ПРИЛОЗИ, Одделение за природно-математички и биотехнички науки, МАНУ, том **34**, бр. 1–2, стр. 83–92 (2013)
CONTRIBUTIONS, Section of Natural, Mathematical and Biotechnical Sciences, MASA, Vol. **34,** No. 1–2, pp. 83–92 (2013)

Received: September 10, 2013
Accepted: October 22, 2013

ISSN 1857-9027
UDC: 004.424.7.021:004.332.5

Original scientific paper

ANALYSIS OF ASSOCIATIVITY AND CONFLICT MISSES

Marjan Gusev*, Sasko Ristov

Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University,
Skopje, Republic of Macedonia

*Corresponding author, e-mail: marjan.gushev@finki.ukim.mk

Cache memory is playing a huge role in determining the performance when solving scientific problems. Most of these problems include a high number of repetition of complex or simple calculations on various data elements stored as arrays in sequential order in the memory. When executing the algorithms, these elements are brought to cache and then used by the processor. This process is usually followed by conflict and capacity misses in the cache, and the performance is degraded by a cache placement function, replacement policy or capacity constraints. In this paper we analyze the algorithms and performance impact of set associative caches when a large array is referenced in sequentially ordered memory. We map the problem of cache use into an IT related mathematical model to analyze the performance and give a scientific explanation for performance drops due to associativity and conflict cache misses in caches.

Key words: high performance computing; performance drops; shared memory multiprocessor; superlinear speedup

INTRODUCTION

All modern processors use cache memory to reduce the gap in the frequency between the slower main memory and the faster CPU [1]. This is more emphasized if data is reused several times (time locality), thus reducing the average memory access time. Another example where the cache speeds up the execution are the algorithms that use data locality, when the access of a given data element is followed by an access of nearby elements. This is also used in data arrays which are accessed sequentially, in the same way they are stored in the memory. Accessing the first element will initiate accesses to next data in the cache.

Usual cache architectures organize data in small blocks, called cache lines. Due to small capacity, cache blocks can store number of data arrays and the cache placement function determines where these data are placed. When a data element is required, the corresponding cache block has to be fetched in the cache and replace another block. Cache replacement policy decides which block will be removed in this case.

Despite the advantages a cache memory provides, time demanding or data intensive algorithms usually use extensive data arrays stored sequentially in the memory. Accessing only parts of these arrays in caches follow various memory access patterns, which can significantly reduce the performance. A typical example is executing the matrix multiplication algorithm, which is presented in Figure 1. The figure presents the measured execution speed (Y-axis) measured in MFLOPS, as a function of matrix size problem N (X-axis).

Performance drops are observed as a consequence of associativity and conflict cache misses, such as those for N = 128, 192, 256 (A), 384, 512(B), 640, 768, 896, 1024, … . We have reported [2] that performance degradation is also observed for regions starting from points E, C and D, i.e. when

$$Speed_E >> Speed_C >> Speed_D$$

despite the expectation that it should be the same for each matrix size *N*.

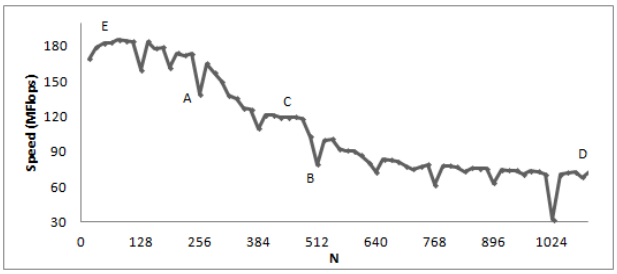

Figure 1. Measured speed for matrix multiplication executions

Let us briefly describe the reasons why these performance drops happen. Sequential data arrays can use specific data access patterns when identifying a cache block, and instead of using all available cache blocks, the algorithm might specify the usage of only one or several cache blocks. This is especially exposed by the associativity in the set associative caches. Increasing the set associativity will reduce the cache misses, but in the same time will increase the access time, and vice versa.

In this paper we define a mathematical model, based on analysis of data patterns in sequential array and used mappings in set associative caches. This analysis gives an explanation about performance degradation that happens as a consequence of the relation between the data access pattern and the cache parameters.

The rest of the paper is organized as follows. Firstly, the related work is presented. Then we present the theoretical analysis of storage patterns, mapping functions and their properties that initiate conflict misses. After the presenting the experimental proof about theoretical explanations for given examples, we conclude our work and present the plans for future work.

RELATED WORK

The cache set associativity impact to the performance has been observed by many authors. For example, Tsilikas and Fleury [3] reported performance drawbacks for some problem sizes when they analyzed the interaction of matrix multiplication (non-linear memory accesses) with the memory hierarchy.

Most explanations found in literature claim that performance degradation happens due to conflict misses and do not give any further detailed explanation. However, there were several engineering solutions and proposals on how to avoid or reduce associativity problems and cache conflict misses. The conventional background for this is the idea to change the storage data pattern and reduce the average memory access time in set associative cache. For example, Sen et al. [4] report how matrix transposition can eliminate conflict misses in the matrix multiplication and enable further faster processing. The idea for padding the data to reshuffle the data pattern will amortize the performance drawback, since it will avoid associativity problem [5].

Several authors propose different cache organization and improving access time by hardware or software optimization techniques. For example, Hongil et al. [6] propose a dynamic and optimized replacement policy for each cache set via workload speculation mechanism. A new software runtime library was designed by Ding et al. [7], which could make the cache memory more intelligence in order to allow the programmers to allocate arbitrary last level cache space for various data sets of different threads.

In our recent research, we have analyzed the cache associative problem for matrix multiplication where the data pattern create performance drawback for a specific matrix size *N*. We have theoretically defined the matrix sizes as a function of cache parameters and experimentally proved the existence of the critical points where maximum performance drawbacks appear [8]. Our theory was proved also for GPUs since they have also set associative cache [9].

THEORETICAL ANALYSIS

This section presents the theoretical background of the performance analysis when accessing a large sequentially ordered memory array by a processor using cache memories.

CACHE MEMORIES

Caches are small memories, which use the benefit of fast access by the processor. According to the algorithms that map memory blocks onto cache blocks we differ three main organizations of caches: directly mapped, set-associative and fully associative. A direct mapped cache maps a block onto a specific location in the cache obtained by a modulo function. In set associative caches, a block is mapped onto a set and then onto a block within a given set. A fully associative cache uses a strategy when the block is mapped onto a location chosen by a cache replacement strategy.

Modern processors use a cache hierarchy of 2 or 3 cache levels and each level has different parameters or strategy where a data element will be

positioned, so in reality, a data element is mapped onto a different position in all cache levels.

Each data element is found in a cache block which is fetched from higher cache level or main memory. A cache block or cache line in modern processors may contain different number of data elements depending on the cache line size and memory data element representation. Memory element representation might be a double precision floating point number, which occupies 8 bytes of memory. It may also be a single precision floating point number or an integer, usually occupying 4 bytes of memory.

Suppose that the analyzed cache memory for level L_i, where $i = 1,2,3$ has M_{Li} blocks. Let the size of memory be M_{mem}, measured in cache blocks, which is equal to the number of memory locations (expressed in bytes) divided by the cache block size. Similar to this, the number of blocks per each cache M_{Li} is equal to the cache size in bytes divided by the cache block size.

Let us denote that the set associative cache on level L_i contains M_{Li} sets and each set contains n_{Li} blocks. For each cache level $i = 1, 2, 3$ there is a simple relation that the size M_{Li} is equal to the product of number of sets S_{Li} and their associativity n_{Li}, and the number of sets can be expressed as expressed in (1).

$$S_{Li} = \frac{M_{Li}}{n_{Li}} \tag{1}$$

The technology used to build faster caches is such that the smaller the caches is, the faster it works. Therefore, cache sizes on corresponding levels satisfies (2).

$$M_{L1} < M_{L2} < M_{L3} < M_{mem} \tag{2}$$

The following is an example of a real processor with explanation of its cache hierarchy.

Example 1 (Specifics of AMD Opteron Processor). *Quad-Core AMD Opteron(tm) Processor 9550 [10] has 4 cores each with its own dedicated 64 KB instruction and 64 KB data L1 cache, dedicated 512KB L2 cache and share 2MB shared L3 cache.*

Cache block (line) size is 64B in all cache levels and when storing double precision flo-ating numbers that use 8B presentation they can store 8 data elements and for single preci-sion floating numbers and integers that use 4B presentation they can store 16 data elements.

All caches use set-associative placing of cache blocks. The associativity of L1 cache is $n_{L1} = 2$, *and raised to* $n_{L2} = 8$ *for L2 cache and to* $n_{L3} = 32$ *in L3 cache. From cache size and cache block size we calculate that L1 cache has* $M_{L1} = 1024$ *blocks, L2 cache has* $M_{L2} = 8192$, *and L3 cache has* $M_{L3} = 32768$ *cache blocks. We conclude that (2) is satisfied.*

The number of sets for L1 cache is $S_{L1} = 512$ *and for both L2 and L3 caches it is* $S_{L2} = S_{L3} = 1024$, *according to (1).*

To simplify the notation in the following analysis, we use M as total number of blocks in the cache, S to express the number of sets in a set associative cache, n the associativity, and finally, a notation depending on the corresponding level in the form of an index.

Definition 1 (Cache placement mappings). *Let x be the memory block that contains the location of a data element requested by the algorithm. It is mapped onto:*

- *a cache memory block with location x mod M in a direct mapped cache;*
- *a set with identification x mod S in a set-associative cache, and inside the set in one of n ossible blocks determined by an appropriate block replacement strategy; and*
- *an empty block or a block chosen by an appropriate replacement strategy in a fully associative cache.*

A cache replacement policy chooses the Least Recently Used (LRU) block to be replaced, or chooses First In First Out (FIFO), random or similar strategy to change the block within a given set.

Direct mapped caches are special cases where the associativity is $n = 1$ and the number of sets is equal to the total number of blocks $S = M$. A fully associative cache is a special case where the number of sets is $S = 1$ and associativity is $n = M$. In this context we will continue to analyze only set associative caches.

Due to smaller capacity and cache associativity, there are cache misses when the processor tries to access data element which is not found in the memory. Cache compulsory misses appear as a result of the cold start, when the cache is empty and data elements are not loaded. Cache capacity misses appear since the cache cannot store all required data elements, and conflict misses appear if the mapping function is such that requested data elements are mapped in a smaller subset of blocks than the available.

The next analysis gives an overview of activities performed by the processor in a multi-level cache organization. Let us analyze what happens when a processor would like to access a data element within a block location x in memory. First, the processor identifies where it is mapped in the L1 cache level. Identification is done via mapping

by modulo function of S_{L1}. Then it is searched in the identified set and if not found in n_{L1} available positions, a cache miss is generated. The same procedure continues on the L2 cache level and then on the L3 cache level.

Suppose that the block is found on L2 cache level. Let us describe the activities that start after cache hit on a higher level and miss on analyzed cache level. A cache replacement policy decides which block will be replaced and the corresponding block is placed in the identified cache block. The activities are recognized as penalty, since more clock cycles will be used to replace the block and use the requested data element from identified cache block.

STORAGE PATTERNS

Main memory is a sequentially ordered list of data elements, and elements are accessed via their address. However, a cache memory does not use sequential ordering, as explained earlier, there are different cache organizations.

The problems analyzed in this paper concern mapping of elements from a linearly ordered list to a specific cache storing organization. During this mapping, a serious number of cache misses might appear and result in performance drops.

The side effects of using set associative caches is that an algorithm might require a series of data elements, which map onto a same cache set, and due to the cache placement, the mapping and replacement policy might be an operation that prevents using the cache benefits. On contrary it generates conflict misses and huge performance drops. In this paper we analyze these mappings determined by the cache organization and give conclusions on how to reorganize data elements and obtain more efficient algorithms.

We do not tackle cache architecture addressing modes, calculation of index, tag and other cache specific organization features that determine the architecture and organization of caches, but rather analyze how algorithms can efficiently use cache hierarchy.

Most of the analysis in cache design are performed on average behavior of benchmark programs. Various attempts have been made to reduce the cache misses and suggest an organization that will have overall best performance, including reducing miss rates with different associativity, victim caches, and compiler optimization, than reducing miss penalties with faster DRAM memories and writ buffers and reducing hit times [1]. Compiler optimizations include loop fusion, loop interchange or techniques to merge arrays. Various conclusions are brought, like lower associativity reduces the hit time, which is of high concern for L1 cache and miss penalty is critical for L2 caches, since it is larger.

In this paper we analyze the algorithms and suggest their reorganization to avoid conflict cache misses as much as possible. We assume that the algorithms will use the benefit of a cache block (line) prefetching and when a data element is accessed then the algorithm is supposed to use all data elements from the cache block. Therefore in the next analysis we refer to cache blocks rather than to data element access. Denote a series of memory accesses produced by an algorithm as an array by Definition 2.

Definition 2 (Request Block Array (RBA)). *RBA is an array of memory block locations, denoted by $x_0, x_1, x_2, \ldots$, where each element of this array x is equal to a cache block location for a block that contains data elements sequentially requested by the algorithm instructions.*

Let us start defining properties of RBA of blocks, as presented in Definition 3. We assume that there is a uniform pattern (equidistance) between two consecutive elements in the RBA.

Definition 3 (Block location offset). *Let there be an algorithm that generates a RBA of K block locations, denoted by $x_0, x_1, x_2, \ldots, x_{K-1}$. Block location offset f is obtained as a difference of block locations requested in consequent instructions*

$$f = x_{i+1} - x_i, \text{ where } i < K.$$

Example 2 gives an overview of an existing algorithm and a RBA.

Example 2 (Matrix multiplication algorithm). *To simplify the algorithm presentation we suppose that matrices have dimension N·N, i.e. N is the matrix size, and the native version of the matrix multiplication algorithm is*

$$C_{N \cdot N} = A_{N \cdot N} \cdot B_{N \cdot N}$$

Suppose that all the data elements are double precision and each occupies 8 bytes. This means that the input matrices $A = [a_{ij}], i,j = 0, \ldots, N-1$ and $B = [b_{ij}], i,j = 0, \ldots, N-1$, and also the result matrix $C = [c_{ij}], i,j = 0, \ldots, N-1$ consist of data elements, which are stored sequentially in the main memory.

The algorithm contains the following computations

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} . b_{kj}$$

At the beginning, the algorithm specifies instructions that request a sequence of memory column elements $b_{0j}, b_{1j}, b_{2j}, \ldots$. Due to sequential ordering in the main memory if the matrix size is $N > 8$ and the cache block size is 8 data elements, then each data element of this array is stored in a different cache block. In this case, RBA presents an array of N different cache blocks, for each iteration that uses a certain matrix column.

Each column element will be found in a block located on memory address $MA(b_{ij}) = MA(b_{00}) + i \cdot N + j$. The corresponding block is determined by the following block location $x(b_{ij}) = MA(b_{ij})$ DIV 8 since there are 8 data elements in a cache block.

The algorithm starts with $i = 0$ and $j = 0$, and then sequentially requests the elements of the first matrix column, i.e. the elements $b_{00}, b_{10}, b_{20}, \ldots$. The first data element to be requested by the processor is found in block $x_0 = x(b_{00}) = x$. Assuming that $N = 512$ the next data element to be requested is found in block with location $x_1 = x(b_{10}) = MA(b_{10})$ DIV $8 = x + 64$. Therefore, for $N = 512$ the corresponding RBA of blocks $x_0, x_1, x_2, \ldots$, is equal to

$$x, x + 64, x + 128, x + 192, \ldots \quad (3)$$

It follows that the block location offset is $f = 64$.

The next two lemmas can be proven by the Dirichlet's box principle (or Pigeonhole principle), which states that if there are more balls to be put than the number of available boxes, then at least one box must contain more than one ball. It is a simple counting argument, and can have several other versions, such as the one which states that it is not possible to place more than K balls in K different places, or there does not exist an injective function on finite sets whose codomain is smaller than its domain.

Lemma 1 gives a constraint about the capacity cache misses since caches are small memories.

Lemma 1. *A capacity cache miss appears if the number of RBA blocks K is higher than the number of available cache blocks M_{Li}, i.e., $K > M_{Li}$.*

Lemma 2 follows an analysis that shows how RBA will be mapped in the cache memory. The worst scenario is when set associative caches may map all RBA blocks onto a single set and direct mapped caches onto a single block location in the cache.

Lemma 2. *A conflict cache miss appears if at least n RBA blocks are mapped onto a single set of a cache memory with associativity value of n.*

So, if the associativity is n, then at most n blocks can be placed in one cache set. The cache conflict appears if the algorithm maps more than n RBA blocks onto a cache set.

In case of direct mapped caches, a conflict cache miss appears if two RBA blocks map onto a same cache block. In case of set-associative caches, a conflict miss may not appear, since there is one set and the problems might appear only as capacity misses, where the number of used blocks is higher than the available blocks.

The next analysis presents conditions for appearance of conflict misses. Let us analyze two blocks x_k and x_l of the RBA. According to Definition 1, a block is mapped onto a cache set on level L_i identified with modulo function, and denote mapped values by $y(x_k) = x_k \, MOD \, S_{Li}$ and $y(x_l) = x_l \, MOD \, S_{Li}$. We would like to express the condition that maps these two blocks x_k and x_l of the RBA onto one set in a set associative cache, i.e. $y(x_k) = y(x_l)$. It follows that $x_k \equiv x_l \, mod \, S_{Li}$. This proves Lemma 3.

Lemma 3. *A conflict cache miss appears in a cache level L_i if there are at least n blocks in RBA which have the same value of modulo function mod S_{Li}.*

As a consequence of Lemma 3 we are interested in analyzing where the blocks of RBA will be mapped according to modulo functions of the block locations. Definition 4 defines an array obtained by mapping the RBA.

Definition 4 (Mapped Block Array (MBA)). *MBA is an array of K cache block locations $y_0, y_1, y_2, \ldots, y_{K-1}$, where each block y_i is obtained as a mapped block location from a RBA block x_i, mapped by a corresponding mapping function used in the appropriate cache level.*

Arrays MBA and RBA have the same number of elements K. Values of RBA are in the range from 0 to the number of blocks in the memory M_{mem} i.e., $0 \le x_i \le M_{mem}$ for $0 \le i < K$. Values of MBA are in the range from 0 to the number of sets in the set associative cache S, that is $0 \le y_i < S$ for $0 \le i < K$.

The whole associativity problem can now have a different formulation by analyzing the properties of RBA and MBA. Definition 5 presents some properties of MBA.

Definition 5 (Set of MBA destinations). *Let there be a MBA obtained by applying modulo functions r of corresponding RBA blocks for a set associative cache with S sets. The set of MBA destinations R is the set of d different elements of MBA, such that*

$$R = \{r_i, 0 \le i < d\} \quad \text{where}$$
$$0 \le r_i < S, \text{ and } r_i \ne r_j \text{ for } 0 \le i,j < d, i \ne j.$$

By analyzing the properties of MBA we can conclude that there are two extremes while mapping the blocks. The first extreme is found by a uniform distribution of elements, i.e. cache blocks are uniformly mapped in all cache sets. In this case there is no associativity problem and only cache capacity problem may appear.

In reality, modulo functions map cache blocks in a smaller number of sets than the available number of sets. A simple conclusion follows that the problems will start if $d < S$, i.e. when smaller number of cache blocks are used than the number of available blocks. The second analyzed extreme where highest associativity problems will appear is when $d = 1$, since only one set will be used instead of whole cache.

However, conflict misses do not appear in all cases when MBA blocks are distributed in smaller number of sets. The associativity value, shows that there are n blocks available in each set, so we also have to analyze this capacity by analyzing the problem "how many blocks will map in a particular set". If this number is greater than the associativity value, than the conflict cache misses will start to generate.

ANALYSIS OF CONFLICT MISSES

The following analysis shows that the distribution pattern of requested data elements is mostly generated by nested loops which represent recursions and iterations. It means that most of these cases actually use FOR NEXT (or WHILE DO, or REPEAT UNTIL) loops, which imply regular pattern in the algorithms, that usually refer to sequentially ordered blocks. Each algorithm request defines a certain offset f in this sequentially ordered list, so the mapping modulo function can be rather easily calculated.

Assume that the first instruction requests a data element from a block with location x. It is mapped onto a set identified with $y = x \ MOD \ S$. The next request according to the algorithm is for a block found on offset f, meaning that the next requested block is located on $x + f$ and the mapped set is $(x + f) \ MOD \ S$. A conflict might appear in cases when they map onto a same set, i.e. if $f \equiv 0 \ MOD \ S$.

There might be a case when a multiple number of f is equal to the number of sets S. In more general case there is a positive integer number that determines multiple factor of f that equals the number of sets. The following analysis presents the value of this integer.

Theorem 1. *If an algorithm specifies an offset of f blocks in the RBA of blocks such that*

$$d \cdot f = S \tag{4}$$

then there are exactly d different positions where the RBA blocks will be mapped, i.e. the number of different elements in MBA is d.

Proof. *Proof can be constructed using the properties of the modulo functions. Let the RBA be*

$$x, x + f, x + 2 \cdot f, \ldots$$

We can construct MBA with elements

$$x \ MOD \ S, (x + f) \ MOD \ S, (x + 2f) \ MOD \ S, \ldots$$

Since $d \cdot f = S$, it follows that $(x + d \cdot f) \ MOD \ S = x \ MOD \ S$, and the following array elements start again to follow the same pattern. We can identify that the different values of reminders is d.

If $d = 1$ then $f = S$ and there is exactly one set where the blocks will be mapped and conflict cache misses will start to appear after n iterations of the algorithm.

If $d > 1$ then $f \cdot d = S$ and the conflict cache misses will start to appear after $d \cdot n$ iterations of the algorithm. The following analysis gives the necessary condition for appearance of conflict cache misses.

Theorem 2. *Conflict cache misses will appear if the associativity is smaller than the number of blocks mapped onto a given set in a set-associative cache memory, if there are d diffe-rent sets where the blocks will be mapped, i.e.*

$$n < \frac{K}{d} \tag{5}$$

Proof 2. *According to Definition 3 the number of RBA blocks is K. Therefore, at least $\left\lceil \frac{K}{d} \right\rceil$ blocks will be mapped onto one set. Relation (5) is derived according to Dirichlet box principle that there are at most n positions, where these blocks will be placed.*

Note that we have analyzed this phenomenon from a different aspect in [8] and derived the same result from a very different perspective. The following example illustrates the RBA of blocks and results obtained in theorems 1 and 2.

Example 3 (Conflict cache misses for MMA with $N = 512$). *Analyze the MMA algorithm as presented in Example 2 for $N = 512$ and its execution on the AMD Opteron processor described in Example 1. There are 8 data elements in each cache block, and*

S_{Li} = 512. *We also suppose that the first matrix column element is found in a block with location x.*

The processor tries to identify each requested block in the L1 cache and maps it onto a specific set, determined by the mapping function expressed in Definition 1. Each block will be mapped to a set location

$$y(b_{ij}) = x(b_{ij}) \; MOD \; S_{L1} = (MA(b_{ij}) DIV \; 8) \; MOD \; S_{L1}$$

in the L1 cache.

The RBA of blocks, as presented in (3), will be mapped onto L1 cache set locations and form a MBA $y_0, y_1, y_2, \ldots$, *equal to* $y(b_{00}), y(b_{10}), y(b_{20}), \ldots$

We can also calculate that $y(b_{i0}) = (x + 64 \cdot i)$ *MOD* 512. *As an illustration, if* $x \equiv 0$ *MOD* 512, *then the MBA will be*

0, 64, 128, 192, 256, 320, 384, 448, 0, 64, 128, …

According to Definition 3, it follows that $f = x(b_{(i+1)0}) \cdot x(b_{(i0)}) = 64$.

We can calculate from Theorem 1 that $d = 8$ *satisfies (4) and it follows that the blocks will be mapped in exactly 8 sets. According to Definition 3, there are* $N = 512$ *iterations and* $N = K = 512$ *blocks in RBA for each iteration. Theorem 2 shows that there are* $\frac{N}{d} = 64$ *blocks to be placed onto one set. Since the associativity of L1 cache is* $n_{L1} = 2$ *there will be at least* 64 – 2 = 62 *conflict misses for each iteration accessing a column matrix, reaching* 62/64 = 96.87% *conflict misses.*

The number of sets for both L2 and L3 caches is 1024, and it follows that $d = \frac{s}{f} = 16 \cdot$

The number of blocks to be placed onto one set is $\frac{K}{d} = \frac{512}{16} = 32$ *The associativity for L2 is* $n_{L2} = 8$ *and for L3 is* $n_{L3} = 32$. *It means that conflict misses will appear for L2 and for L3 cache it will reach the boundary condition. There will be* 32 – 8 = 24 *conflict misses for L2 out of 32 accesses, reaching* 24/32 = 75% *conflict misses. Since the matrix A will occupy some blocks that are to be scheduled within previous analysis for matrix B column, it means that there will be at least some conflict misses in the L3 cache although 32 blocks are to be scheduled in 32 available places*.

The previous analysis assumed that multiple number of *f* equals to *S*, or when *f* is divisor of *S*. However, a more general case will be when *f* and *S* have a common divisor, or $f > S$. The following analysis presents this case.

Theorem 3 *If an algorithm specifies an offset of f blocks in the RBA of blocks such that the greatest common divisor of f and S is a positive integer* $g = GCD(f, S) > 1$, *then*

$$d \cdot f = k \cdot S, \text{ where } d = \frac{s}{\mathrm{g}} \text{ and } k = \frac{f}{g} \qquad (6)$$

There are exactly d different sets in MBA (where the RBA blocks will be mapped).

Proof. Let us a*nalyze the RBA of blocks*

$$x, x + f, x + 2 \cdot f, \ldots$$

and construct MBA with the elements

$$x \; MOD \; S, (x + f) \; MOD \; S, (x + 2 \cdot f) \; MOD \; S, \ldots$$

Since $d \cdot f = k \cdot S$, *it follows that*

$$(x + d \cdot f) \; MOD \; S = x \; MOD \; S \qquad (7)$$

and the following array elements start again to follow the same pattern. Due to the definition of d as the smallest number such that (7) holds and the number of different reminders modulo S is exactly d.

Since Theorem 3 proves that the number of different sets is *d*, we can also confirm the eligibility of Theorem 2. The following example presents the analyzed more general case.

Example 4 (Conflict cache misses for MMA with $N = 768$**).** *We assume that the MMA algorithm presented in Example 2 is executed on the AMD Opteron processor described in Example 1 for* $N = 768$. *There are* $K = N = 768$ *iterations and* $K = N = 768$ *elements of RBA for each iteration. In this case the block address offset is* $f = x(b_{(i+1)0}) - x(b_{i0}) = 96$ *Theorem 3 can be applied, since f is not a divisor of S.*

The corresponding RBA of blocks $x(b_{00}), x(b_{10}), x(b_{20}), \ldots$ *is equal to*

$$x, x + 96, x + 192, x + 288, x + 384, \ldots$$

As an illustration, if $x \equiv 0$ *MOD* 512, *then MBA* $y(b_{00}), y(b_{10}), y(b_{20}), \ldots$ *is*

0, 96, 192, 288, 384, 480, 576, 768, 864, 960, 1056, 1152, 1248, 1344, 1440, 0, 96, …

Let us analyze properties of MBA. According to Theorem 3, $g = GCD(f, S) = 32$. *Then we calculate* $d = 16$, *and it satisfies (6). It follows that the blocks will be mapped in exactly 16 sets. According to Definition 3 there are* $N = 768$ *iterations and blocks in both RBA and MBA for each iteration. Theorem 2 shows that there are*

$\frac{K}{d} = 48$ *blocks to be placed onto one set. Since the associativity is* $n_{L1} = 2$ *the number of conflict misses is* 46 *out of* 48 *accesses, reaching* 46/48 = 95.83% *cache misses.*

The number of sets for both L2 and L3 is 1024, it follows that $d = \frac{s}{g} = 32$. *The number of blocks to be placed onto one set is* $\frac{K}{d} = \frac{768}{32} = 24$. *The associativity for L2 is* $n_{L2} = 8$ *and for L3 is* $n_{L3} = 32$, *which means that conflict misses will appear for L2, but not for L3 cache. There will be* 24 – 8 = 16 *conflict misses for L2 out of 24 accesses, reaching 66.67% conflict misses.*

RESULTS AND DISCUSSION

In this section we conduct experiments to prove the theoretical analysis explained in previous section and explain the conflict cache misses. The experiments are conducted using the Opteron processor [10], whose characteristics are explained in Example 1. The testing algorithm is matrix multiplication, as presented in Example 2.

The diagrams on figures 2, 3 and 4 present the area around these points, (five points –40, –32, –24, –16 and –8 bellow the expected negative impulses and five points +8, +16, +24, +32 and +40 above the matrix size where performance drop is expected). The x-axis presents the matrix size N in each figure in this section and the y-axis depicts the measured speed in MFLOPS.

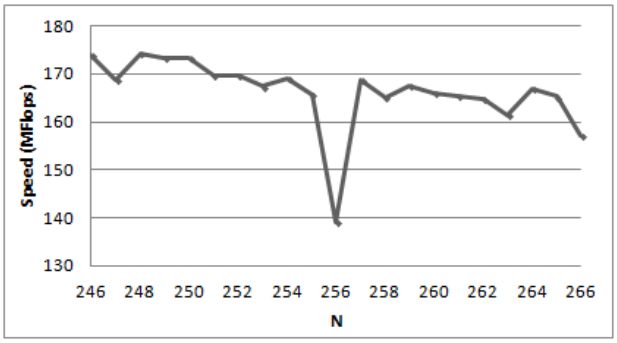

Figure 2: Speeds in the area around $N = 256$

Firstly, we shall illustrate the case where the capacity misses appear, as discussed in Lemma 1. In Example 1 we have presented that L2 cache has associativity $n_{L2} = 8$, and the number of sets is $S_{L2} = 1024$, meaning that the number of cache blocks is $M_{L2} = 8192$. The cache block size fits 8 data elements and there is possibility to store a total of 65536 data elements.

Suppose that there is uniform distribution of data elements of one matrix in all sets in the L2 cache. It means that $N \cdot N$ data elements will fit in the available number until the capacity problems appear. In this case, we calculate that $N = 256$. If $N > 256$, then the capacity misses will start to raise, as presented in Figure 2. This is why the measured speed has decreasing trend for $N > 256$, while for < 256 it has almost horizontal stable value. Note that the capacity misses will start earlier (for smaller N), since the algorithm needs two matrices to be stored in the cache, but the nature of the algorithm is to use a row of matrix A and the whole matrix of B to fulfill each iteration column by column. However, expressive cache capacity misses will appear for specified N, since the column matrix elements will initiate conflict misses with some of the elements of A.

The peak in $N = 256$ appears due to associativity problems. We calculate that each column data element, as requested by the algorithm, is on memory address offset $N = K = 256$ and $f = \frac{N}{8} = 32$, since there are 8 data elements in each cache block. Analyzing L1 cache with $S_{L1} = 512$ sets we obtain $d = 16$, according to Theorem 1. The number of sets in L2 and L3 is 1024 and we obtain that $d = 32$ in these cases.

The existence of associativity misses is shown by Theorem 2. The values for associativity are $n_{L1} = 2$, $n_{L2} = 8$ and $n_{L3} = 32$. Since $N = K = 256$ per each iteration, we calculate that $\frac{K}{d} = \frac{256}{16} = 16$ for L1 cache, and $\frac{K}{d} = \frac{256}{32} = 8$ for L2 and L3 caches. It appears that the conditions in Theorem 2 are satisfied for L1 and L2 caches, but not for L3 cache, i.e. associativity problems appear only in L1 and L2 caches.

The number of conflict misses in L1 cache is $\frac{K}{d} - n_{L1} = 16 - 2 = 14$, meaning that 14 cache misses will appear in 16 accesses, i.e. 14/16 = 87.5% of data accesses will be conflict misses. The analysis for L2 shows that 8 blocks have to be placed in 8 available places. In the previous analysis we assumed only the distribution of the B matrix. However, matrix elements of A occupy at least one set and conflict misses appear even earlier. Additionally, L1 cache misses initiate additional cache misses in the L2 cache.

We continue to present details of those matrix sizes analyzed in Examples 3 and 4, correspondingly for $N = 512$ and $N = 768$.

The results for Example 3 present the area of $N = 512$ and Figure 3 depicts the achieved speed. Performance drop for $N = 512$ is evident in comparison to the neighboring points. The speed has a negative trend, as shown for the previous analysis of $N = 512$ due to increased capacity misses.

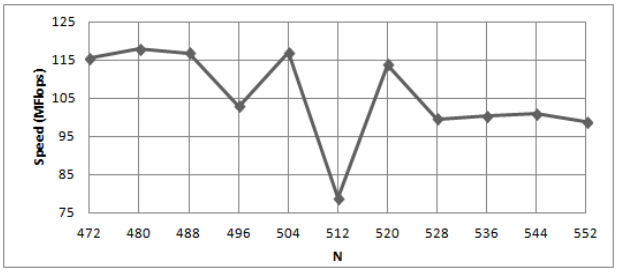

Figure 3: Speeds in the area around $N = 512$

Figure 4 depicts the achieved speed in the area of $N = 768$, theoretically analyzed in Example 4.

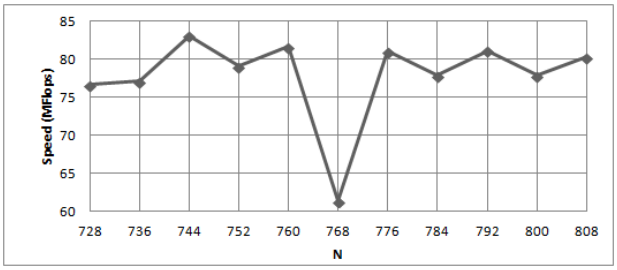

Figure 4: Speeds in the area around $N = 768$

Similarly to the previous case, the theoretical analysis is proved and the speed has decreased in the analyzed point.

CONCLUSIONS

In this paper we present an analysis of associativity and conflict misses in set associative caches. The analysis is performed by analyzing storage patterns and mapping functions used in caches to store blocks.

We define RBA, as an array of blocks that is generated by accesses defined in the algorithm. This array is mapped via modulo functions onto MBA, as array of block locations in the cache. This mapping is dependent on cache hardware characteristics and therefore we analyze relations among data arrays used in algorithms and hardware cache parameters.

Analysis of properties of the MBA using congruences and modulo functions shows that specific problems with RBA with K blocks and algorithm property of address offset f can correlate with the number of sets S in the set associative cache. The relation shown in Theorem 1 explicitly determines the number of used blocks in the set associative cache instead of using all available blocks. This phenomenon generates conflict misses and initiates performance degradation if conditions defined in Theorem 2 are fulfilled. This is illustrated by an example, that covers a case study for $N = 512$ and later on by experimental proof.

Theorem 1 covers only cases when the matrix size is a divisor of the number of sets in set associative cache. However, we also analyzed cases when the matrix size and number of sets might have a common divisor. Theorem 3 covers this case, as a general approach of the associativity problem. Therefore, we illustrate a theoretical explanation in the example with case study for $N = 768$ corresponding experimental proof.

These results can be used in specifying algorithms, by providing a careful analysis of associativity and conflict cache misses for a given data array size and storage pattern used in a repetitive algorithm. If the analysis shows that the associativity will initiate conflict misses, we can increase the data array or change the algorithm pattern to avoid conflict misses, and yet effectively use the cache exploiting time and data locality properties.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, Fifth Edition: A Quantitative Approach. MA, USA: Elsevier, 2012.

[2] M. Gusev and S. Ristov, A superlinear speedup region for matrix multiplication, *Concurrency and Computation: Practice and Experience*, 2013.

[3] G. Tsilikas and M. Fleury, Matrix multiplication performance on commodity sharedmemory multiprocessors, in *International Conference on Parallel Computing in Electrical Engineering, PARELEC 2004*, Sept. 2004, pp. 13 – 18.

[4] S. Sen, S. Chatterjee, and N. Dumir, Towards a theory of cache-efficient algorithms, *Journal of the ACM (JACM)*, **49** 2002 pp. 828–858.

[5] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, *Parallel Computing*, **35** 2009 pp. 178–194.

[6] H. Yoon, T. Zhang, and M. H. Lipasti, Sip: Speculative insertion policy for high performance

caching, Computer Sciences Department University of Wisconsin-Madison, Tech. Rep. 1676, 2010.

[7] X. Ding, K. Wang, and X. Zhang, Ulcc: a user-level facility for optimizing shared cache performance on multicores, in *Proceedings of the 16th ACM Symposium on Principles and practice of parallel programming,* ser. PPoPP '11. ACM, 2011, pp. 103–112.

[8] M. Gusev and S. Ristov, Performance gains and drawbacks using set associative cache, *Journal of Next Generation Information Technology (JNIT)*, **3**, 31 Aug 2012,pp. 87–98.

[9] L. Djinevski, S. Arsenovski, S. Ristov, and M. Gusev, Performance drawbacks for matrix multiplication using set associative cache in gpu devices, in *MIPRO, 2013 Proceedings of the 36th International Convention, IEEE Conference Publications*, Croatia, 2013, pp. 213–218.

[10] CPU-world. (2013, Sep.) AMD Opteron(tm) 8347 @ONLINE. [Online]. Available: http://www.cpu-world.com/CPUs/K10.

АНАЛИЗА НА ПРОМАШУВАЊА ПОРАДИ АСОЦИЈАТИВНОСТ И КОНФЛИКТ

Марјан Гушев, Сашко Ристов

Факултет за информатички науки и компјутерско инженерство,
Универзитет „Св. Кирил и Методиј“, Скопје, Република Македонија

Кеш-меморијата игра голема улога во определувањето на перформансите при решавање на научни проблеми. Многу од овие проблеми вклучуваат голем број на повторувања на сложени или едноставни пресметки врз различни податочни елементи складирани како низи во секвенцијален редослед во меморијата. При извршување на алгоритмите, овие елементи се преземаат во кеш-меморијата и тогаш се користат од процесорот. Овој процес вообичаено е проследен со промашувања во кеш-меморијата поради конфликти или капацитет, а перформансите се намалени од функцијата за сместување во кеш-меморијата, политиката на замена во кеш-меморијата или од ограничувањето на капацитетот.

Во овој труд го анализираме влијанието на алгоритмите и перформансите од сет-асоцијативната кеш-меморија кога е референцирана голема низа во секвенцијално подредената меморија. Пресликан е проблемот на користењето на кеш-меморијата во математички модел базиран на ИТ за да се анализираат перформансите и да се даде научно објаснување за падовите на перформансите поради промашувањата во кеш-меморијата предизвикани од асоцијативноста и конфликтот.

Клучни зборови: Пресметување со високи перформанси; пад на перформанси; мулти-процесор со споделена меморија; супер-линеарно забрзување