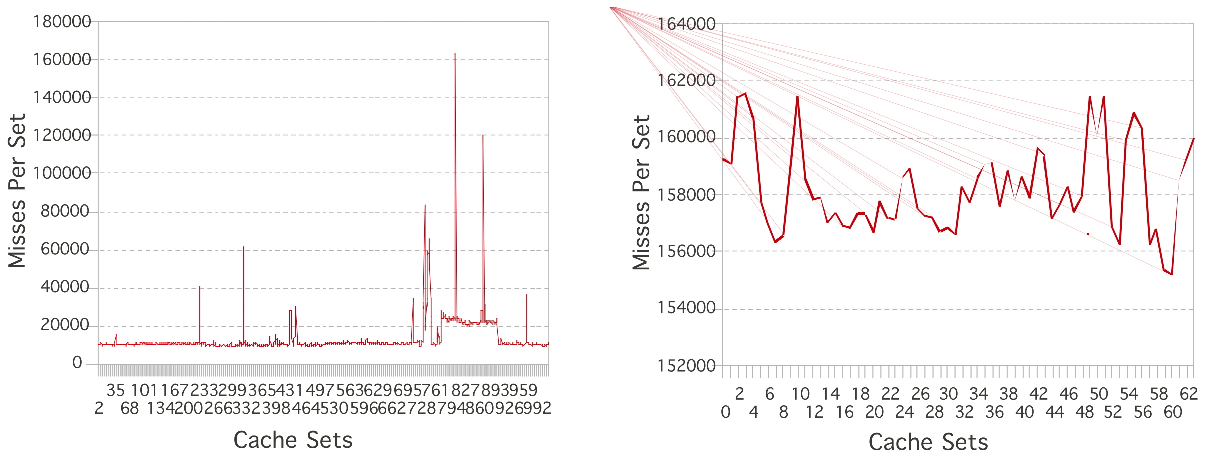

Fig. 1. Non-uniform accesses of a direct-mapped (a) and 16 way associative cache (b).

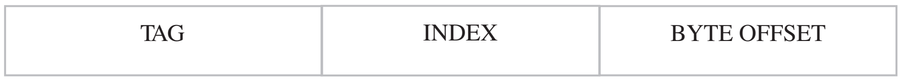

Fig. 2. Cache address mapping.

column-associative caches [6], the cache is viewed first as a direct mapped cache – by mapping an address to a specific cache line. If the desired element is not found, the cache is then viewed as 2-way associative and the second element of the set is searched – the alternate location is obtained by complementing the most significant bit of the index. This approach provides a higher associativity only when needed. Consider the following variation to column associativity (Fig. 3).

We add two fields to traditional caches: L and Partner Index (V is the traditional Valid bit). The L field indicates if the cache line is associated with another line and the Partner index identifies the alternate cache line. This is different from ‘‘column associativity’’ since any two cache-lines can form a set. We can select less frequently used cache lines as partners to more frequently used cache lines. We can either use profiling or dynamically match cache lines as partners by keeping count of accesses and/or misses with each line. In principle we can extend the ‘‘partner index’’ idea to create a linked list of cache lines, effectively increasing the set-associativity for ‘‘hot’’ or heavily accesses lines. It should be noted that the length of the linked list can be viewed as the associativity for the original index, and since the length of the linked list depends on the conflicts for a given index, we use the term ‘‘programmable associativity’’ to refer to this technique. Of course, the longer the list, the more cycles are expended in finding the desired object. While this approach offers a great deal of flexibility, the solution can be costly because of the extra bits in cache and extra cycles needed to find an address. Other implementations can achieve similar goals with restricted flexibility. We will describe some such techniques in Section 3.

The focus of the paper is on architectural solutions. However, software solutions to improve cache localities have been investigated by several researchers (for example, [7–17]). Spatial localities of data can be improved using compile time and/or runtime analysis of dynamically allocated objects and relocating them to contiguous locations in memory. For example, *pool allocators* [14] allocate linked lists in consecutive locations of memory (or pools) in order to improve spatial localities of objects. In [11,12], the authors propose a generalized approach to associate attributes that can be used by custom memory allocators. In addition to the size of an object, one can associate attributes that specify how objects should be allocated with respect to virtual pages or cache lines, or how objects should be allocated in relation to other (related) objects. However, it is unclear how a programmer would be able to understand and specify appropriate attributes; an incorrect or overly conservative specification might defeat the intended purpose of such attributes.

The rest of the paper is organized as following. Section 2 introduces various indexing schemes for addressing caches. Section 3 introduces different techniques to remap a data address to a different cache location (i.e., programmable associativities). We present the results and analyses of our experimental evaluations in Section 4. Section 5 provides our conclusions and our goals for further research.

2. Optimal cache indexing schemes

Traditional cache addressing methods use lower order bits as an index into the cache, as described in the previous section (Fig. 2). Here, we describe several techniques that suggest other ways of addressing caches.

2.1. Quality of index bits

This technique relies on address traces resulting from a program execution. From the traces the unique addresses accessed by the program are identified. Givargis [1] defines two measures with address bits of the unique addresses. The quality of a bit as an index depends on how often the bit takes the value of zero and one. High quality implies that the bit takes zero and one values equally across all the (unique) addresses accessed by the program. The correlation metric identifies the correlation between a pair of bits – there is a high correlation if the two bits either take the same value or

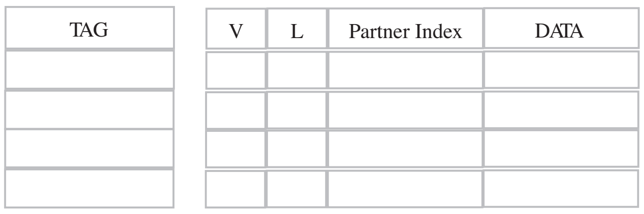

Fig. 3. Associating hot and cold sets (or programmable associativity).

complementary values in all the addresses. Bits with the largest quality values and low correlation are selected until all the m bits needed to index 2^m cache sets are identified.

The first step in the algorithm is to calculate the quality value Q_i for each address bit. The quality value is calculated as follows:

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)}$$

Here, Z_i, O_i denote the number of times bit i takes the values of zero and one, respectively, among all unique addresses in the trace. The maximum value for Q_i is 1, when the address bit assumes 0 and 1 equally.

The correlation between bits i and j is computed as follows:

$$C_{ij} = \frac{\min(E_{ij}, D_{ij})}{\max(E_{ij}, D_{ij})}$$

where E_{ij}, D_{ij} and denotes the number of times bits i and j have equal or complementary values, respectively.

A correlation matrix describes all pairwise correlations. The bit with the highest quality value is selected and the dot product between the quality value vector and the correlation for the selected bit is the new quality vector. From these value vectors, m index bits with the highest values are chosen.

It should be noted that Givargis's method does not differentiate between frequently accessed and rarely accessed addresses in computing quality of bits. This may disadvantage frequently accessed addresses causing conflicts with rarely accessed addresses.

2.2. Address conflict patterns

Patel et al. [18] exhaustively searches for the index bit combinations that results in the least number of conflict misses for a memory trace. The goal is to find index bits that results smallest conflict cost as defined below. The algorithm is centered around computing direct conflict pattern (DCP_{ij}) between two addresses A_i and A_j: DCP_{ij} is computed by performing a bit-wise comparison of the two addresses. Then the total conflict cost CP_i for address A_i is computed by bit-wise ORing DCP_{ij} values for every address A_j that follows A_i in the trace, before A_i is accessed again. The conflict cost is then computed for the length of the trace L. The goal is to identify address bits that cause the least number of conflicts for the given address trace. It should be noted that, unlike Givargis's method, Patel's technique accounts for the frequency of accesses of addresses in the trace. As can be seen from the description above, this method is computationally prohibitive. We will not include this method in our experimental evaluations.

2.3. Prime-modulo hashing function

Unlike previous techniques that rely on address traces, the next several techniques define different addressing functions [19] without regard to specific address traces. In Prime-modulo hashing, the set to which a given address maps is computed using modulo with respect to a prime number.

$$H(a_i) = a_i \bmod p$$

The prime number p is selected such that it is less than or equal to the number of cache sets. The difference between prime modulo indexing and traditional hashing is that we use a prime number instead of the total number of sets in the cache. The hardware to compute prime modulo function can be complex and the time to complete the operation can be long. And, cache fragmentation occurs because not all sets in the cache will be utilized. The table below (Table 1) shows the fragmentation (or wasted cache resource) for different cache sizes.

With larger caches the fragmentation is not significant and thus this technique may be more appropriate for L-2 or Last Level caches.

Table 1
Prime modulo fragmentation.

Cache sets	Cache sets used by prime modulo	Fragmentation (%)
256	251	1.95
512	509	0.59
1024	1021	0.29
2048	2039	0.44
4096	4093	0.07
8192	8191	0.01
16,384	16,381	0.02

2.4. Odd-multiplier displacement indexing

In odd-multiplier displacement hashing [19] the traditional modulo is performed after an offset is added to the original index x_i. The offset is the product of a multiplier $p$and the tag T_i.

$$H(a_i) = (p * T_i + x_i)\mathrm{mod}n$$

Here, n is the total number of sets in cache. This function is based on hashing functions in Aho and Ullman [20] and is related to Raghavan and Hayes's RANDOM-H functions [21]. The choice of the odd multiplier determines the performance of this technique. Some multipliers that work well are 9, 21, 31, and 61.

2.5. Exclusive-OR (XOR) hashing

In this technique set index bits (x_i) are exclusive-ored with selected bits chosen form the tag part (t_i) of the address [19].

$$H(a_i) = t_i \otimes x_i$$

XOR operation reduces conflicts as follows. When the set index bits are the same for two different addresses, at least one of the tag bits will be different for the addresses. When tag bits are XORed with index bits, the conflicting addresses will be mapped to different cache lines. However, this may cause conflicts with other addresses.

2.6. Givargis-XOR

We propose a hybrid of Givargis optimal bits selection algorithm and the XOR hashing scheme. It works as follows, instead of XORing the index bits with tag bits immediately preceding the index bits, an optimal set of tag bits as defined by Givargis method [1] is XORed with the index bits.

In this paper, we compare techniques described here except Patel's [18] technique since it is computationally intractable for large address traces.

3. Programmable associativity

A fully associative cache with perfect replacement policy will access all cache lines equally likely, because data can be placed anywhere in the cache. However, fully associative caches with perfect replacement policies are not realistic. Here, we describe some techniques that increase the effective associativities for cache lines that incur higher misses, without increasing the associativity of the entire cache. It should be noted, however, most of these techniques incur additional cycles to locate the item in a secondary location when a miss occurs in the primary location.

3.1. Column associative cache

In column-associative (or pseudo-associative) caches [6], the cache is fist viewed as a direct mapped cache – by mapping an address to a specific cache line. If the desired element is not found, the cache is then viewed as 2-way associative and the second element of the set is searched. The alternate location is obtained by "flipping" (or complementing) the most significant bit of the index. When there is a miss in both locations, the data residing in the original index location is moved to the alternate location and the rehash bit of the alternate set is set to 1. When a direct miss occurs in a set whose rehash bit is set to one, new data is written into that set and the rehash bit is reset to zero, indicating that it is indexed conventionally.

3.2. Adaptive caches

In Adaptive-cache [3], conflicting data items are relocated to new cache lines by using two tables. SHT (Set-reference History Table) keeps only the set indexes corresponding to Most Recently Used (MRU) sets and OUT (Out-of-position directory) maintain indexes for items evicted from MRU sets. When an access to the cache occurs the OUT directory is accessed in parallel with the cache. If the data is in the cache, the set history table SHT is updated for MRU status. If the data is not present in the cache, the OUT directory is accessed to see if an entry in OUT matches the tag of the address referenced; then the OUT table provides the cache index of the alternate location that contains the address referenced. The data may be swapped between the primary cache location and the alternate location stored in OUT to improve future access latencies. The OUT directory is updated to reflect the new set holding data corresponding to the tag. To simplify cache management a disposable or d bit is maintained for each cache block to indicate whether a block should be evicted or kept in an alternate location. The OUT table is not consulted when the disposable bit is set [3].

On a miss, the data residing in a block is simply replaced if the disposable bit is set. However, if the disposable bit is reset, then an alternate block has to be identified to hold the data that would otherwise be evicted from the cache. If the OUT directory has empty slots then an empty line is used to hold the data. The OUT directory is then updated with this new entry. On the other hand if the OUT directory is full then the least-recently used slot in the OUT directory is used to hold the displaced address. The SHT is updated on every access to maintain the table of MRU status of sets. The performance of this technique depends on the number of entries in SHT and OUT tables. Based on experimentation good sizes for the SHT and OUT are 3/8 and 4/16 of the total number of lines in the cache.

3.3. B-cache

Zhang's B-cache [4] reduces accesses to frequently missed sets and increases the accesses to less active sets. We will illustrate the functionality of a B-cache with a simple example. Let us assume that the following addresses reference a B-cache: 1, 4, 7, 1, 3, and 9. To keep the explanation simple let us ignore the byte-offset, and we will use 4-bit addresses that can be divided into cache index and tag. We will use a 4-line direct-mapped cache for our example. In a traditional directly mapped cache we need 2 bits as cache index, leaving 2 bits as tag. The B-cache changes the mapping by using only the least significant bits as cache index (referred to as non-programmable index or NPI) and uses the 2 bits that follow as programmable cache index bits (called PI). This can be viewed as effectively partitioning cache into two halves; we refer the first half (00 & 10) as group 1 and refer the second half (01 & 11) as group 2. Under such a mapping, the first access to address 1 ("0001") will map to two possible cache lines and placed in one of two cache lines, say 01. The second reference 4 ("0100") is placed in 00, while the third reference, 7 ("0111") is placed cache line 11. The next reference to address 1 will result in a hit. Assuming we will use LRU for replacement, the recency status of set 00 is updated. When address 3 ("0011") is referenced, the line 11 (as compared to line 01) is selected because of the recency information. On every access the corresponding programmable index is set (in this case the PI is 0). When address 9 ("1001") is referenced, there is a miss in the cache. However using the programmable index bit, we will select line 01 to store the address 9. If there is a miss in the cache but a hit in the PI, the block with the PI match is the candidate for replacement. On a miss in both the PI and the cache, a line is chosen using a replacement policy and the programmable index is reprogrammed with the PI of the new block.

B-cache effectively increases the set-associativity using programmable and non-programmable indexes. In fact, we can view the example presented here as equivalent to a 2-way set associative cache. However, using two-programmable index bits, B-cache should be viewed as a partially decoded 4-way associative cache. The length of the programmable and non-programmable index is determined by the mapping factor (*MF*) and B-cache associativity (*BAS*).

$$MF = \frac{2^{PI+NPI}}{2^{OI}}$$

Here, *OI* is the number of index bits in the direct-mapped cache, *PI* and *NPI* are the number of programmable and non-programmable bits. The B-cache associativity (*BAS*) determines how the cache is partitioned into clusters. In the example above the 4-line cache is divided into two clusters each with two lines, thus BAS is 2. The miss-rate of a BAS cluster cache approaches that of a BAS-way associative cache. The BAS value is computed as follows:

$$BAS = \frac{2^{OI}}{2^{NPI}}$$

In this paper, we compare the three techniques presented in this section. We will also explore the use of different indexing schemes (as described in Section 2) with B-cache programmable indexes.

4. Experimental methodology and results

Several SPEC[1] 2006 and MiBench[2] benchmarks are run using the SimpleScalar toolset [22]. SimpleScalar is a cycle accurate processor simulator that supports out-of-order issue and execution. All the benchmarks used are pre-compiled for the Alpha instruction-set-architecture. We modified the cache memories in SimpleScalar to implement the various techniques described in this paper. We collected statistics on hits, misses and total accesses per set.

Our simulations are based on 32 KB direct-mapped L1 data and instruction caches with 32 byte blocks, and a 256 KB unified L2 cache with 32 byte blocks. The sizes of OUT and SHT tables for the Adaptive-cache are 3/8 and 4/16 of the cache size respectively. We use LRU replacement policy in our B-cache implementation.

4.1. Comparing indexing techniques

We explored how the different indexing schemes described in Section 2 perform in terms of cache accesses. We did not evaluate Patel's indexing scheme because of the intractability of the computations needed to find optimal indexes. In this section we show the results of using Givargis, Odd multiplier, Prime modulo and XOR schemes for SPEC 2006 and Mibench benchmarks.

[1] http://www.spec.org.

[2] www.eecs.umich.edu/mibench/.

In Figs. 4 (SPEC) and 5 (MiBench), we show the percentage reduction in cache misses achieved using these techniques, when compared to traditional (modulo) indexing scheme. A negative value indicates an increase in cache miss rates when compared to traditional indexing scheme.

4.1.1. Remarks

These figures show that none of the techniques perform consistently well. On average, Givargis technique performs worst among the techniques studies in this paper, while XOR, Prime-modulo and odd-multiplier techniques perform reasonably well. XOR requires very minimal hardware (XOR gates) while odd-multiplier requires an integer multiplier unit. Prime-modulo is not practical since it requires hardware to compute a prime modulo. Givargis's [1] method needs no additional hardware but the off-line analysis can be expensive. The reason for the poor performance of Givargis' method is that it does not differentiate between frequently accessed and rarely accessed addresses. Moreover, when applications' memory access differ from run to run, it will be necessary to apply this off-line analysis for each run. Previous studies [1] indicated that Givargis's technique produces improvements when used with smaller line sizes.

4.1.2. Impact of indexing schemes with higher associativities

We then explored the impact of the various indexing schemes when higher associativities are used. We explored L-1 caches with 2-, 4-, 8- and 16-way associativities. The results are shown in Table 2.

Once again, Givargis' technique performs poorly. The cache miss rate improvements for other techniques are minimal and as expected the improvements are diminished with higher associativities.

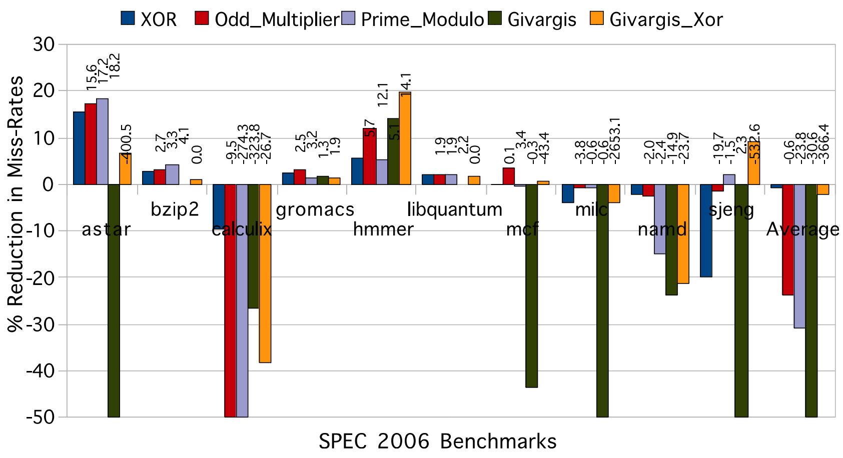

Fig. 4. Percent reduction in miss-rates for SPEC 2006 benchmarks.

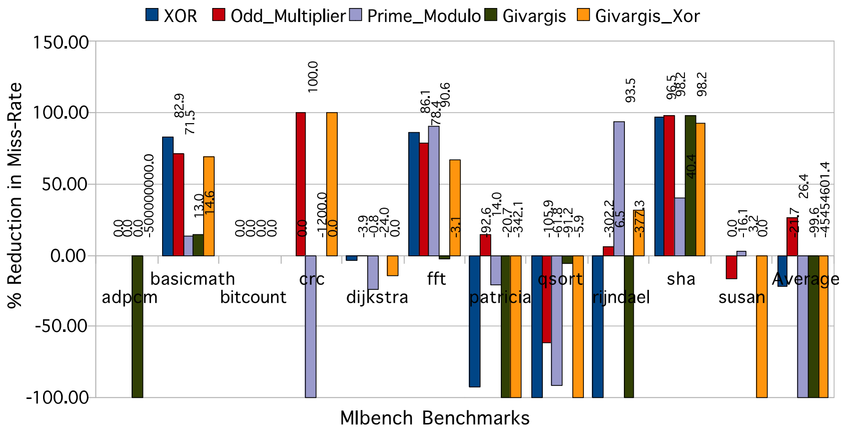

Fig. 5. Percent reduction in miss-rates for Mibench benchmarks.

Table 2
Percentage cache miss-rate reduction for a 2, 4, 8, and 16 way associative cache with different cache indexing schemes.

Benchmark	XOR	Odd_Multiplier	Prime_Modulo	Givargis
Two way				
astar	1.54	0.38	0.38	−216.92
bzip2	2.89	3.31	3.31	−21.07
calculix	−6.67	0.00	−11.11	−122.22
gromacs	0.93	0.00	0.93	−18.69
hmmer	2.30	16.59	5.07	−23.96
libquantum	0.00	0.00	0.26	0.00
mcf	0.06	−0.50	−0.44	−17.91
milc	0.00	0.00	0.00	−1575.86
namd	12.12	7.58	6.06	−266.67
sjeng	−28.00	16.00	16.00	−324.00
Average	−1.48	4.34	2.05	−258.73
Four way				
astar	0.87	0.87	0.44	−58.08
bzip2	4.19	4.19	4.19	−48.17
calculix	0.00	2.70	0.00	−97.30
gromacs	0.00	0.00	0.00	−51.90
hmmer	0.71	1.43	1.43	−54.29
libquantum	0.00	0.00	0.26	0.00
mcf	−0.36	−0.57	−0.64	−28.35
milc	0.00	0.00	0.00	−665.52
namd	0.00	0.00	0.00	−1016.67
sjeng	0.00	0.00	0.00	−530.00
Average	0.54	0.86	0.57	−255.03
Eight way				
astar	0.00	0.00	0.00	−37.61
bzip2	1.37	1.37	1.37	−91.10
calculix	0.00	0.00	0.00	−61.76
gromacs	0.00	2.44	2.44	−187.80
hmmer	0.00	0.00	0.82	−59.84
libquantum	0.00	0.00	0.26	0.00
mcf	−0.61	−0.61	−0.53	−33.64
milc	0.00	0.00	0.00	−320.69
namd	0.00	0.00	0.00	−977.78
sjeng	0.00	0.00	0.00	−375.00
Average	0.08	0.32	0.44	−214.52
Sixteen way				
astar	0.00	0.00	0.46	−34.26
bzip2	1.00	1.00	1.00	−173.00
calculix	0.00	0.00	3.03	−54.55
gromacs	0.00	0.00	0.00	−306.90
hmmer	0.00	0.00	0.00	−65.22
libquantum	0.00	0.00	0.26	0.00
mcf	−0.95	−0.95	−0.87	−39.42
milc	0.00	0.00	0.00	−48.28
namd	0.00	0.00	0.00	−950.00
sjeng	0.00	0.00	0.00	−212.50
Average	0.01	0.01	0.39	−188.41

4.2. Comparing column associative, adaptive and B-caches

Figs. 6 and 7 show the percentage reduction in miss-rates achieved by these techniques when compared to traditional direct mapped caches, for SPEC 2006 and Mibench benchmarks. Miss-rates alone, however, do not fully demonstrate the performance improvement of these schemes. A better metric to use is the average memory access time (AMAT), which accounts for the extra cycles incurred in the implementation of some of these techniques. We estimated that the Adaptive-cache incurs three extra cycles if there is a hit in the OUT directory. This is because of the cycles used to search the OUT directory and for the second cache lookup of that entry. Consequently, the hit-time is split into two fractions, one for direct hits to the cache and the other for hits in the OUT directory. In case of column-associative cache, we added one extra cycle when data is not found in the primary location but found in the secondary location. Figs. 8 and 9 show the reduction in average memory access times (AMAT) achieved by these schemes.

4.2.1. Remarks

In general all three methods reduce cache misses. When using average memory access times (AMAT) for comparison, B-cache outperforms the other methods, because this method does not incur additional cycles to locate an address unlike

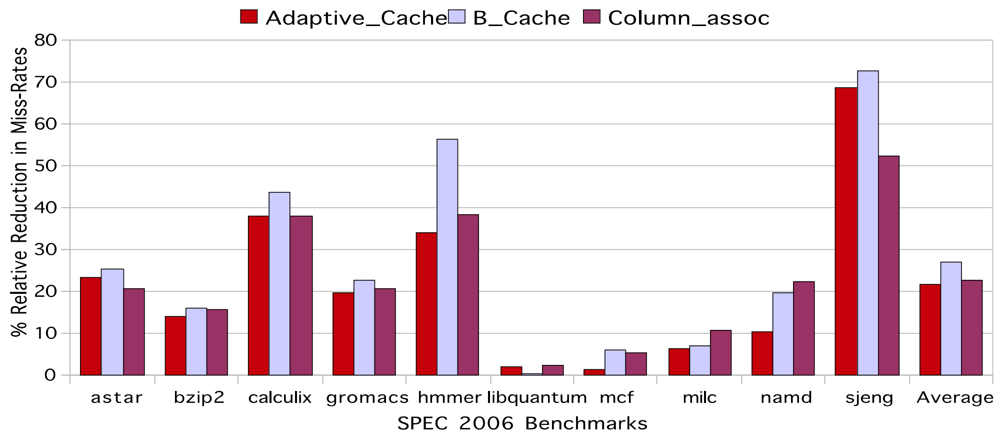

Fig. 6. Percent reduction in miss-rates for SPEC 2006 benchmarks.

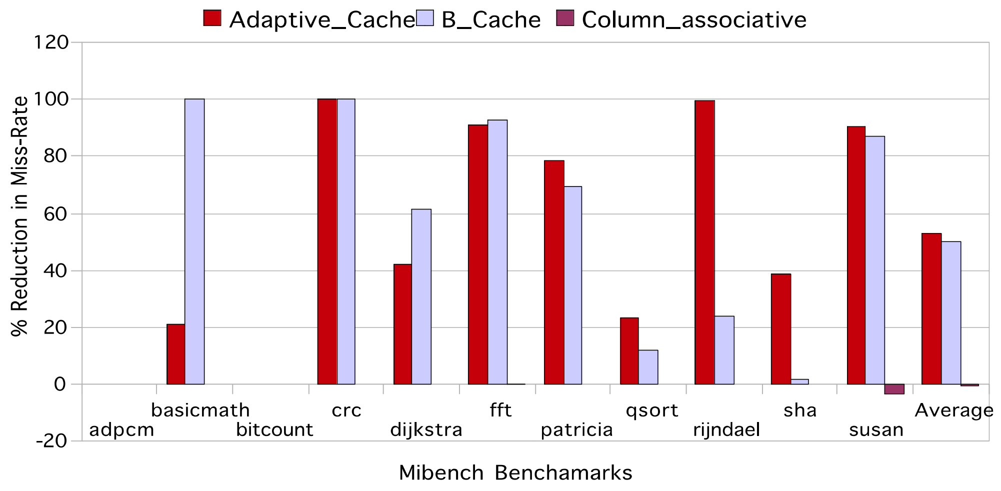

Fig. 7. Percent reduction in miss-rates for Mibench benchmarks.

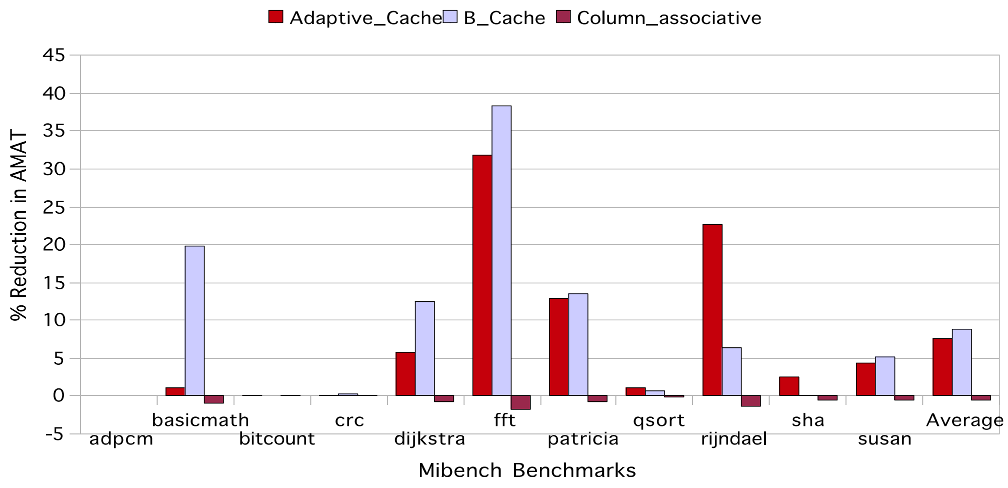

Fig. 8. Percent reduction in average memory access times for Mibench benchmarks.

adaptive-cache and column-associative techniques. On the other had, column-associative method is the simplest technique to implement. B-cache relies on a specific implementation of caches (viz., designed with multiple banks of SRAM).

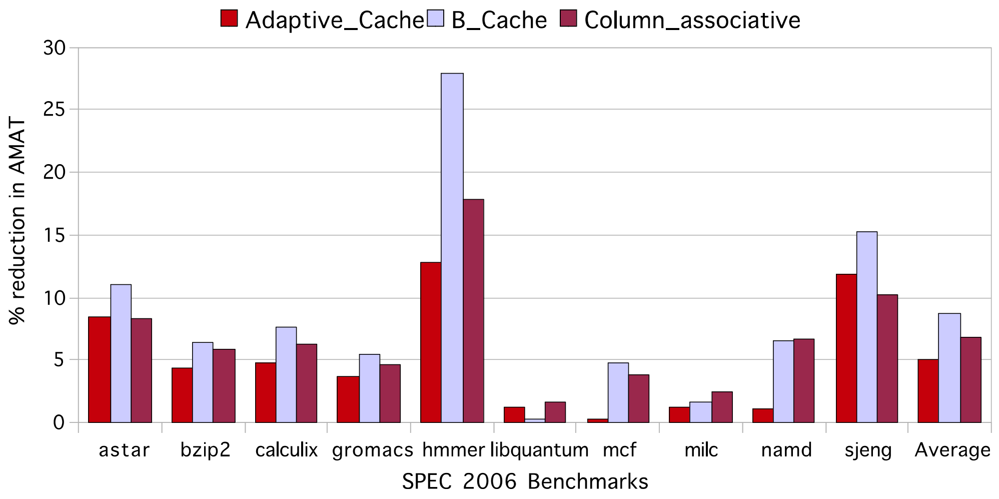

Fig. 9. Percent reduction in average memory access times for SPEC 2006 benchmarks.

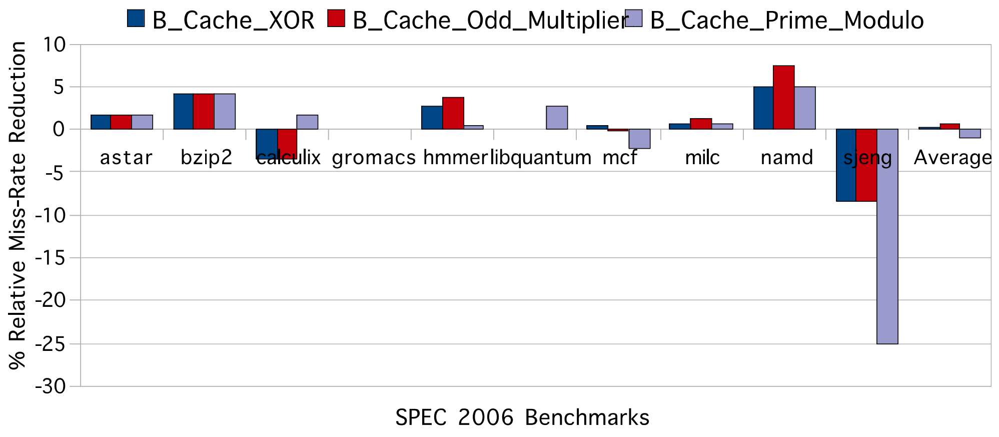

Fig. 10. Miss-rate reductions over conventional B-caches.

Adaptive-cache requires additional tables (OUT and SHT) and incurs additional cycles to locate entries in these tables. However, adaptive-cache technique may be useful in managing non-uniform caches (NUCA) and cooperative shared caches (see Section 4.4).

4.2.2. Different indexing schemes with B-cache

We wanted to explore the possibility of changing indexing schemes with B-cache. More specifically, we compared B-cache using traditional indexing for programmable indexes against B-cache using XOR, Prime-modulo and odd-multiplier techniques for selecting programmable index bits. Fig. 10 shows the percentage reduction (or increase) in misses resulting from the various indexing schemes when *compared B-cache using traditional indexing* for SPEC 2006 benchmarks. In most cases the new indexing methods show some reductions in cache misses. B-cache using XOR and odd-multiplier techniques show larger improvements than Prime-modulo technique. For some benchmarks, however, new indexing schemes actually increase cache misses.

4.3. Evaluation of uniformity of memory accesses

While results so far show some improvements in terms of miss rates, we also wanted to explore the uniformity of cache accesses achieved by various techniques. Zhang [4] measured uniformity by computing the percentage of sets that are “Frequently Hit Set (FHS)”, “Frequently Missed Set (FMS)”, and “Least-Accessed Set (LAS)”. A set is FHS if it received at least two times the average number of hits; a set is FMS if it received at least twice the average number of misses and a set is LAS if it received less than half the average number of hits. In order to more formally describe the behavior of cache access patterns, it is necessary to convert the accesses and misses into probability distributions. We can then measure various statistical

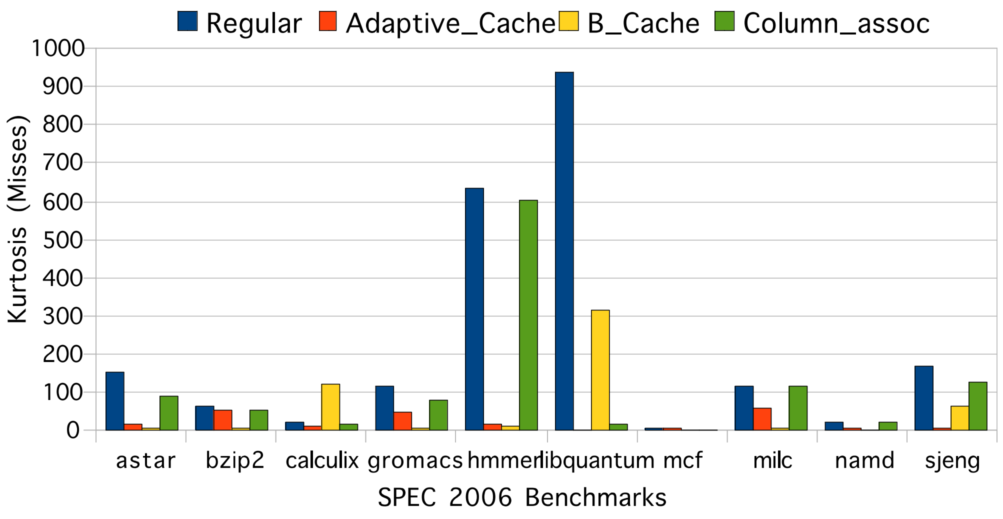

Fig. 11. Kurtosis values for the misses per set for SPEC 2006 benchmarks.

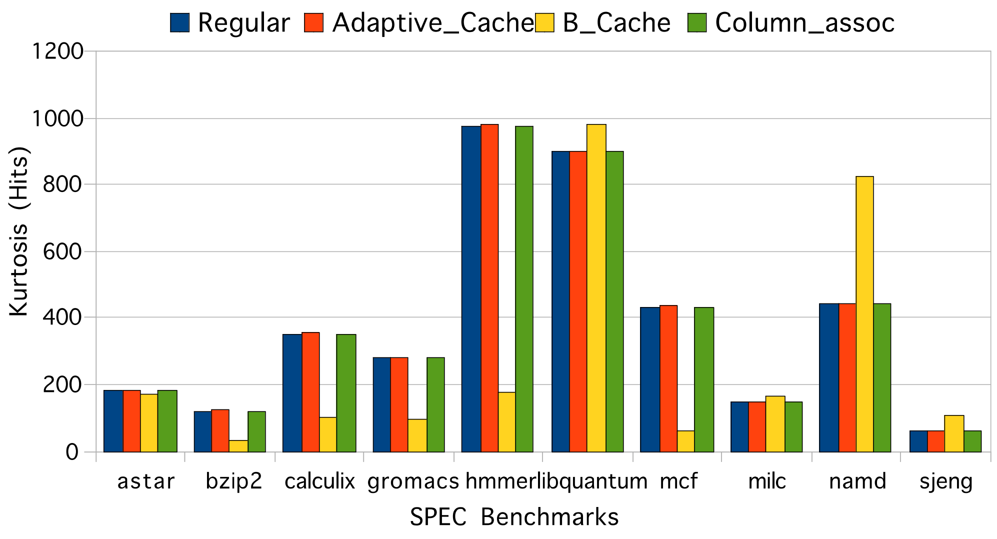

Fig. 12. Kurtosis values for hits for SPEC 2006 benchmarks.

values knows as central moments. Most commonly used moments are: mean (first moment) and standard deviation (second moment). Higher moments describe the shape of the distribution. We use two such moments in this paper.

4.3.1. Skewness and Kurtosis

Skewness (third central moment) is a measure of symmetry, or more precisely, the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the center point (mean). If the left tail is more pronounced than the right tail, the function is said to have negative skewness. If the reverse is true, it has positive skewness.

Kurtosis (fourth central moment) is a measure of whether the data are peaked or flat relative to a normal distribution. That is, data sets with high Kurtosis tend to have distinct peaks and have long tails. This also indicates very few values near the peaks. Data sets with low Kurtosis tend to have a flat top near the mean rather than sharp peaks. A uniform distribution would be the extreme case with zero Kurtosis. For our purpose, a highly non-uniform behavior results in a high Kurtosis, while a more uniform access behavior leads to lower Kurtosis.

In order to better assess the uniformity achieved across the sets we computed the Kurtosis and Skewness of hits and misses to each of the 1024 L1 cache lines for Adaptive, B- and Column-associative caches. The results are shown in Figs. 11–14. We did not include similar data for the various indexing schemes, since those techniques do not significantly change the uniformity of accesses (and this is apparent from the miss reduction data shown in Figs. 4 and 5).

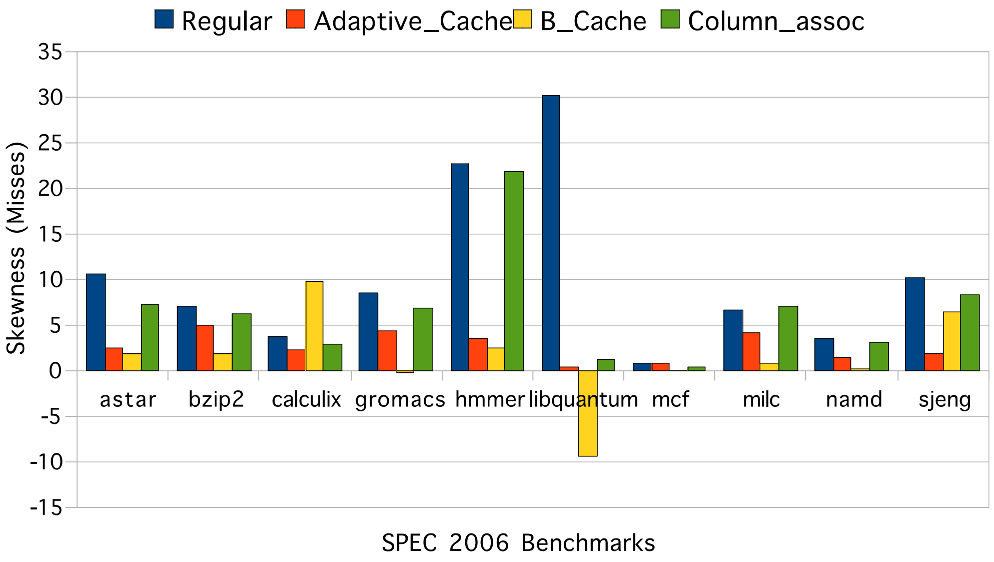

Fig. 13. Skewness values for the misses per set for SPEC 2006 benchmarks.

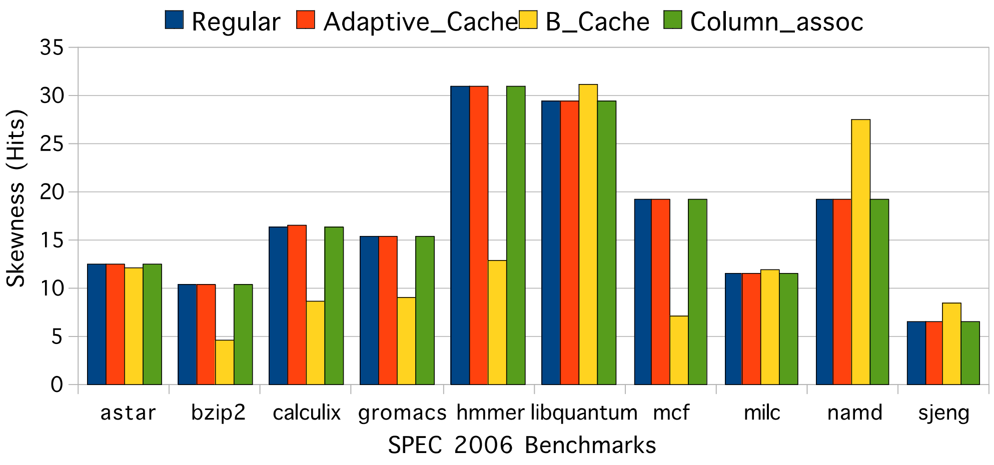

Fig. 14. Skewness values for the hits per set for SPEC 2006 benchmarks.

4.3.2. Remarks

As can be seen from the Figs. 11 and 13, the Kurtosis and Skewness values for cache misses show a reduction (or improved uniformity of accesses). In particular, the Adaptive-cache shows significant reductions in Kurtosis (Fig. 11) and Skewness (Fig. 13). The goal of the techniques evaluated in this paper is to reduce conflict misses by distributing accesses more uniformly across the cache. At the same time, hits can remain non-uniform because they do not cause performance penalties. This is evident from the Kurtosis (Fig. 12) and Skewness (Fig. 14) for hits. The negative skewness indicates that for that benchmark, misses have been relocated to lower numbered cache sets.

4.4. Multiple indexing schemes

Our reason for this study is to see if cache memories can be designed with multiple, programmable addressing mechanisms, each addressing scheme designed for a specific application. Here, we provide some preliminary results of using different indexing schemes for multiple threads. We used MSim[3] to simulate SMT like multithreaded system. While XOR technique performs better than other indexing techniques for single threaded applications (data presented thus far), this is

[3] www.cs.binghamton.edu/~msim/.

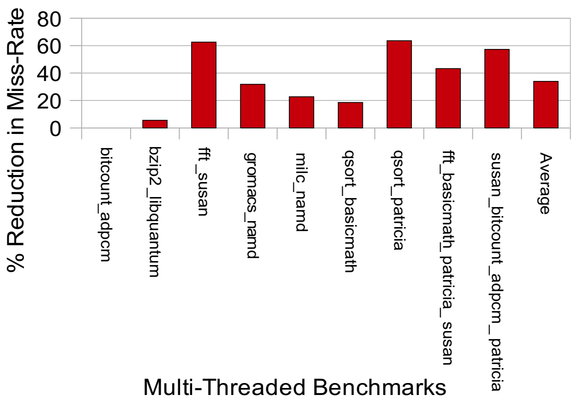

Fig. 15. Multiple address decoders for multithreaded systems.

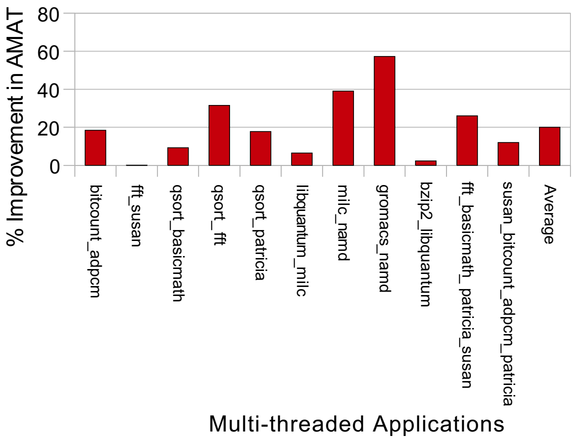

Fig. 16. Percentage reduction in AMAT using Adaptive caches in multithreaded systems.

not the case for multi-threaded systems, because the low order address bits of the data accessed by the threads will not differ much. This implies that the index bits and some of the tag bits of addresses will be the same for multiple threads. For this reason we used Odd-multiplier indexing with different multipliers for each thread.

Fig. 15 shows the percentage reduction in cache miss rates (when compared to using a single addressing technique based on conventional modulo indexing). Data points with 2 names represents the case when 2 threads are running and data points with 4 names reflects the case when 4 threads are running concurrently. As can be seen, using different address decoding for each application can significantly improve the performance of shared caches. We are currently extending the experimentation for multicore systems running multiprogramming and high-performance applications.

We are also exploring how Pier's adaptive cache [3] can benefit multithreaded and multicore system. Since the low order bits of all applications in multicore systems will be similar, B-cache does not perform any better than Adaptive cache. In one preliminary experiment, we divided the shared cache equally among the threads (either 2 or 4). We use Pier's Adaptive caches with SHT and OUT tables so that lightly used sets of one thread can be "donated" to other threads – to hold frequently conflicting items, effectively increasing the size of the cache partition for that thread. This can also be viewed as providing victim cache for threads from underutilized cache portions of other threads. Fig. 16 shows our preliminary data.

The figure shows the percentage reduction in average memory access times (AMAT) using adaptive-caches (with partitioned cache), when compared to traditional indexing (and no cache partitioning). The data shows that partitioned caches using adaptive-cache method can substantially improve the performance of multithreaded systems with shared caches. It is worthy to note that average memory access times (AMAT) includes the additional access delays incurred by adaptive caches in locating data items not present in their primary location.

5. Conclusion

In this paper, we conducted a side-by-side comparison of several techniques aimed at improving performance of direct-mapped or low-associative caches. These techniques aim to reduce conflict misses by spreading cache accesses more uniformly across cache sets. We reported the reduction (or increase) of cache miss rates and reduction (or increase) of average memory access times. To compare the uniformly achieved by the techniques we used Kurtosis and Skewness of accesses; lower values for Kurtosis imply more uniform accesses across cache sets.

As can be seen from the data presented in this paper, none of the techniques improve cache perform consistently for all benchmarks. Some techniques perform better than others for some benchmarks. On average, however, among the different indexing schemes, XOR, odd-multiplier and Prime-modulo techniques perform reasonably well. XOR and odd-multiplier techniques require very minimal addition to hardware. A variation to the XOR method can be to exclusive-or process or thread-ID with index bits with shared caches (such as L2 or L3). This is likely to spread memory access of different threads and minimize conflicts.

Among the programmable associative techniques, B-Cache consistently performs better, both in terms of miss-rates and average memory access times. Column-associative requires very minimal hardware extensions. B-Cache improves the uniformity of memory accesses better than any other technique studies in this paper. Adaptive-caches may be useful in addressing NUCA (non-uniform caches).

Although not included in this paper, we explored the impact of cache block size on the effectiveness of the techniques studied in this paper. In general, larger cache blocks (64 or 128 bytes) make these techniques less effective. This is particularly true for Givargis and B-cache techniques.

Our reason for this study is to see if cache memories can be designed with multiple, programmable addressing mechanisms, each addressing scheme designed for a specific application. We are currently conducting experiments to evaluate multiple programmable address decoders with cache memories. We are exploring the use of Adaptive-caches for building cooperative shared caches in multicore systems. The idea is to “donate” less frequently used sets from one core to other cores, dynamically increasing available cache capacity at each core. In this paper we have included some preliminary data from such studies.

Acknowledgements

This research is supported in part by the NSF Net-Centric Industry/University Research Center (Net-Centric IUCRC) and a gift from AMD.

References

[1] Givargis T. Improved indexing for cache miss reduction in embedded systems. In: IEEE/ACM design automation conference (DAC). Anaheim; 2003. p. 872–80.

[2] Naz A, Adamo O, Kavi K, Janjusic T. Improving uniformity of cache access patterns using split data caches. In: Proceedings of ISCA conference on parallel and distributed computer systems (PDCS-2009). KY: Louisville; 2009.

[3] Peir J, Lee Y, Hsu W. Capturing dynamic memory reference behavior with adaptive cache topology. In: Proceedings of the 8th international conference on architectural support for programming language and operating systems (TOPLAS); 1998. p. 240–50.

[4] Zhang C. Balanced cache: reducing conflict misses of direct-mapped caches. In: ACM international symposium on computer architecture (ISCA-06); 2006. p. 155–66.

[5] Adamo O, Naz A, Kavi K, Janjusic T, Chung CP. Smaller split L-1 data caches for multi-core processing systems. Proceedings of IEEE 10th international symposium on pervasive systems, algorithms and networks (I-SPAN 2009). Taiwan: Kao-Hsiung; 2009. p. 14–6.

[6] Agarwal A, Pudar SD. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In: Proceedings of the international symposium on computer architecture (ISCA-93); 1993. p. 179–80.

[7] Berger E, Zorn B, McKinley K. Reconsidering custom memory allocation. Proceedings of the 17th ACM conference on object-oriented programming systems, languages and applications (OOPSLA ’02); 2002. p. 1–12.

[8] Calder B, Krintz C, John S, Austin T. Cache-conscious data placement. In: Proceedings of 8thACM international conference on architectural support for programming language and operating systems (ASPLOS); 1998.

[9] Carr S, McKinley KS, Tseng C-W. Compiler optimizations for improving data locality. In: Proceedings of ACM international conference on architectural support for programming language and operating systems (ASPLOS); 1994. p. 252–62.

[10] Chilimbi TM, Larus JR, Hill MD. Tools for cache conscious data structures. In: Proceedings of ACM conference on programming languages, design and implementation (PLDI); 1999.

[11] Jula A, Rauchwerger L. How to focus on memory allocation strategies. Tech Rept TR 07-003, Department of Computer Science, Texas A&M University.

[12] Jula A, Rauchwerger L. Custom memory allocation for free. In: Proceedings of the 19th international conference on Languages and compilers for parallel computing (LCPC ’06). New Orleans: Springer-Verlag; 2007. p. 299-313.

[13] Kulkarni C, Ghez C, Miranda M, Catthoor F, Man HD. Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications. In: IEEE transactions on computers, vol. 54(1); 2005.

[14] Lattner C, Adve V. Automatic pool allocation: improving performance by controlling data structure layout in the heap. Proceedings of ACM conference on programming languages design and implementation (PLDI); 2005. p. 129–42.

[15] Luk CK, Mowry T. Compiler based prefetching for recursive data structures. In: Proceedings of the 7th international conference on architectural support for programming languages and operating systems (ASPLOS VII); 1996. p. 222–33.

[16] Seidl ML, Zorn BG. Segregating heap objects by reference behavior and lifetime. Proceedings of the eight international conference on architectural support for programming languages and operating systems (ASPLOS VIII); 1998. p. 12–23.

[17] Wolf, ME, Lam MS. A data locality-optimizing algorithm. In: Proceedings of ACM programming language, design and implementation (PLDI’91); 1991. p. 30–44.

[18] Patel K, Macii E, Benini L, Poncino M. Reducing cache misses by application-specific re-configurable indexing. In: Proceedings of the 2004 IEEE/ACM international conference on computer-aided design (ICCAD ’04); 2004. p. 125–30.

[19] Kharbutli M, Irwin K, Solihin Y, Lee J. Using prime numbers for cache indexing to eliminate conflict misses. Proceedings of the international symposium on high performance computer architecture (HPCA); 2004.

[20] Aho AV, Ullman JD. Principles of compiler design. Addison-Wesley; 1997.

[21] Raghavan R, Hayes J. On randomly interleaved memories. In: Proceedings of the international conference on Supercomputing; 1990.

[22] Burger D, Austin TM. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department of Technical Report #1342; 1997.

Krishna Kavi is currently a professor and the Director of the NSF Net-Centric Industry/University Cooperative Research Center at the University of North Texas. He also served as the Chair of CSE department at UNT. His research interests are in computer systems architecture, memory systems and software engineering. He published more than 150 technical papers in these areas.

Izuchukwu Nwachukwu recieved his BS in Computer Engineering from the University of Arizona, and a MS from the University of North Texas, in 2008 and 2011, respectively. His research in computer architecture and cache memories. Currently he is working on the use of core specific indexing methods to reducing miss-rates in multi-core systems.

Ademola Fawibe is a MS student in the Department of Computer Science and Engineering at the University of North Texas. He completed his BS in Computer Science at the University of North Texas. His research interests fall in the computer architecture discipline, include cache architectures and transactional memories.