Evaluation of Techniques to Improve Cache Access Uniformities

Izuchukwu Nwachukwu, Krishna Kavi, Fawibe Ademola and Chris Yan
University of North Texas
Denton, Texas, USA
{iun0001, krishna.kavi, ademolafawibe, chrisyan}@unt.edu

Abstract — **While higher associativities are common at L-2 or Last-Level cache hierarchies, direct-mapped and low associative caches are still used at L-1 level. Lower associativities result in higher miss rates, but have fast access times on hits. Another issue that inhibits cache performance is the non-uniformity of accesses exhibited by most applications: some sets are underutilized while others receive the majority of accesses. Higher associative caches mitigate access non-uniformities, but do not eliminate them. This implies that increasing the size of caches or associativities may not lead to proportionally improved cache hit rates.**

Several solutions have been proposed in the literature over the past decade to address the non-uniformity of accesses; and each proposal independently claims improvements. However, because the published results use 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. 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. Our research is investigating the use of multiple techniques within a processor core and across cores in multicore system to improve the performance of cache memory hierarchies. The study reported in this paper allows us to select best possible solutions for each running application. In this paper, we have included some preliminary results of using multiple solutions simultaneously when running multiple threads.

Keywords-Cache Memories; Cache Indexing; Non-Uniformity of Cache Accesses; Performance Improvement;

I. INTRODUCTION

In this paper we report a comprehensive study of techniques to improve uniformity of cache accesses and minimize cache conflicts. The approaches can be classified into two groups.

- Architectural modifications to improve cache uniformity, which include Adaptive cache, B-cache, and Column associative cache. We call this group as Programmable Associativities.
- Cache indexing functions to uniformly distribute accesses across cache sets, which include XOR, odd-multiplier, prime-modulo and a techniques proposed by Givargis [6].

The purpose of this paper is to evaluate schemes that fall in one of these two categories. While several of these techniques have been reported in the literature over the past decade, the published results use different benchmarks and different experimental setups and it is not easy to compare them. In this paper we report a side-by-side comparison of these techniques.

In conventional direct mapped caches the next data item whose memory address index bits map to that line evicts data stored in cache line. Set associative caches mitigate such evictions by employing several cache lines per each set. However set-associative caches incur higher access latencies when compared direct-mapped caches.

Several researchers (for example, [6], [8], [11], [13]) have reported that accesses to cache memories are non-uniform: not all cache sets are equally accessed and the heavily accessed sets lead to most of the conflict misses and thus to poor performance. Consider for example Figure 1, which shows the accesses to the L-1 data cache for the FFT Mibench program (X-axis corresponds to cache line number and Y-axis corresponds to the number of cache accesses). The graph shows that a small number of sets are heavily accessed while a majority of sets are under-utilized. About 90.43% of the cache sets get less than half of the average accesses while 6.641% get twice the average accesses. Spreading the cache accesses more uniformly across all cache sets may reduce the conflict misses. Higher associativities mitigate the non-uniformity of accesses, but do eliminate them. The non-uniformity is even more detrimental to shared caches in multicore and multi-threaded systems.

The work by [16] reduces conflict misses in direct and set-associative caches by adjusting the placement of a program procedures. They propose 2 approaches: the intermediate block profile (IBP) algorithm, which is dependent on the processors' cache configuration and the neutral procedure placement, which is independent of the cache specifications. The procedure placement algorithms aim to reduce conflict misses caused by procedure switching. More specifically, the algorithm iterates through all the hot procedures and selects the displacement value that yields the highest benefit. Liang [16] also proposed a cache locking mechanism to improve instruction cache performance. The temporal reuse distance mechanism employed assesses the benefit of locking each block address. In order to identify lock-worth block addresses extra instructions are inserted at the end of the program however, the cost of executing these instructions are negligible.

Ghosh [17] propose an algorithm for efficiently determining the cache parameters that would improve performance. This

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.12

approach is useful in embedded applications where the applications being run is known.

We previously observed ([1], [8]) that the degree of non-uniformity is application dependent. In some benchmarks, even if most accesses fall to a small number of sets, most of these accesses are hits (as in Figure 1). Thus a single technique will not address the needs of all applications and different solutions are needed for different applications. Fortunately several different techniques are available and we compare these techniques in this paper.

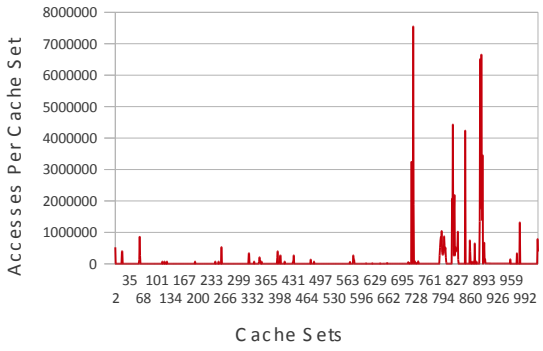


Figure 1 Non-Uniform Cache Accesses for Mibench benchmark FFT

This data can be used to explore the idea of using different indexing schemes for each thread or application when they are running concurrently in multithreaded and multicore processors. In this paper we include preliminary results of multiple indexing techniques for SMT like multithreaded systems.

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 index bits, defining modulo 2^m hashing (Figure 2)

TAG	INDEX	BYTE OFFSET

Figure 2: Cache Address Mapping

A more general view treats this process as finding a *hash function* that maps a given key (representing the specified address) to a bucket in which the data may (or may not) be found. Cache access uniformity can be improved by finding a "*perfect hash function*" by using different m bits of the address for set index (or the bucket to store data). The size of the bucket will determine the set-associativity: 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 [6]. In the Section 2 we will present several heuristics for computing cache indexes.

1.2. Dynamic relocation of addresses (or programmable associativity).

As stated previously higher associativities can lead to more uniform utilization of a cache and reduce its conflict misses. In pseudo-associative caches [10], 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. This approach provides a higher associativity only when needed. Consider the following variation to pseudo associativity (Figure 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 and the Partner index identifies the second cache line. We can select less frequently used (or cold) cache lines as partners to more frequently used (or hot) cache lines. We can either use profiling, or dynamically match cache lines as partners by keeping count of accesses and/or misses to each set. In principle we can extend the "partner index" idea to create a linked list of cache lines, effectively increasing the set-associativity for selected "hot" sets. 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 added cycles to find desired data.

TAG	V	L	Partner	DATA

Figure 3: Programmable Associativity

Other implementations can achieve similar goals with restricted flexibility. We will describe two such techniques in Section 3. It should be noted, however, most of these techniques incur additional cycles on a miss in the primary location, to locate the item in a secondary location.

II. Optimal Cache Indexing Schemes

Traditional cache addressing methods use the lower order address bits as an index into the cache, as described in the previous section (see Figure 2). In this section we describe several techniques for selecting optimal indexing.

A. Givargis

Givargis [6] relies on address traces resulting from a program execution. From the traces, the unique addresses accessed by the program are identified. Two measures are defined 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 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:

$$Q_i = \frac{\min(Z_i, O_i)}{\max(Z_i, O_i)} \quad . \qquad (1)$$

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

$$C_{i,j} = \frac{\min(E_{i,j}, D_{i,j})}{\max(E_{i,j}, D_{i.j})} \quad . \qquad (2)$$

Where E_{ij} and D_{ij} denotes the number of times bits i and j have identical or different, 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 computed. The next high quality bit is then selected and the correlation vectors are updated, and this process is repeated until the required number of index bits are selected.

B. Prime Modulo

Unlike the Givargis approach, which relies on address traces, the Prime Modulo technique [7] utilizes a different hashing function without regard to specific address traces. In Prime modulo hashing, the set to which a given address maps is computed using modulo of a prime number (instead of using the number of sets for computing the index as done in conventional caches).

$$\text{Cache Index} = \text{Address modulo } p \qquad (3)$$

The prime number p is selected such that it is less than or equal to the number of cache sets. There are several drawbacks with this hashing scheme. It is expensive to implement in hardware given the prime modulo arithmetic. The computation is likely to take several cycles. Cache fragmentation occurs because not all the sets in the cache will be utilized. However, with larger caches the fragmentation is not significant.

C. Odd-Multiplier Displacement

In odd-multiplier [7], a multiple of tag value is added to traditional index bits, and modulo arithmetic is applied to the resulting value – the modulo depends on the number of sets, as with traditional caches.

$$\text{Cache Index} = (p * T_i + I_i) \bmod s \qquad (4)$$

Here p is the odd integer uses as the multiplier, T_i is tag and I_i is the index of the address; s is the number of cache sets.

This function is based on hashing functions in [5] and is related to Raghavan and Hayes's RANDOM-H functions [12].The choice of the odd-multiplier determines the performance of this technique. The authors recommend using 9, 21, 31, and 61 as multipliers.

D. Exclusive-OR Hashing

In exclusive-or hashing [7], the set index bits are exclusive-OR-ed with selected bits chosen form the tag portion of the address.

$$\text{Cache Index} = (t_i \text{ XOR } I_i) \bmod s \qquad (5)$$

Here t_i is a portion of the tag and I_i is the index of address (the number of tag bits selected is equal to the number of index bits).

The Exclusive-Or 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 exclusive-or-ed with index bits, the conflicting addresses will be mapped to different cache sets. However, this may cause conflicts with other addresses. Since caches exhibit non-uniform accesses patterns, XOR technique has the potential to reduce conflicts at heavily accessed sets.

E. Givargis-XOR

We propose a hybrid technique that combines Givargis' approach with the XOR indexing. We select high quality and low correlation tag bits using Givargis' approach and then Exclusive-Or these bits with index bits before finding a cache set.

F. Optimal Indexes

Patel [9] exhaustively searches for the index bit combinations that results in the least number of conflict misses for a memory trace. To obtain the cost for a given index combination we simply sum the conflict pattern (CP) for all addresses in the trace as described below.

$$Cost = \sum_{i=0}^{L-1} CP_i \qquad (6)$$

Here L is the length of the entire trace file, the CP_i for an address is the Boolean condition, which includes all possible conflict conditions between an address and its successors in the trace. CP_i can be computed by OR-ing all the direct conflict patterns (*DCP*'s) relative to address a_i and its successors DCP_{ik}.

$$DCP_{i,k} = \wedge_{k=1,n} \gamma_k^* \, y'_k \qquad (7)$$

The direct conflict pattern between 2 addresses a_i and a_j can be represented as the Boolean condition for which a_i and a_j would map to the same cache set.

III. Programmable Associativity

In principle, a fully associative cache with a perfect replacement policy will access all cache lines uniformly, because data can be placed anywhere in the cache. However, fully associative caches with perfect replacement policies are not realistic and only serve as a theoretical lower bound for cache miss rates. In this section we describe some techniques that increase the effective associativities for cache lines that incur higher misses, without increasing the associativity of the entire cache.

A. Column-associative Cache

In column-associative (or pseudo-associative) caches [2], 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. When there is a miss in both locations, the data residing in the original index location is moved to the alternate location, instead of being evicted 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.

B. Adaptive Group Associative Cache

In Adaptive-cache [11], conflicting data items are relocated to new sets by using two tables. SHT (Set-reference History Table) keeps only the set indexes corresponding to Most Recently Used (MRU) sets. The OUT (Out-of-position directory) maintains indexes selected from Least Recently Used (LRU) 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 however, the data is not present in the cache, the OUT directory is accessed to see if an entry in this structure 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 (based on direct mapped index) 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. 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 a nearby disposable 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. The disposable bit of the entry is reset and the tag of the data being evicted to an alternate location is stored in the OUT directory. The SHT is updated on every access to maintain the table of MRU sets. The performance of this technique depends on the number of entries in SHT and OUT tables. Based on empirical results, possible sizes for the SHT and OUT are 3/8 and 4/16 of the number of lines in the direct mapped cache.

This technique can be viewed as selective victim caching [14] since not every evicted data is place in the victim cache; only victims belonging to MRU sets.

C. B-Cache

Zhang's B-cache [13] reduces accesses to frequently missed sets and increases the accesses to less active sets. In this method, we will extend normal index bits with additional bits. The combined index bits are divided into Programmable (PI) and Non-Programmable Index Bits (NPI). NPI work as traditional index bits while programmable index bits work as associative matching. While the potential associativity possible is given by 2^{PI}, B-caches uses a subset of combinations resulting from PI bits. One can select different combinations possible with PI bits in order to increase associativity with some indexes (or NPI combinations) and restrict other indexes to fixed locations. Thus it is possible to selectively increase associativity for highly utilized sets. The length of the programmable and non-programmable index is determined by the mapping factor (MF) and B-cache associativity (BAS), as expressed below.

$$MF = \frac{2^{PI+NPI}}{2^{OI}}. \qquad (6)$$

Here OI (original index) 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.

$$BAS = \frac{2^{OI}}{2^{NPI}}. \qquad (7)$$

In this paper, we compare the techniques presented in this section and the previous section. We will also explore hybrid techniques that combine indexing methods (Section 2) with programmable associativities.

IV. Experimental Methodology and Results

In this evaluation we use MiBench benchmarks (we are currently repeating our experiments with SPEC as well as HPC applications). We use SimpleScalar tool-set to simulate different cache configurations [4]. SimpleScalar is a cycle accurate processor simulator that supports out-of-order issue and execution. All the benchmarks used are compiled for the Alpha Instruction-set-architecture. We simulated an out-of-order processor and collected the miss rates and accesses per set to assess the uniformity achieved with the schemes discussed in this paper.

The cache configuration used in the simulations is as follows: 32kB direct mapped L1 data and instruction caches with 32 byte blocks. We used a unified L2 cache with 256kB and an LRU replacement policy. The baseline configuration against which all the schemes are compared is a direct-mapped cache with 1024 sets and 32 bytes a line, using traditional 10 bit indexes (i.e., modulo 1024 hashing).

The sizes of OUT and SHT tables for the Adaptive cache are 3/8 and 4/16 of the number of cache sets respectively. The replacement policy used in our B-cache implementation is LRU.

A. Comparing Cache Indexing Schemes

We not only explore the performance improvement attained by the schemes described in this paper, but we also evaluate the non-uniformity 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 comparing Givargis, Odd multiplier, Prime modulo and XOR schemes for Mibench benchmarks.

Figure 4 shows the percentage reduction in cache misses achieved using these techniques, when compared to conventional cache indexing. A negative value indicates that the use of a given indexing function increases cache miss-rate when compared to traditional directly-mapped cache. Figure 4 shows that none of the techniques perform consistently well. On average, Givargis' technique displays the worst performance among the techniques studied in this paper. In using Givargis' method, we did not use bits from byte-offsets to find high quality index bits. Ignoring these bits appear to impact the performance. Thus for smaller cache blocks (say 8-bytes), fewer bits are ignored in finding index bits, and Givargis's method appears to show better performance for such caches, but perform poorly for caches with wider cache lines (say with 32 or 64 bytes).

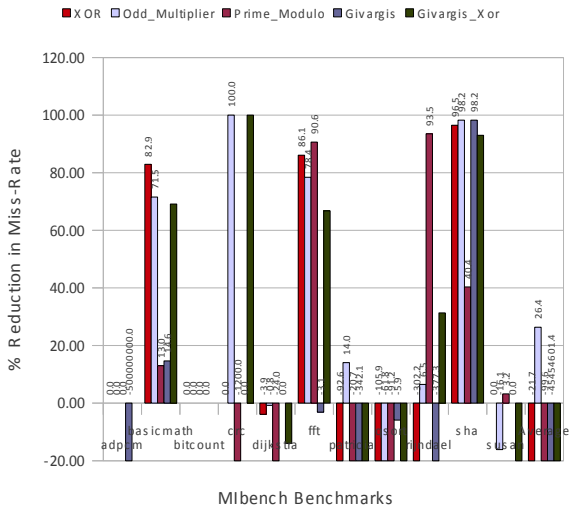


Figure 4: Cache miss rates for different indexing methods

While this study indicates that none of the techniques consistently outperform traditional modulo indexing, some specific applications benefit from a specific indexing scheme, such as odd-multiplier, prime-modulo or XOR schemes. Some indexing schemes are ideal for some applications and reduce performance for others. Also XOR and Givargis's approache require minimal hardware extensions.

One of our research goals is to explore multiple indexing schemes within a single cache system as shown in Figure 5. Applications can be profiled off-line to determine the indexing scheme that yields fewer misses. The default will use conventional indexes. During the execution of an application, system will be set to use the chosen indexing scheme. We will show some preliminary results of using multiple indexing schemes for multithreaded applications later in this paper.

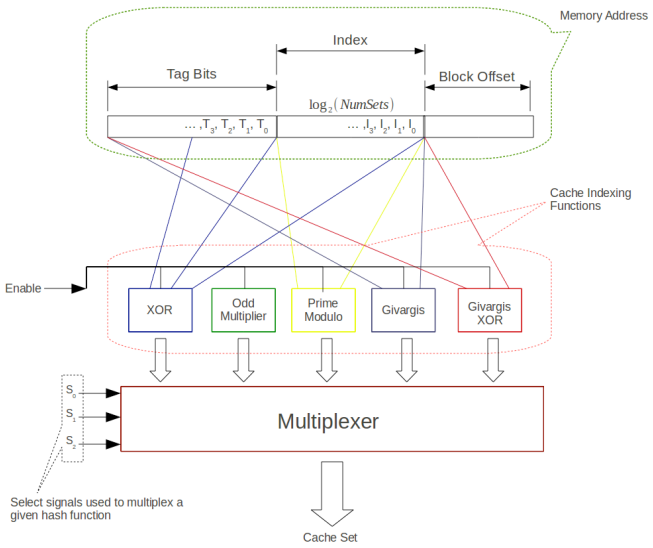


Figure 5: Proposed design to select optimal indexing scheme depending on the application.

B. Comparing Programmable Associativity Schemes

Figure 6 show the percentage reduction in miss-rates achieved by the techniques discussed in Section 3 (labeled as programmable associativities) when compared to traditional direct mapped caches for Mibench benchmarks. While column-associative cache shows higher improvements for most applications, all three techniques show reduction in cache misses. It has been observed [13] that the B-Cache as implemented here achieves the same miss rates as an 8-way associative cache, while using a direct-mapped cache. The B-cache posts the smallest performance improvements for all the Mibench applications evaluated.

Since some of the methods compared require longer access times (including searching in alternate locations), we compared the average memory access times for these schemes. The adaptive cache incurs 3 extra cycles if there is a a miss in primary cache set and a hit in the OUT-directory. This is due to the additional 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 hit to the cache and the other for hits in the OUT-directory. The formula used to compute the average memory access-time for the adaptive-cache is given below.

$$AMAT_{AdaptiveCache} = (FractionofDirectHits \times 1cycle) + ((1 - FractionofDirectHits) \times 3cycles) + (MissRate \times MissPenalty) \quad (8)$$

In the case of a column associative cache, the hit-time and the miss-penalty are comprised of 2 different components. Similar to the adaptive cache, a hit during the first lookup of the cache incurs 1 cycle, a hit in the second cache lookup incurs 2 cycles.

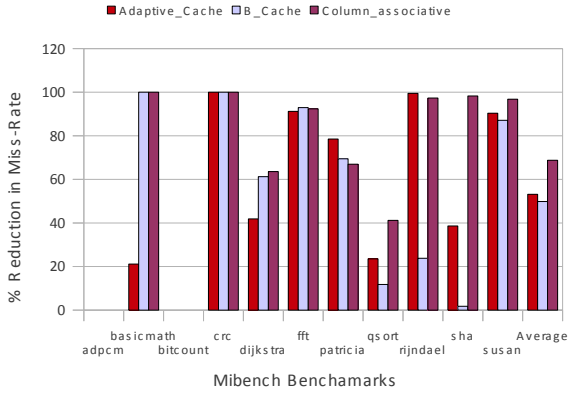


Figure 6. Cache miss rates for different programmable associative techniques

The same is true for the miss-penalty, the miss-penalty during the rehash lookup incurs one additional cycle-time. The rehash-bit ensures that the rehash lookup occurs less frequently, since it uniquely identifies cache lines, which have been rehashed. The formula below shows the AMAT calculation for a column associative cache.

$$AMAT_{ColumnAssociativeCache} = (FractionofRehashHits \times 2cycles) + ((1 - FractionofRehashHits) \times 1cycles) + ((FractionofRehashMisses \times MissRate) \times (MissPenalty + 1)) + (((1 - FractionofRehashMisses) \times MissRate) \times MissPenalty) \quad (9)$$

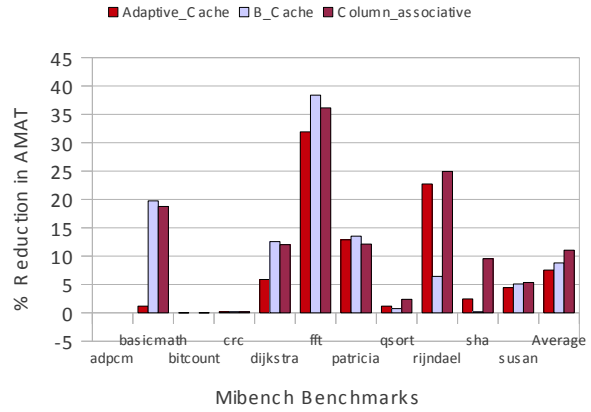


Figure 7. Percent reduction in average memory access-times when compared to a directly mapped cache.

For some benchmarks, for example bit-count, CRC and quick-sort, the performance improvements are negligible. These benchmarks access all cache sets uniformly and there are very few conflict misses that can be eliminated by Column Associative, Adaptive or B-Caches.

We explored a modification of the column-associative cache, where non-conventional indexing schemes are used as the initial index into the cacahe. More specifically, we compared the column-associative cache using traditional indexing for non-programmable indexes and using Exclusive-OR, Prime-Modulo and Odd-Multiplier techniques. Figure 8 shows the percentage reduction (or increase) in misses resulting from the various indexing schemes when compared to a column-associative cache. From the figure, pairing the column-associative cache with odd-multiplier indexing scheme shows the most improvement in performance. For some benchmarks the performance detilorates when non conventional indexing schemes are used like with Calaculix and Sjeng.

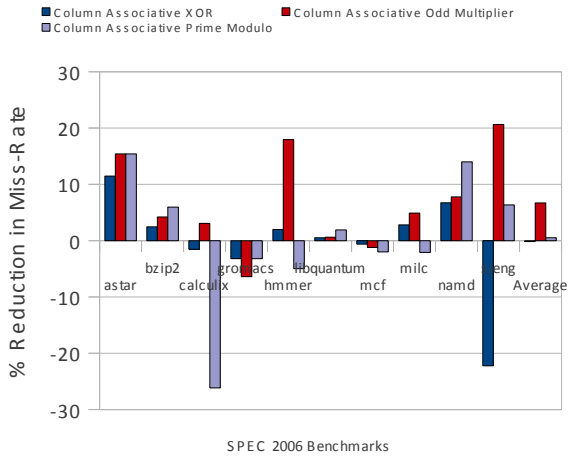


Figure 8. Reduction in miss-rates when XOR, Odd-multiplier and Prime-modulo indexing schemes are used as the primary index to a column-associative cache.

C. *Cache Uniformity*

While results so far show some improvements in terms of miss rates, and average memory access times, we also wanted to explore the uniformity of cache accesses achieved by these various techniques. Zhang [13] measured uniformity by computing the percentage of sets that are "Frequently Hit (FHS)", "Frequently Missed (FMS)", and "Least-Accessed (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 accesses. In order to more formally describe the behavior of cache access patterns, we convert the accesses, hits and misses into probability distributions. We can then measure various statistical values known 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.

D. *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 is peaked or flat relative to a normal distribution. That is, data sets with high Kurtosis tend to have distinct peaks and 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 using these schemes, we computed the kurtosis and skewness of misses in each of the 1024 sets (we did not report these statistical measures for hits, since non-uniform hits do not cause performance penalties). The results are shown in Figures 9-12. An increase in Kurtosis and skewness indicate that the technique actually excasberated the non-uniform behavior of cache misses. A reduction of Kurtosis and Skewness indicates that the technique improved uniformity of misses to different cache sets. The figures show that while the different indexing techniques (viz., Givargis, Prime Modulo, Odd-Multiplier, XOR) improve uniformity of misses for some programs, the improvement is not significant. These techniques actully increase the non-uniformity for some benchmarks. However, the programmable associativity techniques (viz., Adaptive and B-Caches) show significant improvements in unformities of misses (reduced Kurtosis and Skewness).

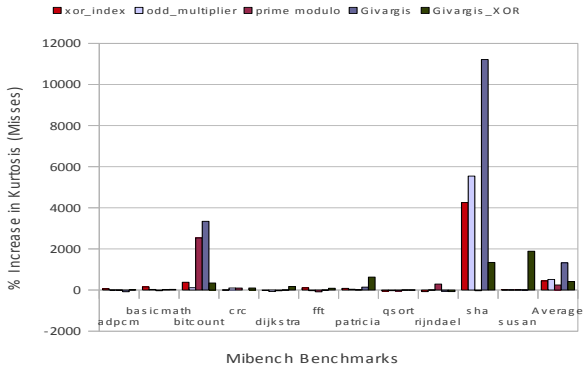
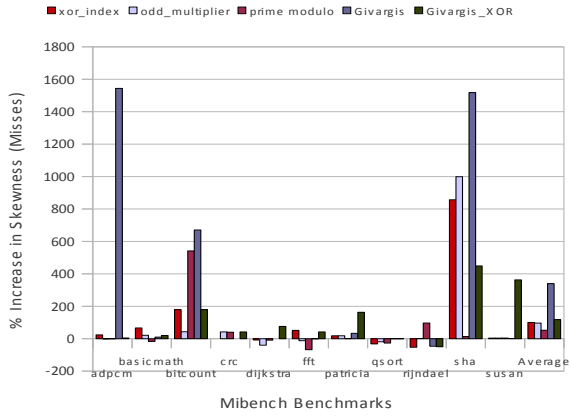


Figure 9: Kurtosis of Misses for Different Indexing



Schemes

Figure 10. Skewness of Misses for Different Indexing Schemes

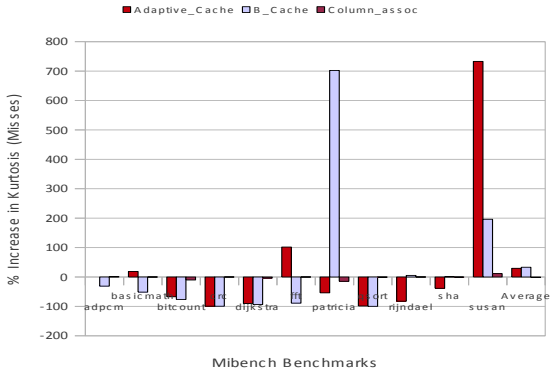


Figure 11. Kurtosis of Misses for Different Programmable Associativities

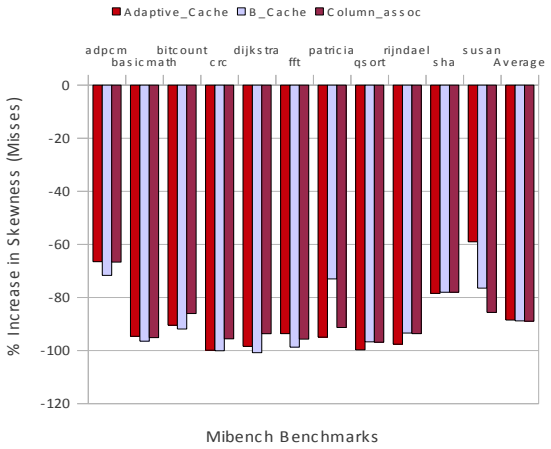


Figure 12. Skewness of Misses for Different Programmable Associativity Schemes

E. Multiple Indexing Schemes

In this section we show some preliminary results of using different indexing techniques with different threads in a multithreaded system. We use M-Sim to simulate SMT like multithreaded system. We evaluated the reduction in cache misses when 2 concurrent threads are executing, with each thread using a different indexing schemes. In our initial experiments we used odd multiplier technique (see Section 2) with different multipliers for each thread. Figure 13 shows the results.

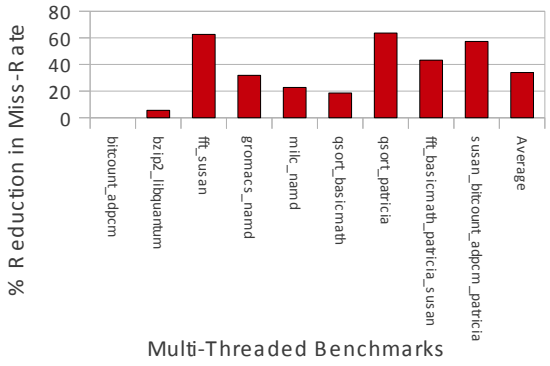


Figure 13. Multiple Indexing Schemes in Multithreaded Systems

The figure shows significant reductions in cache misses when different indexing schemes are use with each of the two threads (the two names listed with each bar).

We also explored how the Adaptive caches can benefit multithreaded systems. In this experiment we divided the cache equally among the two threads. But we used Pier's SHT and OUT tables so that lightly used sets in one partition could be used to place displaced data from the other partition, thus increasing the cache sizes available to each thread adaptively. Figure 14 shows the results.

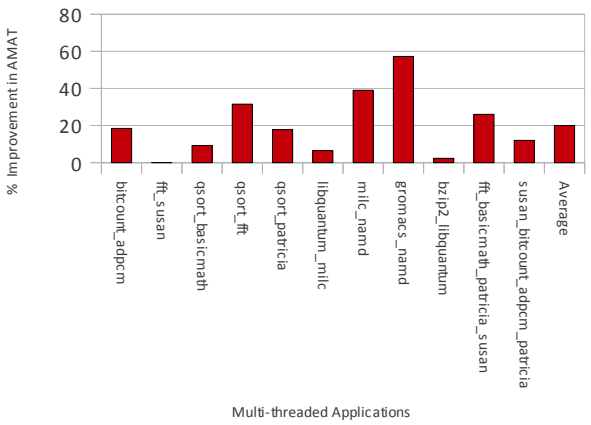


Figure 14. Percent reduction in AMAT for multithreaded applications using the adaptive partitioned scheme

V. CONCLUSION

Two classes of techniques have been evaluated in this work. Techniques that employ non-traditional hash functions to index into the cache and programmable associativity techniques. Indexing techniques incur much lower implementation overhead, with regards to area and complexity. In general XOR performs well and incurs minimal hardware extensions. Still all indexing techniques, unlike programmable associativity techniques, are static; as they do not adjust dynamically to a given applications memory access pattern. As a result the performance of cache indexing schemes vary significantly with the applications memory access behavior.

Programmable associativity schemes on the other hand, results in much better improvements in performance. In evaluating the merits of these programmable associativity schemes the performance impact of the overhead incurred by these techniques have to be taken into account. In addition, the hardware costs have to also be evaluated. The adaptive cache yields performance improvements however, it has significant hardware overhead that are introduced in order to assess cache set utilization in hardware and leverage this information to achieve better uniformity. The simplest of the programmable associativity schemes is the column associativity cache, which moved to-be evicted blocks to an alternate location. Unlike the adaptive-cache and the B-cache, the column associative cache does not rely on an in-depth analysis of the cache behavior, but naively addresses the uniformity problem. However, it posts the greatest reduction in AMAT when compared to the other programmable associativity schemes.

We also introduced a technique to improve the performance of multithreaded applications. This technique improves performance by combining the benefits of thread isolation with the ability to identify less-frequently accessed sets and divert traffic away from frequently accessed sets. Experiments show that this scheme can reduce the AMAT in applications by 60% for some multi-threaded applications.

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

VI. REFERENCES

[1] O. Adamo, A. Naz, K. Kavi, T. Janjusic and C.P.Chung. "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*) Kaosiung, Taiwan, December 14-16, 2009

[2] A. Agarwal and S. D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches." In *Proc. of the Int. Symp. on Computer Architecture,* 1993, pp. 179–180.

[3] A. V. Aho and J. D. Ullman. Principles of Compiler Design, chapter 7..6, pages 434.8. Addison-Wesley, 1997

[4] D. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin-Madison Computer Sciences Dept. Technical Report #1342, June 1997.

[5] K Ghose and MB Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation." In *Proc. of IEEE Int. Symp. on Low Power Electronics and Design*, 1999.

[6] T. Givargis, "Improved Indexing for Cache Miss Reduction in Embedded Systems," In *Proc. of Design Automation Conference*, 2003.

[7] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using PrimeNumbers for Cache Indexing to Eliminate Conflict Misses," *Proc.Int'l Symp. High Performance Computer Architecture,* 2004.

[8] A. Naz, O. Adamo, K. Kavi and T. Janjusic. "Improving uniformity of cache access patterns using split data caches", *Proceedings of ISCA PDCS-2009,* Sept. 2009, Louisville,

[9] K. Patel, E. Macii, L. Benini, and M. Poncino. Reducing cache misses by application-specific re-configurable index-ing. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design* (ICCAD '04), pages 125–130, 2004.

[10] D.A. Patterson and J. Hennessy: Computer Organization and Design, 3rd Edition. Morgan Kaufmann Publishers. San Francisco, 2005.

[11] J. Peir, Y. Lee, and W. Hsu, "Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology." In *Proc. of the 8th Int. Conf. on Architectural Support for Programming Language and Operating Systems*, 1998, pp. 240–250.

[12] R. Raghavan and J. Hayes. On randomly interleaved memories. In *Supercomputing*, 1990

[13] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. ACM I*nternational Symposium on Computer Architecture*, pages 155–166, June 2006.

[14] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Comput. Archit. News, 18(3a):364–373, 1990.

[15] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In DAC '10: Proceedings of the 47th annual Design Automation Conference.

[16] Liang, Y. and Mitra, T. Improved procedure placement for set associative caches. In Proceedings of CASES. 2010, 147-156.

[17] Arijit Ghosh , Tony Givargis, Cache optimization for embedded processor cores: An analytical approach, ACM Transactions on Design Automation of Electronic Systems (TODAES), v.9 n.4, p.419-440,October,2004.