Reducing Conflict Misses in Caches by Using Application Specific Placement Functions

Pepijn de Langen Ben Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Phone: +31-15-278-3644
E-Mail:{pepijn|benj}@ce.et.tudelft.nl

Abstract— **Most if not all contemporary processors use caches to hide memory latency. In order to maintain a high clock speed, chip designers often resort to L1 caches that have no associativity, i.e.: direct-mapped caches. Since most processors in the past were designed to run a variety of applications, these caches were also designed to perform well on a variety of applications. Currently, however, many processors are embedded into devices that perform a dedicated task. As a result, application specific optimizations have become much more interesting than in the past. Memory addresses are mapped to cache lines by so-called indexing or placement functions. In traditional caches, this mapping is done by selecting a series of least significant bits from the memory address. In this work, we investigate the possibilities of adapting cache placement functions to specific application behavior. More specifically, we target to improve the performance of direct-mapped caches by using improved bit-selection placement functions. We evaluate several offline techniques to find these improved placement functions.**

Keywords: caches, memory, mapping, indexing

I. Introduction

In the vast majority of caches, data is stored in cache lines. These lines typically store data from several consecutively memory locations. In order to differentiate between the different memory locations on a cache line, a *block-offset* is used. An *index* is used to select the proper cache line to store or retrieve certain data. To differentiate between different memory locations that are mapped onto the same place in the cache, a *tag* is stored with each cache line.

Traditionally, caches were designed as follows. Assume a cache for a 32-bit address space consisting of N cache lines, which are each 2^B bytes in width. In a traditional cache organization, the block-offset is found by selecting the B least significant bits of an address. Bits B to $B + N$ are then used to as the index, and the remainder is used as the tag.

This research was supported in part by the Netherlands Organisation for Scientific Research (NWO).

Most applications perform well on this traditional cache design. However, specific applications might benefit from a more tailored configuration. More specifically, tailoring the function which maps memory addresses to cache lines might improve cache performance significantly by reducing conflicts between certain data structures. This translation from memory addresses to cache lines is often referred to as the mapping, hash, or placement function.

The simplest and most used approach to this placement function is using a bit-selection of the original memory address. Several researchers have proposed solutions to finding improved placement functions based on bit-selections. Givargis [2] proposed a heuristic algorithm which determines qualities for different bits based on the relative number of occurrences of zeroes and ones. This is combined with correlation with already selected bits to determine a sub-optimal bit-selection. Patel et al. [5] proposed an algorithm to find the optimal bit-selection for a certain trace of memory addresses. The downside of this algorithm is that is requires a significant amount of time to finish.

Several researchers have investigated the possible improvements that can be attained by using other placement functions. The most promising proposal in this area is known as XOR-mapping [3], [6], where two bits from the original memory address are used to determine one bit in the cache index. Other researchers have proposed alternative ways to construct the indexing function based on, for example, prime numbers [4]. Others have used an approach where the compiler was instructed to perform address placement in a way to reduce data cache misses, by using profiling information [1].

In this work, we investigate the possibility to im-

prove cache performance by using an alternative selection of bits to form the index. We review some existing offline approaches to this problem and investigate the possibilities of these and other techniques.

This paper is organized as follows. In Section II, different approaches to the problem of finding improved indexing functions are explained and discussed. Experimental results of different approaches are presented in Section III. Conclusions are drawn in Section IV, where we also present some pointers for future research.

II. Indexing Algorithms

In this section, we review different approaches to how direct-mapped caches can be indexed. Traditionally, the least significant bits in a memory address are used for the block-offset. The next bits are then used to index the cache, while the remainder is used as a tag. Since many applications exhibit a great deal of spatial locality, this choice often provides good cache hit-rates. Nonetheless, certain applications might incur a significant improvement when using a different set of bits to index the cache.

In this work, we assume that the least significant bits are always used to denote the block-offset, to make sure that larger data formats are stored in a single cache line. Otherwise, loads and stores might be serviced from several different cache lines, incurring extra cycles to combine bytes to larger data formats.

Two offline approaches to find improved selections are outlined below. First, we present our implementation of a (non optimal) heuristic approach proposed by Givargis. Thereafter, we describe an algorithm by Patel et al., which returns the optimal result set of index bits for a certain trace of memory addresses.

A. Givargis

The fundamental method in the heuristic algorithm proposed by Givargis [2], is to generate a selection based on qualities assigned to the address bits. The quality of an address bit is defined as the extend to which this bit divides the referenced addresses equally. In other words, the maximum quality is attained if there are as many 1's as 0's in this position in the list of unique addresses. If the the number of 1's and 0's are less evenly distributed, the quality will be less. The quality of bit i is defined as:

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

where Z_i and O_i denote the number of times this bit is zero or one respectively.

Besides assigning qualities to each bit, the algorithm also updates these values by correlating them with already selected bits. Instead of reading the whole trace, this heuristic takes a list of unique addresses as input. As a result, the order in which the addresses are parsed is irrelevant.

We have implemented the heuristic algorithm as described by Givargis. Because a few things were unclear from the description of the algorithm, we had to make a number of assumptions. First, it is not mentioned how block-offset bits are used in this scheme. From the results in the original paper, it seems that all bits are used to select the proper index. We feel this is incorrect, because it would be extremely hard to fetch a multi-byte value from the cache if these bytes are not stored consecutively. Second, Givargis reports the same improved index mappings for three caches with different line sizes. Since the original paper makes no mention on how block-offset bits are used, this might have been neglected in this paper, rendering the results incorrect. In our implementation, the bits that are used for block-offset are not considered when constructing the placement function. Furthermore, our implementation uses a list of unique addresses with these bits removed. In other words, we first translate all byte addresses to block addresses.

B. Patel et al.

Patel et al. [5] proposed an algorithm to find the set of index bits that results in the least number of cache misses for a certain data stream. Our implementation is a slightly modified version of this algorithm due to some implementation problems. The differences are outlined below.

The fundamental basis of this algorithm is the *direct conflict pattern ($DCP_{i,j}$)*, which expresses a boolean condition for which a_i and a_j would conflict with each other. The DCP actually denotes which bits are different between two addresses. If none of these bits are used for the cache index, this would imply that both addresses are mapped to the same cache line, causing a conflict. The DCP can easily be found by performing a bit-wise XOR on two memory addresses.

Patel et al. then define the *total conflict pattern CP_i*. The CP denotes all possible conflicts between address a_i and its successors in the memory trace. The CP can be computed by OR-ing all DCP's between address a_i and its successors. Successors in this case are the addresses between a_i and the next reference

to a_i. If a_i is the last reference to that cache line, the DCP is defined to be 0. In the work by Patel et al., the number of conflict misses are then found by summing the CP for all addresses in the trace.

We found, however, that combining DCP's into a CP might prove to be rather difficult. In many cases, combining these DCP's results in a CP which cannot be expressed very efficiently. In fact, combining N DCP's can easily result in a CP with N terms. Since the distance between two accesses to the same cache line can be significant, this turned out to be hard to implement. Furthermore, we would like to point out that Patel et al. incorrectly state that the expression in their example can be simplified by removing redundant terms. In this example, this term actually refers to a different miss condition and is therefore definitely not redundant.

In our implementation, we first construct a binary tree which expresses all possible combinations of index bits. The top node determines whether or not the lowest order bit is used for the index. The succeeding nodes determine this fact for the next bit, until exactly 10 bits are selected. Every node has an associated cost and a timestamp denoting the last update. The end-nodes in this tree express valid combinations, which have exactly 10 bits selected. Instead of combining DCP's into a CP, we directly update the tree with information from the DCP. A DCP in fact denotes one or several branches in this tree. In our implementation, we increment the *cost* of all branches denoted by the current DCP. To make sure all DCP's relative to a_i can cause at maximum one miss, we also record the timestamp of the last update in each branch. By checking this timestamp, we make sure a branch is never incremented more that once for all DCP's that originate from one address. The total cost for a certain configuration is then found by adding all the costs on the path from the root of the tree to an end-node.

III. Experimental Results

We have implemented two algorithms based on the work by Givargis and Patel et al. respectively. In this section, the results from experiments with these algorithms are discussed.

As noted before, the results provided by Givargis indicate that all bits are available to be used for indexing the cache. In our opinion, it would be extremely hard to build a cache in which sub-word bytes are not stored consecutively. Furthermore, Givargis also reports improved indexes in which the lower order bits are not used. Especially in the case of instruction caches, but definitely also for data caches this would imply a total absence of spatial locality beyond the cache line size. Third, Givargis reports the same improved indexes independent of the used cache line size. Different cache line sizes imply a different number of bits to be reserved for the block-offset. As a result, we would expect the improved configurations to be different as well. Lastly, the heuristic algorithm proposed by Givargis is not optimal. In fact, it does not take into account in which order references appear, nor does it use any information on the number of times data is referenced. Therefore, one would expect to see at least some examples of performance degradation. In the results presented by Givargis, performance is always improved. Because of these anomalies, we cannot trust the results provided by Givargis and will therefore not compare them to our results.

We have performed experiments with two different cache configurations: an 8kB cache with 8 bytes per cache line and 32kB cache with 32 bytes per cache line. Both caches are direct-mapped and have 1024 cache lines, indexed by 10 bits. The reasons for experimenting with these two different configurations are the following. Both Patel and Givargis use a rather small line size in their experiments. We also include experiments with a small cache line size in in order to make the results comparable. We have decided for a 8 byte line size since our traces are based on mostly 8 byte loads and stores. We have included a cache with 32 byte per cache line, since this is actually more representative of modern processors. In order to make the indexes comparable we have decided to keep the number of index bits fixed to 10, resulting in a 8kB cache and a 32kB cache.

Tables I and II list the found configurations for both short (100,000 references) and longer (1 million references) memory traces. The number of misses resulting from these configurations are depicted in Figure 1. For the 32kB caches, the configurations are listed in Tables III and IV, and the number of misses resulting from these configurations are depicted in Figure 2. In these figures, normal refers to the baseline case, where the index is constructed out of the lower order bits.

From these figures, it can be seen that significant improvements are only attained for a few benchmarks. In general, the performance improvements upon the baseline cache are relatively small. One of the reasons for this is that the baseline cache already performs very well for a number of applications, leaving little room for improvements.

The configurations in Tables I to IV show that all

Benchmark	Givargis	Patel
crafty	00001010 00000000 00001011 11111000	00001000 00000000 01001011 11111000
eon_cook	00000000 10000010 00000111 11111000	00000000 00100000 10000111 11111000
gap	00000000 00100000 00001111 11111000	00000001 00000000 00001111 11111000
mcf	00000000 00100000 00001111 11111000	00000000 10000000 00001111 11111000
parser	00000000 00000000 00101111 11111000	00000000 00010000 00001111 11111000
twolf	00000000 00000000 11001111 11110000	00000000 00010000 00100111 11111000
vortex_one	00000000 00000100 00001111 11111000	00000010 00000000 00001111 11111000

TABLE I

Configurations attained for 8kB caches with 100,000 references.

Benchmark	Givargis	Patel
crafty	00000000 00000000 01001111 11111000	00000000 00001000 00110011 11111000
eon_cook	00000000 00000000 01100111 11111000	00000000 00100000 10000111 11111000
gap	00000000 00000000 00011111 11111000	00000000 00010000 00001111 11111000
mcf	00000000 00000000 01111111 11100000	00000000 00000000 00011111 11111000
parser	00000000 00000000 01001111 11111000	00000000 00000000 00011111 11111000
twolf	00000000 00000000 00111111 11110000	00000000 00000000 00101111 11111000
vortex_one	00000000 01000000 00001111 11111000	00000100 00000000 00001111 11111000

TABLE II

Configurations attained for 8kB caches with 1 million references.

Benchmark	Givargis	Patel
crafty	00000010 00000000 01111011 11100000	00001000 00000100 00011111 11100000
eon_cook	00000000 00010000 10101111 11100000	00000001 00100000 00101111 11100000
gap	00000100 00000110 00001111 11100000	00000010 00000010 00011111 11100000
mcf	00000000 00100000 01101111 11100000	00000101 10000000 00001111 11100000
parser	00000000 00000000 01111111 11100000	00000001 00010000 00011111 11100000
twolf	00000000 00000001 00111111 11100000	00000000 00000100 01011111 11100000
vortex_one	00000000 00000100 00111111 11100000	00000100 00000010 00101111 11100000

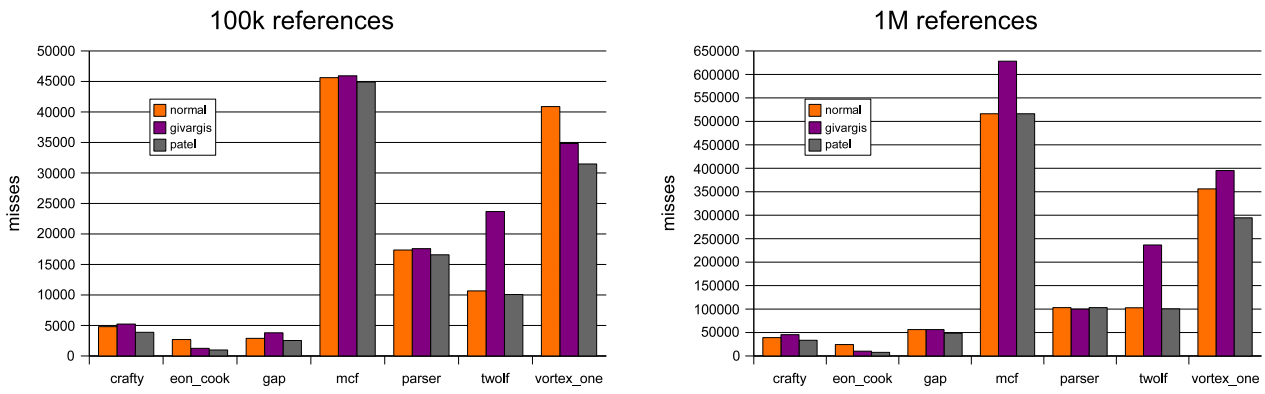
TABLE III

Configurations attained for 32kB caches with 100,000 references.

Benchmark	Givargis	Patel
crafty	00000000 00000000 01111111 11100000	00000000 00000100 01011111 11100000
eon_cook	00000000 00000000 01111111 11100000	00000001 00100000 00101111 11100000
gap	00000000 00000010 00111111 11100000	00000000 00010000 01011111 11100000
mcf	00000000 00010000 01101111 11100000	00000000 00000000 01111111 11100000
parser	00000000 00001000 11001111 11100000	00000000 00000000 01111111 11100000
twolf	00000000 00001000 00111111 11100000	00000000 00000000 01111111 11100000
vortex_one	00000000 01000000 00111111 11100000	00000000 00001100 00011111 11100000

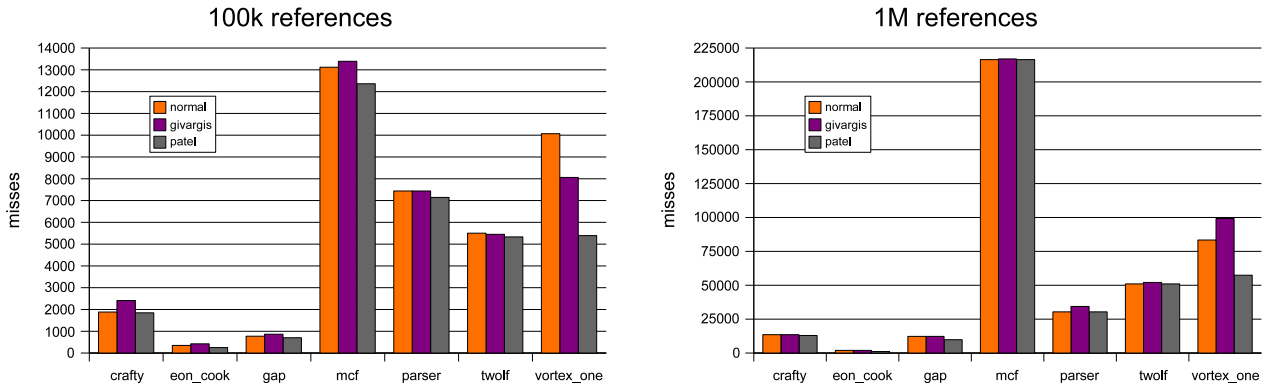
TABLE IV

Configurations attained for 32kB caches with 1 million references.



(a) (b)

Fig. 1. Results for 8 bytes per cache line



(a) (b)

Fig. 2. Results for 32 bytes per cache line

of the generated solutions are very similar to the baseline case. In most cases, the solutions differ from the baseline only in 1 or 2 higher order bits. The lower order bits in the baseline index are in general left untouched. Givargis' method prefers a high order bit over one of the lowest order bits from the original index in only three cases: `mcf` on a 8kB cache with a short trace and `twolf` on a 8kB cache with a short or longer trace. Figure 1 shows that these three solutions are exactly the ones where Givargis' method performs the worst. In these cases, the number of cache misses increases significantly compared to the baseline cache.

The method proposed by Givargis actually only performs well on few occasions. The only significant improvement that we found was for `eon_cook` with the 32kB cache, where this method was able to reduce the number of misses by a factor of about 2. However, since this benchmark already performs well on the baseline cache, the absolute reduction of the number of cache misses is quite small.

According to Figures 1 and 2, the most significant improvements are found for the `vortex_one` benchmark on all 4 caches, and for `crafty`, `eon_cook`, and `gap` for the 8kB cache when using the longer trace. As expected, the 8kB cache with 8 byte cache lines attains in better improvements than the 32kB cache with 32 byte cache lines. The main reason for this is that the bigger cache line already resolves a significant number of misses. Furthermore, the larger cache size also helps in reducing cache misses, leaving less room for further improvements than in case of the 8kB cache with small cache lines.

The most important conclusion that can be drawn from these results, is that in all optimal configurations

that we found, the 7 lowest order bits from the original index are left untouched. This indicates that, for this set of benchmarks, only the 3 higher order bits need to be considered in order to determine the optimal index bit selection. For the Patel's algorithm, this would drastically prune the solution space. More specifically, for the 32kB cache the solution space would be decreased to only 6840 different configurations. When the solution space is of this limited size, a brute-force method of checking all permutations actually becomes a feasible solution and might prove to be more efficient than using Patel's algorithm.

The method proposed by Givargis does not seem to work well. For most benchmarks, this heuristic does not find configurations that improve upon the baseline case. In fact, this heuristic reduces performance significantly in several cases.

IV. Conclusions

We have shown that application specific cache placement function can result in a decreased number of cache misses, and therefore improve performance. The results indicate, however, that only in few cases this improvement is significant. One of the reasons for this is that the used benchmarks already perform well on the baseline cache. For applications that do not perform as well on basic direct-mapped caches, these improvements might be more significant.

The method proposed by Givargis did not prove to be an effective way to find improved cache configurations. We have found few instances that improved performance upon the baseline cache. In most cases, however, this method resulted in an increase in the number of cache misses. This heuristic might be improved by incorporating information about the number of times certain data is referenced and the order in which data is referenced. We intend to investigate this in future research.

The algorithm by Patel et al. returns the optimal set of index bits. However, the amount to time needed to find this optimum is significant. Since the attained improvements are relatively small, this might not warrant the effort of performing this optimization.

None of the configurations produced by the optimal algorithm was significantly different from the baseline configuration. In fact, the 7 least significant bits in the baseline index were left untouched in all the optimal configurations we found. This indicates that, for this type of benchmark, only the 3 highest order index bits need to be considered. Limiting Patel's algorithm to only these bits will drastically reduce the time required by this algorithm. Depending on the length of the trace and the cache configuration, it might even be more efficient to just test all permutations of the higher order bits in a brute-force way. In future research, we intent to investigate this and possible ways to reduce the amount of time required by Patel's algorithm.

We conclude that application specific placement functions can in fact be used to reduce the number of cache misses. For this goal, the method proposed by Givargis did not prove to be very useful. The optimal configuration provided by Patel's algorithm showed that the significant improvements can only be attained for some applications. For the majority of benchmarks, the number of cache misses was hardly reduced. In future research, we intend to show whether other hashing methods are able to further reduce the number of cache misses for these benchmarks.

References

[1] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149, 1998.

[2] T. Givargis. Zero cost indexing for improved processor cache performance. *ACM Transactions on Design Automation of Electronic Systems*, 11(1):3–25, 2006.

[3] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through xor-based placement functions. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 76–83, 1997.

[4] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating conflict misses using prime number-based cache indexing. *IEEE Transactions on Computers*, 54(5):573–586, 2005.

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

[6] H. Vandierendonck, P. Manet, and J.D. Legat. Application-specific reconfigurable xor-indexing to eliminate cache conflict misses. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 357–362, 2006.