Skewed associativity improves program performance and enhances predictability*†

François Bodin, André Seznec
IRISA-INRIA, Campus de Beaulieu
35042 Rennes Cedex, FRANCE
e-mail : bodin,seznec@irisa.fr

Copyright Notice

*A shorter version of this study appeared in the Proceedings of the 22th International Symposium on Computer Architecture (June 1995)

†This work was partially supported by Esprit project BRA Apparc and by CNRS (GDR-ANM)

Abstract

Performance tuning becomes harder as computer technology advances. One of the factors is the increasing complexity of memory hierarchies. Most modern machines now use at least one level of cache memory. To reduce execution stalls, cache misses must be very low. Software techniques used to improve locality have been developped for numerical codes, such as loop blocking and copying. Unfortunately, the behavior of direct mapped and set associative caches is still erratic when large data arrays are accessed. Execution time can vary drasticly for the same loop kernel depending on uncontrolled factors such as array leading size. The only software method available to improve execution time stability is the copying of frequently used data, which is costly in execution time. Users are not usually cache organisation experts. They are not aware of such phenomena, and have no control over it.

In this paper, we show that the recently proposed 4-way skewed associative cache yields very stable execution times and good average miss ratios on blocked algorithms. As a result, execution time is faster and much more predictable than with conventional caches. It is therefore possible to use larger block sizes in blocked algorithms, which will further reduce blocking overhead costs.

Keywords:

cache, predictable performance, numeric kernels, loop blocking, skewed-associative caches

1 Introduction

Performance tuning has become very complex on today's computers. One of the factors of this complexity is the use of memory hierarchies, and particularly of cache memories. As the miss penalty becomes higher and higher, cache performance becomes a bottleneck. Unfortunately, the behaviors of direct-mapped and set-associative caches are very sensitive to small application parameters variations. Since caches are not perfect (limited associativity, non-optimal replacement strategy), performance may suffer unpredictably from conflict misses even on blocked loops [6, 12].

Such unpredictable behaviors can be extreme as illustrated in a recent study by Schlansker et al [8]. They showed that, even with very regular memory access patterns such as sequential access to fixed size memory subblocks, the miss ratio on a 32-way set-associative cache depends heavily on parameters such as the number of rows of the whole matrix. In their example, depending on whether the number of rows is 2727 or 2729, nearly all the accesses hit in the cache or nearly all of them miss.

For most users, getting a predictable and stable performance is a major issue. The previous example clearly illustrates that using set-associative caches does not suffice.

Recently, the skewed-associative cache, a new associative cache structure has been proposed in [9, 10]. In this paper, we investigate the sensitivity of skewed-associative caches to parameters such as array sizes or relative array placements in numerical kernels on dense structures.

The main contribution of this article is to show that, unlike usual set-associative caches, a 4-way skewed-associative cache is quite insensitive to these application parameters. As, a result better average miss ratio is achieved by skewed-associative than set-associative cache and performance becomes much more predictable.

When using direct-mapped caches or set-associative caches, copying is usually the only viable software solution to avoid unpredictable catastrophic behaviors for some array dimensions; when using a 4-way skewed associative cache, blocking is sufficient, thus extra array copying costs are avoided.

Our simulations also established that, even when using restructuring technique such as blocking and copying, performance of direct-mapped caches is significantly worse than performance of set and skewed associative caches. Moreover our experiments show that even when blocking and copying is used, the execution time may vary within a large range for direct-mapped and set-associative caches, depending on relative array placements.

On usual direct-mapped or set-associative caches, the behavior of caches on blocked algorithms degrades very rapidly when the blocking factor increases. The second major contribution of this article is to show that, when using a 4-way skewed-associative cache, a larger fraction of the cache may be used for blocking than on conventional caches.

The remainder of the paper is organized as follows. Related work is described in Section 2. In Section 3, the principles of a skewed-associative cache are presented. In Section 4 we present a very simple experiment which explains why skewed-associative cache should exhibit a better average behavior than a standard set-associative cache. In Section 5, we present the simulation tools which have been used in the paper. In Section 6, the impact of various array placements is studied on a few numerical kernels. Original, blocked and block-copied algorithms are studied. In Section 7, we study the impact of the blocking factor on performance and we try to characterize the fraction of the cache size that is available for blocking (resp. blocking & copying) on set-associative and skewed-associative caches. Section 8 summarizes this study.

2 Related work

Improving performance by reducing capacity and conflict misses in numerical applications by software technique has been addressed in many studies. The first studies [4, 13, 14, 7, 3] focused on limiting the size of the current working set of the applications, thus reducing the number of capacity misses on the cache. Blocking or anyother unimodular transformations can be used at compile time to enhance spatial and temporal locality in applications.

But blocking is not a sufficient technique for many applications and many cache configurations. In order to overcome this difficulty, blocks exhibiting high level of reuse may be copied in order to control data placement and avoid placement conflicts in the cache [12, 4, 6]. However copying may induce a large computing overhead on many numerical kernels. Techniques for determining whether copying is needed or not address direct-mapped caches and are still very conservative(e.g. [12, 6]). These techniques would have to be applied at run-time when the sizes and addresses of arrays are unknown at compile time (e.g. calls to library routines).

In order to avoid unpredictable and catastrophic behavior of caches without the cost of copying, Schlansker et al [8] proposed to use a complex hashing function for the set selection in order to obtain a good and predictable behavior. Their proposal suffers from two major drawbacks: a high degree of associativity is needed (in the 16-32 range) and complex hardware mechanism is needed to pseudo-randomize the set selection in the cache.

3 Skewed-associative caches

3.1 Principle

Skewed associative caches have been recently proposed in [9, 10]. A X-way set-associative cache is built with X distinct banks as illustrated in Figure 1 . The memory block at address D may be physically mapped onto physical line $f(D)$ of any of the distinct banks. This vision of a set-associative cache fits with the physical implementation: X banks of static RAMs.

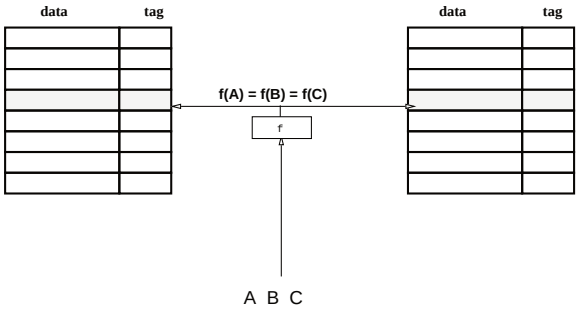


Figure 1: 3 data blocks conflicting for a single set on a two-way set-associative cache. A, B and C compete for only two locations

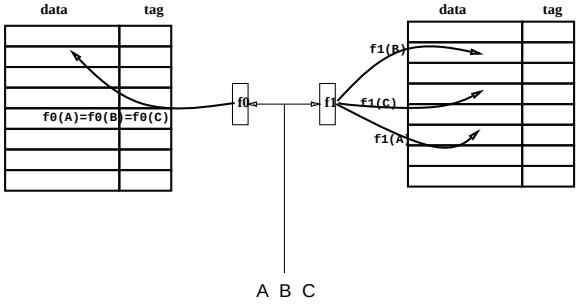


Figure 2: A, B and C compete for the same location in bank 0, but can be present at the same time, as they do not map to the same location in bank 1

For a skewed associative cache (Figure 2), a different mapping function is used for each cache bank: A memory block at address D may be mapped onto physical line $f_0(D)$ in bank 0, onto physical line $f_1(D)$ in bank 1, etc.

It has been shown in [9, 10] that for general applications skewed-associative caches exhibit an average lower miss ratio than set-associative caches.

3.2 Choosing the skewing functions

In this section, we give some insight on the properties that might exhibit functions chosen for skewing the blocks in the distinct cache banks in order to obtain a good hit ratio.

First we give some notations used:

- L is the number of lines in the cache
- X is the number of cache banks.
- S is the set of memory blocks in the main memory. $\|S\|$ is the cardinal of S.
- CL_i is the set of lines in cache bank i.
- D is a memory block. For sake of simplicity, we shall also note D the address of the first byte in the block.
- f_i is the mapping function from S to CL_i.

3.2.1 Equitability

First of all like in classical caches, for each line in the cache, the numbers of memory blocks that may be mapped onto this cache line must be equal. In more mathematical terms a mapping function is equitable when:

$$\forall\ C\ \in\ CL_i, \|f_i^{-1}(C)\| = \frac{\|S\|}{\|CL_i\|}$$

3.2.2 Inter-bank dispersion

In a usual X-way set-associative cache, when (X+1) memory blocks contend for the same set in the cacheD: one of the blocks must be rejected from the cache (Figure 1), since only X lines are available.

Skewed-associative caches avoid this situation by scattering the data: mapping functions can be chosen so that whenever two memory blocks conflict for a same line in bank i, they have a very low probability to conflict for a location in bank j (Figure 2).

Ideally, mapping functions should be chosen such that the set of blocks that can map onto a cache line of bank i will be equaly distributed over all the lines in the other banks.

In more mathematical terms , complete inter-bank dispersion is achieved when:

$$\forall\ D\ \in S,\ \forall\ i \neq j\ ,\ \|\{d\ \in\ S / f_i(D) = f_i(d)\ \&\ f_j(D) = f_j(d)\}\| = \frac{\|S\|}{\|CL_i\|^2}$$

3.2.3 Local dispersion in a single bank

Many applications exhibit spatial locality, therefore the mapping functions must also be chosen so that two "almost" neighbor memory blocks do not contend for the same physical line in any cache bank.

The different mapping functions must respect a certain form of local dispersion on their respective bank; the mapping functions f_i must limit the number of conflicts when mapping any region of consecutive memory blocks within bank i.

3.2.4 Simple hardware implementation

A key issue for the overall performance of a microprocessor is the pipeline length. Using different mapping functions for each cache bank will have no effects on the cycle time, as long as the computations of the mapping functions can be performed in a non critical stage of the pipeline and do not lengthen the pipeline cycle. Let us notice that in most of the new generation microprocessors, the address computation stage is not the critical stage in the pipeline (e.g. in TI SuperSparc, two cascaded ALU operations may be executed in a single cycle).

In order to achieve this, we have to choose mapping functions which have the simplest implementation, so as to introduce very few extra gates and delays.

3.3 An example of skewing functions

We present here the skewing functions which were used in the simulations that illustrate this paper.

Let us consider a skewed associative cache built with 2 or 4 cache banks, each one consisting of 2^n cache lines of 2^c bytes,
let σ be the perfect-shuffle on n bits [11][1], data block at memory address $A_3 2^{c+2n} + A_2 2^{n+c} + A_1 2^c$ may be mapped:

1. on cache line $f_0(A) = A_1 \oplus A_2$ in cache bank 0
2. *or* on cache line $f_1(A) = \sigma(A_1) \oplus A_2$ in cache bank 1
3. *or* on cache line $f_2(A) = \sigma^2(A_1) \oplus A_2$ in cache bank 2
4. *or* on cache line $f_3(A) = \sigma^3(A_1) \oplus A_2$ in cache bank 3

These functions were chosen for the following reasons:

1. They satisfy the previously listed criterions for "good" skewing functions. Eeach bit in $f_i(A)$ is obtained with a single two-entry XOR. Complete inter-bank dispersion is not achieved; nevertheless, the set of blocks which may be mapped on a precise cache line in one bank is spread over a large number of lines in the other banks.

   Since two blocks with the same highest order bits (A3 and A2) cannot be mapped on the same cache line by function f_i, local dispersion is achieved.

2. They are relatively simple to compute : this limits the simulation time and represent a potential cheap implementation.

The results illustated in this paper are not specific to this particular choice of skewing functions: the bit permutations used on A2 might be different, the global behavior would be the same. Other skewing functions satisfying the listed criterions were used in [9, 10].

3.4 Which replacement policy for skewed-associative cache

When a miss occurs in a X-bank caches, the memory block to be replaced must be chosen among X blocks. Different replacement policies may be used. LRU replacement policy or pseudo-random replacement policy are generally used on set-associative caches.

LRU replacement policy is generally considered as the most efficient policy. Implementing a LRU replacement policy on a two-way set-associative cache is quite simple. A single bit tag per cache line is sufficient: when a line is accessed, this tag is asserted and the tag of the second line of the set is deasserted. More generally a LRU replacement policy for a X-way set-associative cache can be implemented by adding only X bit tags to each line.

Unfortunately, we have not been able to find a simple hardware implementation of a LRU replacement policy on a skewed-associative cache: as the set of lines on which a block has to be replaced vary with the new block to be introduced, the information needed in order to determine the last referenced line in the set is the complete date of the reference.

Using a pseudo-random replacement policy generally induces slightly more misses on caches than using a LRU replacement policy. We propose here a very simple replacement policy which requires only one tag bit per cache line and for which we experimentally obtained a good behavior [10]:

- The bit tag RU (Recently Used) is asserted when the cache line is accessed
- Periodically the bit tags RU of all the cache lines are reset: we experimentally determine that a good period is each $\frac{\text{cache size in bytes}}{4}$ accesses to the cache.

When a block misses in the cache, the replaced block is chosen among the X possible blocks in the following priority order:

[1] The perfect-shuffle on a n-bit number is the one-position circular shift on a n-bit number

1. Randomly among the blocks for which the RU tag is clear
2. Randomly among the blocks for which the RU tag is set, but which have not been modified since they have been loaded in the cache
3. Randomly among the blocks for which the RU tag is asserted and which have been modified.

This replacement policy is quite simple to implement in hardware. An interesting property of this replacement policy is to limit the copy back of data to memory (or the L2 cache) and therefore limiting memory traffic.

We call this replacement policy: Not Recently Used Not Recently Written (NRUNRW).

4 How a skewed-associative cache handles conflict misses

A very simple experiment was conducted in order to illustrate the benefits that can be expected from a skewed-associative cache.

Let us consider a 512 lines cache. Let us consider a collection of X data blocks each with a random address. This collection is iteratively read 10 times. Direct-mapped, 2, 4, 8, 16 and 32-way set-associative, 2 and 4-way skewed-associative were simulated. The experiment was repeated on 100 different collections for every collection size. The cache is empty when starting the experiment.

4.1 Data dispersion

Figure 3 illustrates the average ratio of blocks that remain valid in the cache after a single read pass of the whole collection for collection sizes varying from 32 to 512. The number of valid blocks in the 2-way skewed-associative cache is greater in the 2-way set-associative cache and slightly less than in the 4-way set-associative cache.

The number of valid blocks in the 4-way skewed-associative cache is approximately equal to that of an 8-way set-associative cache, but is less than for 16-way and 32-way set-associative caches.

After a single read sequence, for the same associativity degree, more data will be retained by a skewed-associative cache than by a set-associative cache.

4.2 Self data reorganization

A second phenomenon, we call *self data reorganization* accentuates the advantage of the skewed-associative over the set-associative cache.

Figure 4 illustrates the average ratio of the blocks in the sequence that remain valid in the cache after ten successive reads of the whole sequence.

For direct-mapped and set-associative caches, the number of valid blocks does not evolve after the first read sequence: if a block is missing then its set is full. Loading it will invalidate another block in that set.

However, in the skewed-associative cache, the number of data blocks present at the same time in the cache depends on the precise mapping of each data block in the cache. Let us consider a memory block D present in the cache at time t. Among the other possible locations for a block D, there may be an empty location. Block D may be removed from the cache by a miss on an other block D'. The next time D will be referenced, D can be mapped in an empty location in bank j and thus the number of data alive at the same time in the cache will increase.

For instance, after ten iterations of the read sequence, the number of valid blocks in the 4-way skewed-associative cache (resp. 2-way) is in the range of the 32-way set-associative cache (resp. 8-way).

In blocked algorithms, block sizes are chosen in such a way that the size of the reused data is smaller than the cache size. It may be expected that the self data reorganization in the skewed-associative cache will limit conflict misses on such blocked algorithms and allow larger blocking factors to be used.

Notice that this example does not show sure performance gains. Set distribution is not random in real applications. In many cases, due to spatial locality set-associative caches work really well, but desastrous set distribution may also be seen as it was shown in [6, 8] and as it will be emphasised in the next sections.

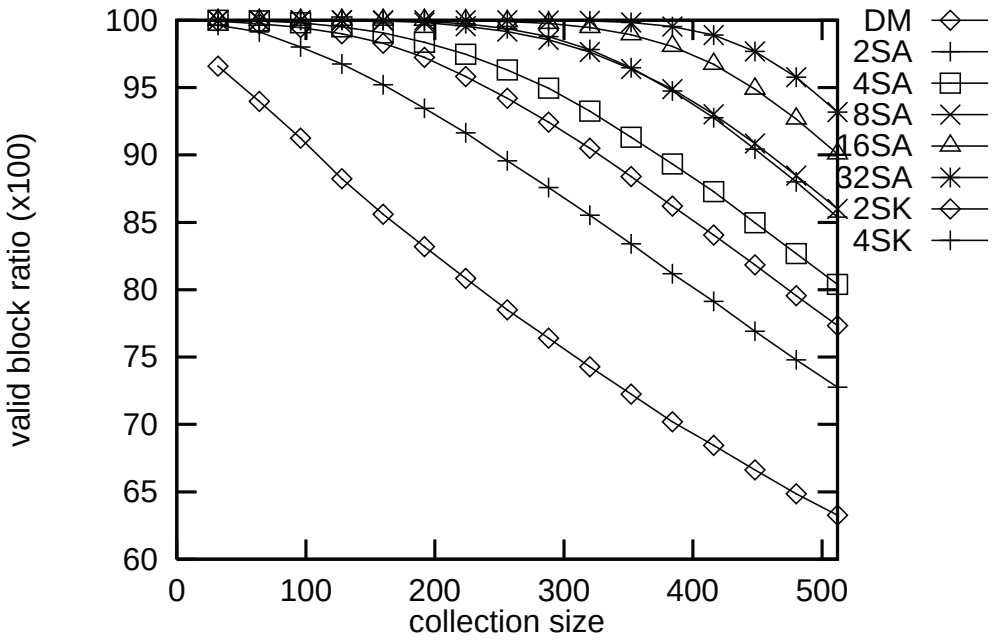


Figure 3: Ratio of valid blocks after one read

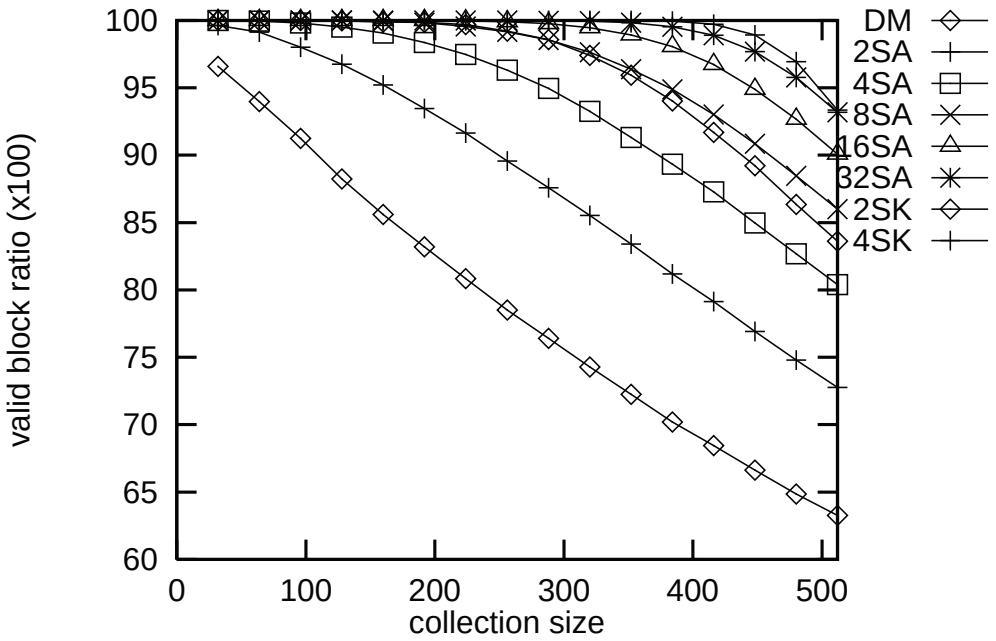


Figure 4: Ratio of valid blocks after ten reads

5 Evaluation methodology

In order to capture effective program behavior including loop management and scalar references, we chose to use real program execution traces.

Trace generation: The *Spa* package developed by Gordon Irlam [5] was used to generate address traces for programs executed on a SUN SparcStation10. F77 Fortran compiler with -O4 -dalign optimizations was used.

No modification of the source or of the binary code to be analyzed was required. User code of a single application can be completely traced except for the OS kernel code. Since we studied the behavior of numerical kernels consisting of a few nested loops, only data references were piped to a cache simulator.

Simulated cache configurations Five cache organizations were simulated in a single path: direct-mapped, 2-way and 4-way set-associative, 2-way and 4-way skewed-associative caches.

In order to limit the sizes of the problems needed to exceed cache capacity (to limit simulation time), a 8Kbyte cache was simulated. The cache line size chosen for the simulation was 32 bytes.

All simulations were done assuming a write back write allocate cache.

Evaluation of the execution time In order to accurately estimate the overhead associated with blocking and copying on the execution time for the different algorithms, a superscalar processor was simulated:

- The simulated configuration was one branch unit, two integer units, one load/store unit and two floating-point units; a 2-bit branch target buffer was implemented.
- Up to four instructions were executed on each cycle. A synchronous in-order execution was assumed i.e. if an instruction cannot be executed due to a data or resource hazard then all subsequent instructions are delayed.
- An ideal cache was assumed (i.e every reference hits).

We call the execution time obtained from this simulation the *ideal execution time*.

In the remainder of paper, we assume a 10 cycle miss penalty. N_{miss} being the number of misses, the execution time of a kernel is simply modeled by:

$$T_{exec} = \textit{ ideal execution time } + 10 * N_{miss} \tag{1}$$

Measuring sensitivity to array placement In order to measure the sensitivity of the cache behavior to parameters such as array sizes or block sizes, systematic experiments were repeated while varying block size and/or leading size (i.e. number of rows in a matrix).

6 Performance predictability and array placement

The experiments presented in this section evaluate the cache performance for three loop kernels : 100×100 matrix-matrix multiply, 340×340 2D Jacobi loop and 100×100 LU factorization. In all the experiments, we change the leading dimension of the arrays used in the loops to highlight the impact of the array declaration on the cache behavior. We simulated original, blocked and copy blocked versions of the kernels.

The three original kernels were chosen because they exhibit different characteristics:

1. The matrix-matrix multiply is a the three-fold nested loop where each data is reused many times. Automatic blocking technique may be used on this kernel.
2. The 2D Jacobi loop is a two-folded nest loop where data reuse is limited. This loop can be automatically blocked. Due to limited data reuse, overhead associated with copying is very high.

3. The LU loop is a three-fold nested loop. Data reuse is very high. Deriving automatically a blocked version of this loop is not easy; the blocked and copy blocked versions of the LU loop used in the paper were derived by hand.

For these three kernels, we measured the impact of blocking and blocking & copying on the number of cache misses, on extra memory references and on extra instructions and execution times. For **blocked** and **copy blocked** versions of the applications, the block size was chosen so that the total size of the blocks is approximately equal to half of the cache size. As shown in Section 7, this happens to be a good approximation for skewed-associative caches.

6.1 Matrix-matrix multiply

The three versions of the programs which were simulated are illustrated in Figure 5. The number of floating-point references in the different versions of the algorithm is predictable from the Fortran code:

- in the original loop, each element of matrix A is invariant in the inner most loop and is read and written one time (20000 references) and each element of matrices B and C are read 100 times (2000000 references).
- when blocking, each element of matrix A is read and written five times instead of once, thus leading to 80000 extra memory accesses (see Figure 5).
- in the copy blocked version, each element of matrix C is copied one time (20000 references) and each element of matrix B is copied five times (100000 references), thus leading to a total of 200000 extra memory references over the original loop.

The number of instructions executed by the different versions depends highly on the quality of the F77 compiler. Statistics on the execution of the three simulated algorithms are reported in table 1. For our examples, the F77 compiler automatically unrolls the inner most loop 4 times. The large differences between ideal execution time of the original version of the algorithm and that of the blocked and blocked & copied algorithms are due to the unrolling: since the blocking factor used (23) is not a multiple of 4, three extra iterations in each inner most loop are executed unrolled , thus generating more instructions and disabling the parallel execution of the instructions.

	Original	Blocked	Bl & Co
number of floating point ref	2020000	2100000	2220000
number of data ref	2024880	2120751	2243408
number of instructions	5942009	7069993	7351430
ideal execution time	2093138	2711801	2807272

Table 1: Characteristics on the different matrix-multiply versions

The data reuse in the 100×100 matrix-matrix multiply is very high: each element of matrix B and C are used 100 times. Moreover each cache block contains 4 words, leading to 400 reads of each block. Such an optimal reuse can only be obtained when the whole matrices fit in the cache. For the blocked and blocked copied versions, a relatively correct estimation of the minimal number of misses is obtained by assuming a perfect cache but no reuse across the blocks: 27500 misses.

In Figure 6 and Figure 7, we respectively illustrates the number of misses and the execution times for each of the three versions run with numbers of rows of the arrays varying from 100 to 550.

Original loop and blocked loop: It clearly appears that direct-mapped and 2 and 4-way set-associative caches exhibit erratic behaviors on the original version as well as on the blocked version of the algorithm. For instance, the execution time of the blocked algorithm range from from 3 200 000 cycles to 16 000 000 cycles, even with a 4-way set-associative cache.

```
do i=1,100
  do j=1,100
    tmp=a(i,j)
    do k=1,100
      tmp=tmp+b(i,k)*c(k,j)
    enddo
    a(i,j)=tmp
  enddo
enddo
```

(a)

```
do j=1,100,23
  do k=1,100,23
    do i=1,100
      do JJ=0,min(100-j,22)
        tmp=a(i,j+JJ)
        do KK=0,min(100-k,22)
          tmp=tmp+b(i,k+KK) *c(k+KK,j+JJ)
        enddo
        a(i,j+JJ)=tmp
      enddo
    enddo
  enddo
enddo
```

(b)

```
do j=1,100,23
  do k=1,100,23
    do JJ=0,min(100-j,22)
      do KK=0,min(100-k,22)
         cc(JJ+23*KK)= c(k+KK,j+JJ)
      enddo
    enddo
    do i=1,100
      do KK=0,min(100-k,22)
         bb(KK)=b(i,k+KK)
      enddo
      do JJ=0,min(100-j,22)
         aa=a(i,j+JJ)
         do KK=0,min(100-k,22)
            aa=aa+bb(KK)* cc(JJ+23*KK)
         enddo
         a(i,j+JJ)=aa
      enddo
    enddo
  enddo
enddo
```

(c)

Figure 5: (a) Original matrix-matrix multiplication, (b) blocked matrix-matrix multiplication, (c) blocked matrix-matrix multiplication with copies

The number of misses on the 2-way skewed-associative cache is less irregular, and becomes quite regular on the 4-way skewed-associative cache. Notice that the average miss ratio is also lower on a 4-way skewed-associative cache than on anyother cache structure: for the blocked version, the average miss count is around 62 000 for the 4-way skewed-associative cache against 126 000 for the 4-way set-associative cache.

Blocked and Copied: Data used in the inner-most loops is copied in contiguous memory locations. Since the blocking factor has been chosen for blocked data to fit in half of the cache size, conflict misses are eliminated in the inner-most loop. Associating blocking and copying brings relatively stable numbers of misses for the matrix multiply.

Nevertheless, conflict misses may still occur during the copying phase between the arrays to be copied and the copied arrays. In that case, number of misses on the blocked & copied version is not completely independent of the array leading dimension (particularly for set-associative caches). This still has a serious impact on execution time: with a 4-way set-associative cache, the execution time rangess between 3 180 000 to 4 180 000 cycles.

We still notice that the 4-way skewed-associative cache has a lower miss ratio average and a more stable behavior than that of the other caches.

Note also that the direct-mapped cache displays a stable although poor performance.

For all cache organizations, except the 4-way skewed-associative cache, the average execution time is clearly better on the blocked & copied loop than on the blocked loop.

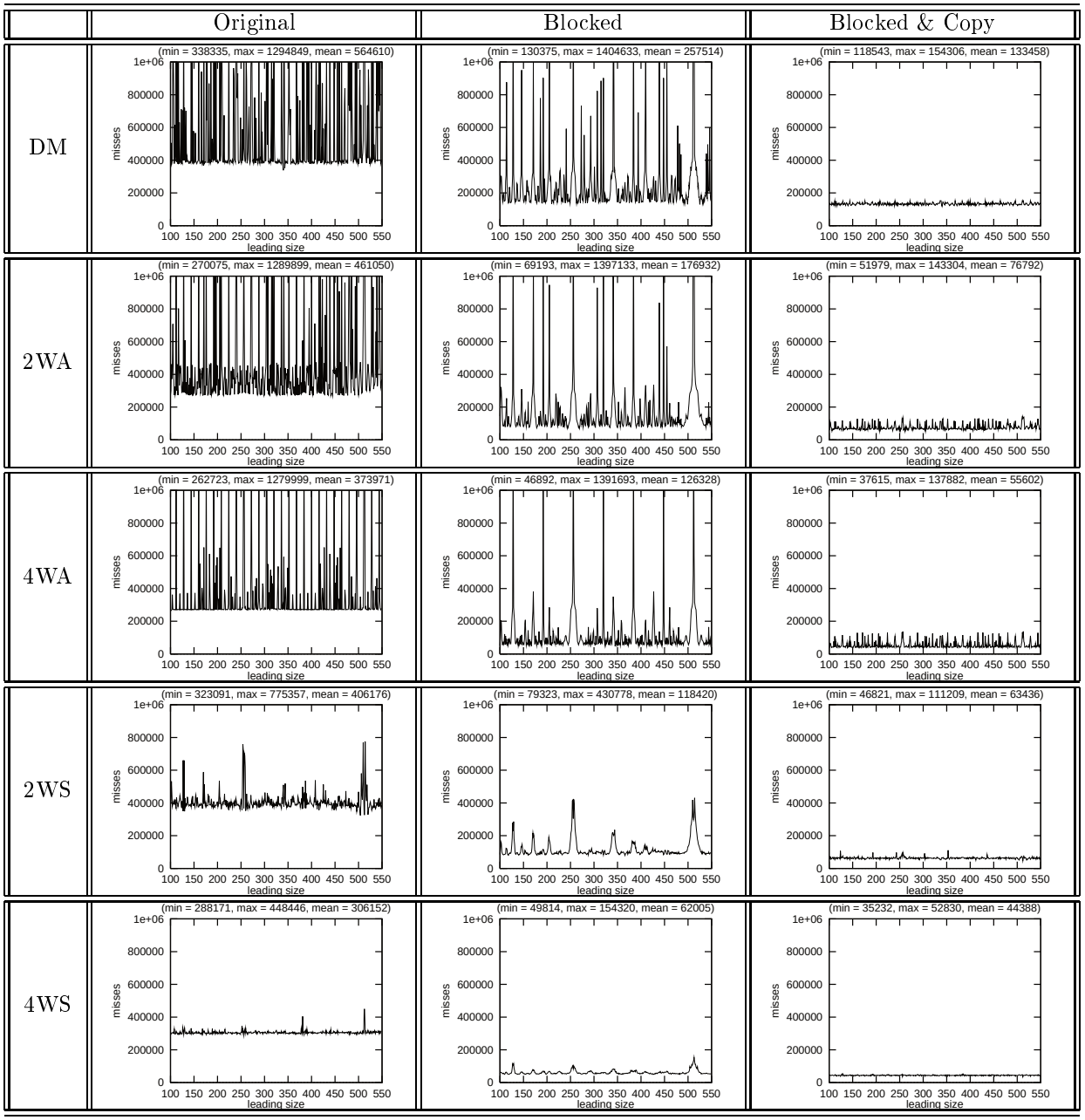


Figure 6: Matrix-matrix multiply. DM stands for Direct Mapped, 2WA for 2 way set-associative cache, 4WA for 4 way set-associative cache, 2WS for 2 way skewed-associative cache, 4WS for 4 way skewed-associative cache.

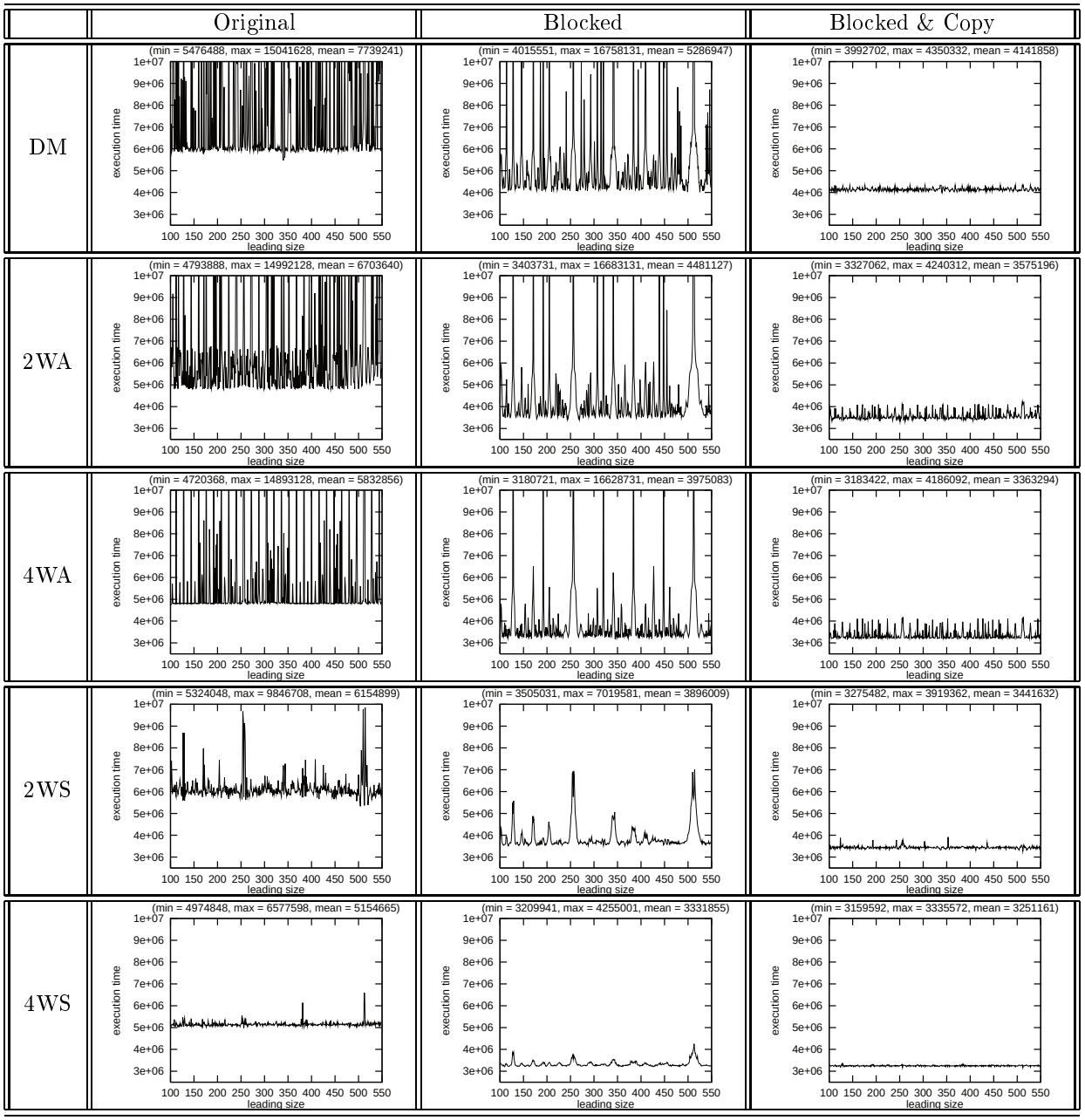


Figure 7: Execution times for 100*100 matrix-matrix multiply

Hot-spots on hashing functions

For blocked algorithm, the bad behavior of set-associative caches is due to hot-spots on the indexing function used in the cache: numerous blocks are competing for the same set.

Using a pseudo-randomize hashing function to compute the set number as proposed in [8] limits the hot-spots and leads to a more predictable performance. Computing a pseudo-random hashing function requires some complex hardware. Using such a function can not be realisticaly considered for L1 caches. Nevertheless, our experiments showed that the average performance which would be obtained by using such a pseudo-random hashing function to index a 4-way set-associative cache is lower than the performance obtained by using the usual truncation bit hashing function: a blocked algorithm exhibits some spatial locality, but pseudo-random hashing functions do not exploit it.

The good behavior of the skewed-associative cache is not due to the particular choice of the skewing functions: these skewing functions do not randomize the distribution of requests. On the matrix multiply, for each of the selected skewing functions, there were particular values of the leading dimension of the arrays which conducted to hot-spots.

The good behavior of the skewed-associative cache is explained by the use of distinct hashing functions to index the cache banks: hot-spots are different for each cache bank.

```
do  j = 1,340
   do  i = 1,340
      temp = c0 * vxo(i,j) + dty2 * (vxo(i -1,j) + vxo(i + 1,j))
      + dtx2 * (vxo(i,j + 1) + vxo(i,j -1)) -dtx * (po(i,j) - (po(i,j-1)) -c1
      temp = temp * ivx(i,j)
      vxn(i,j) = temp
      temp = c0 * vyo(i,j) + dty2 * (vyo(i -1,j) + vyo(i + 1,j))
           + dtx2 * (vyo(i,j + 1) + vyo(i,j -1)) -dty * (po(i -1,j) -  po(i,j)) -c2
      temp = temp * ivy(i,j)
      vyn(i,j) = temp
   enddo
enddo
```

Figure 8: 2D Jacobi loop nest

	Original	Blocked	Bl & Co
floating point ref	1734008	1734008	2639946
data ref	2783428	2848872	2957116
instructions	6383412	6611319	8281656
ideal execution time	4179673	4164447	4748921

Table 2: Characteristics on the different 2D Jacobi versions

6.2 2D Jacobi Loop Kernel

The kernel studied here is a 2D Jacobi loop extracted from an application called PENAL [2], that computes permeability in porous media using a finite difference method. The original loop nest is shown in Figure 8. In this kernel, the data reuse is more limited than in the matrix-multiply: 5 reuses per datum on arrays vx0, vy0, 3 reuses per data in array po[2] , and only one access to each element of arrays ivx, ivy, vxn and vyn. ivx and ivy are integer arrays. Since there are 4 floating point words or 8 integer words per cache block, the cold misses on these seven arrays represent $\frac{5*340*340}{4} + \frac{2*340*340}{8} = 173\ 400$ misses.

Blocking the loop does not induce any extra reference to the arrays, this explains why the *ideal execution times* for the original loop and for the blocked loop are in the same range.

In terms of array accesses, the extra cost of copying is huge. The three arrays vx0,vy0 and po have to be copied generating 905 938 extra references on floating data: more than one third of the floating-point data references are done while copying! Paradoxically, the total number of memory references in three versions of the algorithm (table 2) are in the same range because the f77 compiler generates six references to scalars used for address generation in the original and the blocked algorithms inducing 674400 extra memory references.

The numbers of misses are given in Figure 9 for the array leading size ranging from 340 to 600 .

Original loop: The $10 * 340$ floating-point distinct elements and $2 * 340$ distinct integer elements are used in one iteration of the outermost loop, this exceeds the size of the cache. Most of the data used in iteration j will be invalidated in the cache before it can be reused during iteration j+1. This effect can be seen on Figure 9.

Blocked loop: As, for the matrix multiply, some pathological behaviors can be observed for usual cache structures, while the behavior of the 4-way skewed-associative cache is quite uniform. Small irregularities are essentialy due to good or bad alignement of the arrays on cache blocks.

Blocked & Copied loop: Its advantage is to exhibit a regular behavior, however the price of copying is huge:

- As previously mentionned, the number of array references is increased by 50%, so the *Ideal execution time* is also significantly longer than for other kernel implementations.
- for all cache organizations but the direct-mapped, the average number of misses is 50% higher than for the blocked loop.

[2] 4 reuses are present, but analysis of the assembler code confirms that one of this reuse is captured by registers

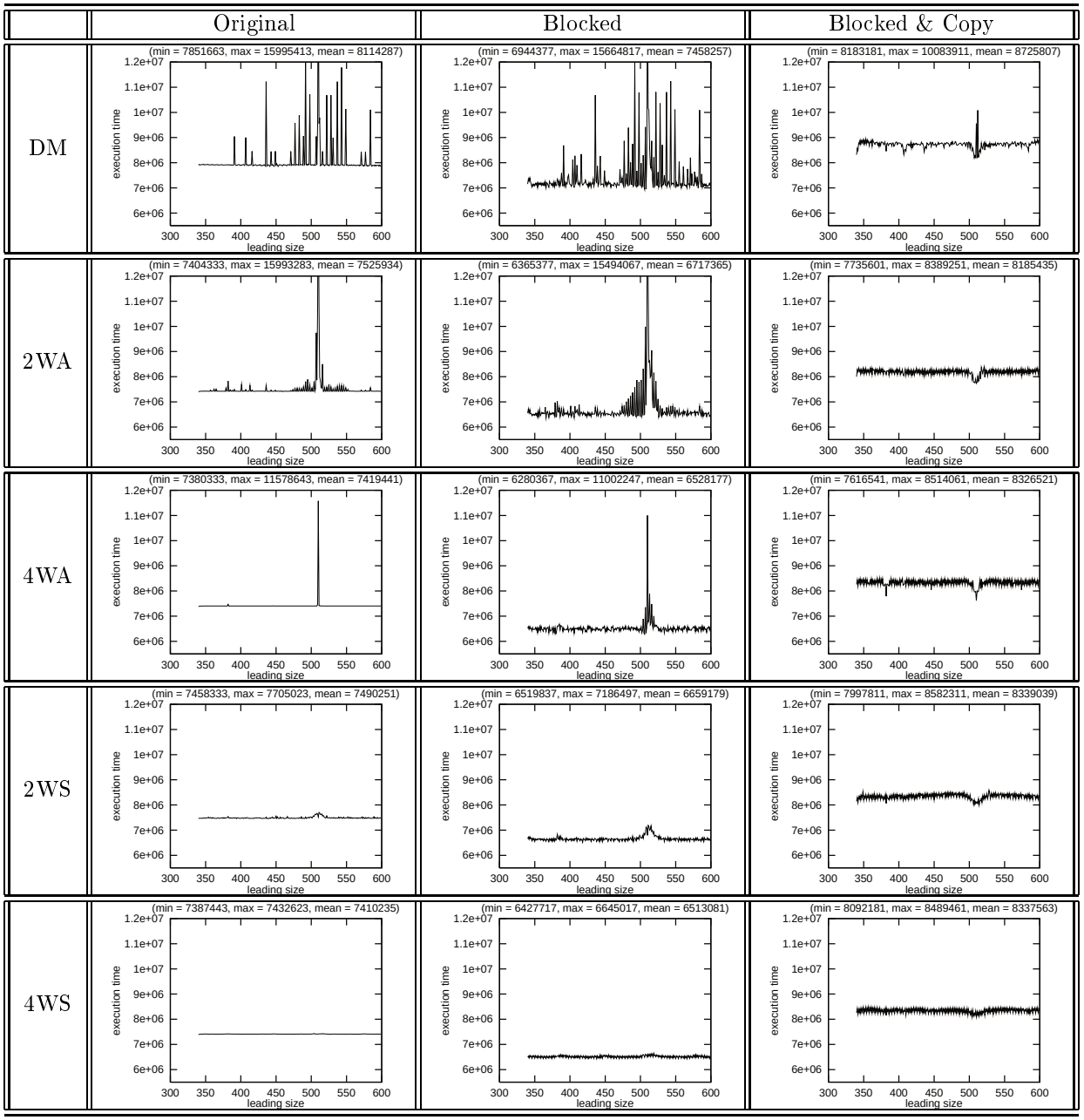


Figure 9: Execution times for the 2D Jacobi loop

6.3 LU factorization

The last loop nest we experimented is a 100×100 LU factorization without pivoting. The blocked and blocked copied versions were derived by hand from the following algorithm (refer to figure 10:

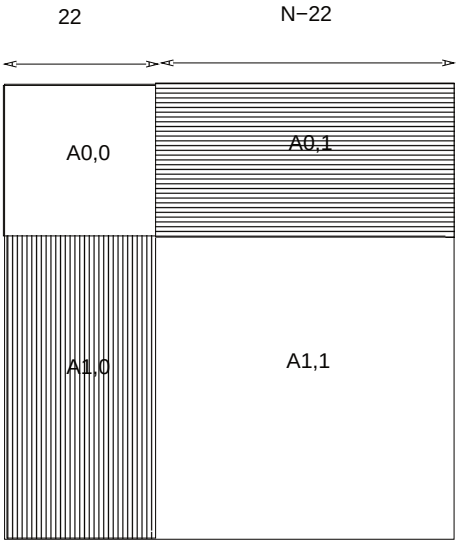


Figure 10: Blocked LU

- if $N \leq 22$, direct LU factorization of matrix A else
  1. LU factorization of submatrix A0,0
  2. update of the $(N-22) * 22$ submatrix A0,1 (resolving (N-22) triangular systems $Tx = y$ with the same matrix T).
  3. update of the $22 * (N-22)$ submatrix A1,0 (multiply of a triangular matrix by a full matrix)
  4. update of the $(N-22) * (N-22)$ submatrix A1,1 by $A1,1 = A1,1 - A1,0 * A0,1$
  5. LU factorization of the submatrix A1,1

The characteristics of the three codes are given in Table 3. Notice that for the blocked and blocked & copied versions of the algorithms, the numbers of memory accesses (and also *the ideal execution times*) are significantly lower than for the original LU factorization. Steps 2, 3 and 4 were coded so that each multiply-substract induces two memory accesses instead of three memory accesses for the original loop.

The number of misses for the three codes are given in Figure 11.

Original loop: Only the direct mapped cache exhibits erratic behavior. The average number of misses for the direct mapped cache is 12 % higher than on the other caches.

Blocked loop: Blocking is effective on average on this kernel but some very high miss ratios are exhibited for some particular values of the leading size with direct mapped and set associative caches.

Blocked & Copied loop: Copying is effective to reduce peak miss rates. Nevertheless significant behavior differences for different parameters are encountered for direct-mapped and set-associative caches. For instance on the 4-way set-associative cache, the execution time still ranges from 1255328 to 1910138 cycles.

As with the previous experiments, the 4-way skewed associative cache exhibits a good stable behavior for the blocked version of the algorithm, therefore copying is not needed.

	Original	Blocked	Bl & Co
floating point ref	1004952	728111	768991
data ref	1010074	738036	778279
instructions	3076031	2465279	2737860
ideal execution time	1400495	1023964	1049508

Table 3: Characteristics on the different LU versions

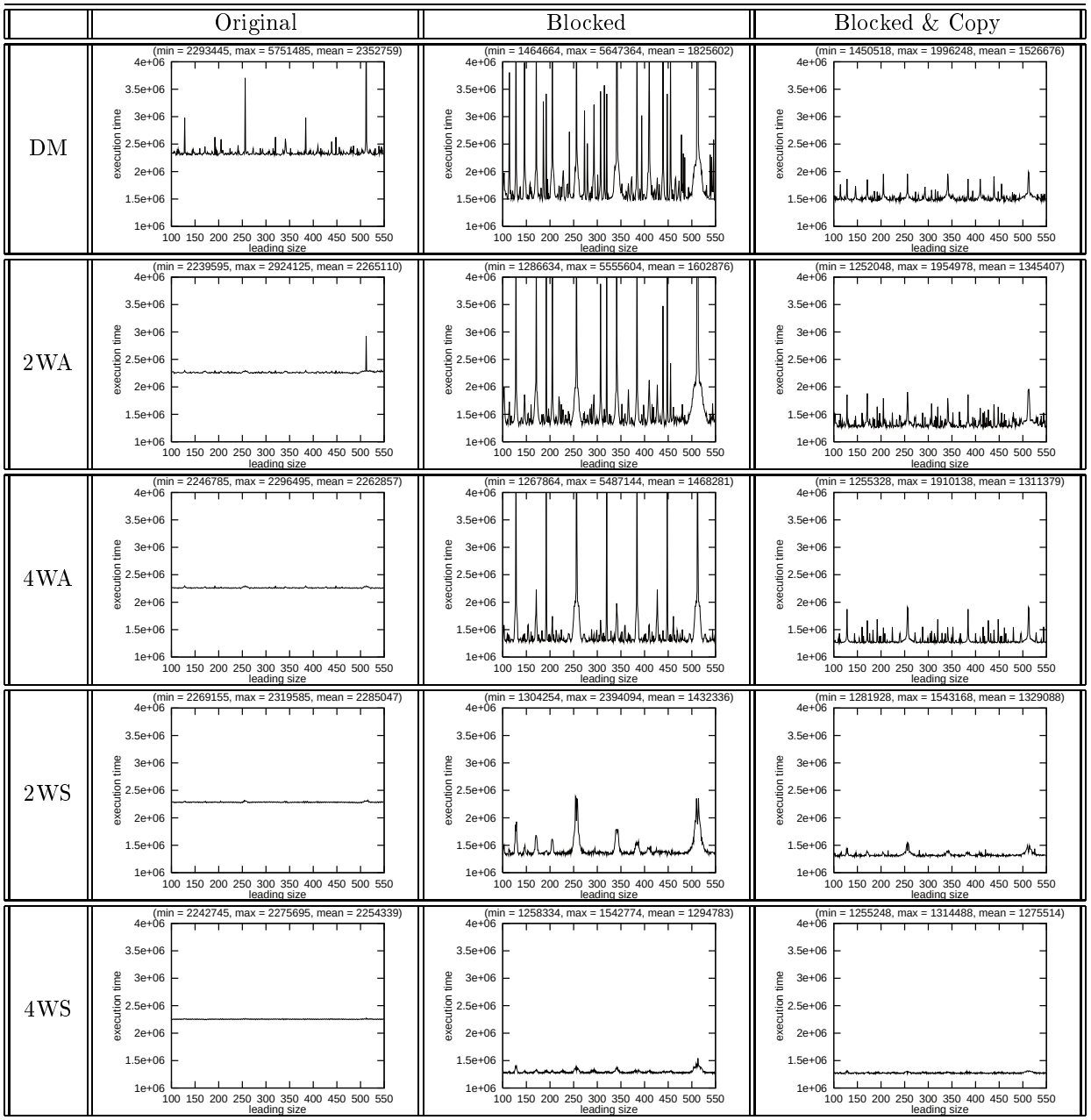


Figure 11: Execution times for LU

block size	8	12	16	20	24	32
required cache fraction	7 %	15 %	26 %	41 %	58 %	103 %
array ref.	3888000	3744000	3686400	3628800	3600000	3571200
data ref.	4054635	3818480	3734266	3655908	3618925	3583406
instructions	13316447	11726052	11139048	10580132	10311207	10049304
Ideal execution time	5867338	4961618	4627352	4309654	4157018	4008524

Table 4: Statistics on 120*120 blocked matrix multiplications and required cache spaces for different block sizes

7 Use of cache space

In the previous experiments, the blocking factor was computed so that the data reused in a block fits in approximately half of the cache size. For each algorithm, the best blocking factor depends on the cache organization.

Experiments previously presented in Section 4 seems to indicate that skewed-associative caches allows better usage of the cache space than set-associative caches.

The experiments presented in this section measure the influence of the blocking factor on the execution time for a 120×120 matrix-matrix multiply for both blocked and blocked & copied versions. The 120×120 size was chosen because the respective numbers of block matrix multiplies vary gracefully for blocking factors between 8 and 32[3].

7.1 Blocking

Statistics on the blocked matrix-matrix multiply for the different blocking factors are reported in Table 4. When the block size increases, the overhead due to loop blocking decreases. Therefore using a large block size is desirable, if it does not increase too much the number of misses.

The size of the data subject to reuse in a block, for the arrays A [4], B and C are respectively 1, bl, bl^2 with bl being the block size. In Table 4, we illustrate the required fraction of the cache for each block size used for a 120×120 matrix-matrix multiplication; statistics on the execution are also reported.

Simulations were done with blocking factor between 8 and 32 and the leading dimension of the matrix varying between 120 and 220 rows.

On Figure 12, we illustrate the minimum, maximum and average execution times in function of the block sizes[5].

In this experiment, we noticed that the block size had little influence on repartition of the miss peaks for conventional caches. The array sizes for which interferences appear are quite of the block size, but the magnitude of the interference phenomena increases with the block size.

Performance analysis

1. For all cache structures, the minimum execution time is obtained for a block size of 20. It can be noticed that when using the direct mapped cache, this minimum execution time is significantly higher than with the other cache structures (30 % higher).

2. For set-associative caches and direct-mapped caches, the shapes of the curves for minimum execution times and average execution times are different. The lowest average execution time is obtained with a block size of 12 for a direct-mapped cache and of 16 for 2-way and 4-way set associative caches.

   Note that there is a 20 % time difference between the minimum execution time and the average execution time on a 4-way set-associative cache.

[3] As the f77 compiler unroll four iterations of the inner most loop, only multiple of 4 were considered as blocking factors

[4] A is register allocated

[5] Maximum execution times for direct-mapped and set-associative caches are in the range of 20-30 millions of cycles, and do not appear on the curves!

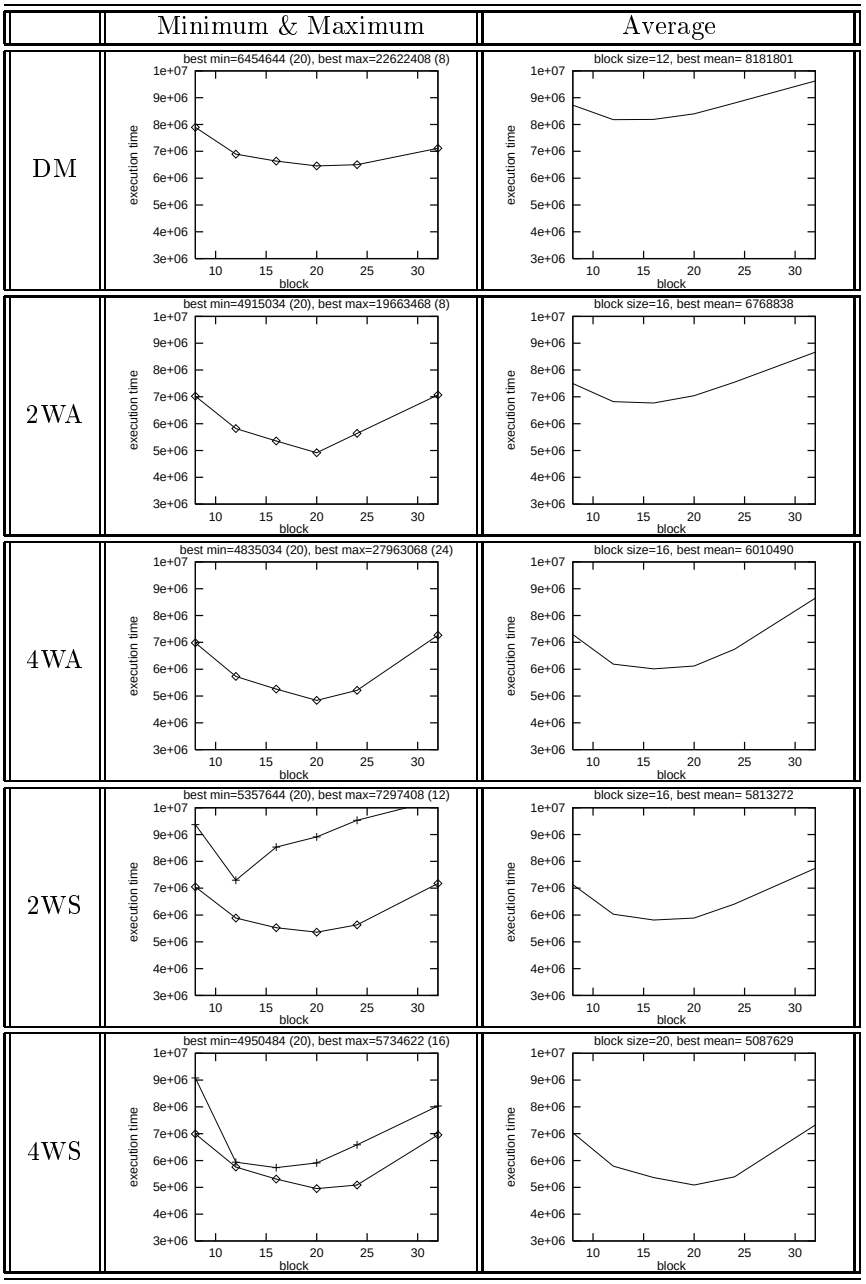


Figure 12: Blocked matrix multiply with various block sizes (4 to 32) and different leading sizes (120 to 220)

3. For the 4-way skewed-associative cache, the difference between minimum execution time and average execution time is only 3%. The best mean execution time is obtained with a block size of 20.

4. The best average performance for 4-way skewed-associative caches is about 18% better than that of 4-way set-associative cache.

5. When using direct-mapped and set-associative caches, the execution times for some specific values of the leading dimension rises to clearly unacceptable values (over 20 million) independently of the blocking factors. When using a 4-way skewed-associative cache, the difference between maximum execution time and average execution time remains within 20%.

Summary For an 8Kbyte 4-way skewed-associative, a blocking factor of 20 or 24 may be used to the best average performance. The lowest performance remains in acceptable limits. 40 to 60 % of the cache space may be used for blocking without generating any desastrous behavior when varying the matrix leading dimension.

For direct-mapped caches (resp. 4-way set-associative caches) , the best average performance is obtained with a lower blocking factor of 12 (resp. 16). Only 15 % (resp. 25 %) of the cache space is used for blocking. The average performance is quite disappointing for direct-mapped caches as well as set-associative caches. Moreover, catastrophic execution times are encountered for some matrix leading sizes.

7.2 Blocked & copied algorithm

When using direct-mapped or set-associative caches, blocking is clearly insufficient because it still leads to unacceptably low performance for some array leading size. Copying may be used to limit placement conflicts. We study here the impact of the blocking factor on performance for direct-mapped caches and set-associative caches as well as for skewed-associative caches.

Statistics on the different blocked & copied version executions are reported in Table 5. Figure 13 illustrates the execution times for the different simulations.

In terms of *ideal execution time*, the overhead due to blocking & copying over blocking appears to be quite important for low blocking factors (12 or 16).

Figure 13 clearly illustrates that for direct-mapped caches as well as for set-associative caches, larger blocking factors may be used with the block and copy method than with plain blocking thus resulting in better average performance. The average performance remains relatively poor for direct-mapped cache.

For direct-mapped and set associative caches, peak execution times are also dramatically lowered when blocking and copying the data compared to when the data is only blocked. Nevertheless the leading size of arrays still impact significantly on performance. For instance the execution time of the 4-way set-associative cache with a blocking factor of 20, the execution time varies between 4934514 cycles and 6767734 cycles: more than 30 % !

Compare this with the remarkable stability of the execution time of the 4-way skewed-associative cache. With a blocking factor of 24, the execution time varies between 4960040 cycles and 5208650 cycles (only a 5% difference).

Summary For direct-mapped and set-associative caches, blocking and copying is more efficient than plain blocking. About 40 % of the cache space may be used for blocking and copying. Nevertheless, direct-mapped caches still exhibits relatively poor average performance while the behavior of the set-associative cache stays relatively unstable.

When using a 4-way skewed associative cache, more than 60 % of the cache space may be used for blocking and copying. Moreover note that the performance is stable and predictable.

7.3 Summary

Figure 7.3 summarizes the advantage of 4-way skewed-associative cache over a 4-way set-associative cache.

Matrix multiply and LU decomposition are illustrated with sizes 120 *120. The curve of the 4-way set-associative cache is obtained by taking for each value of the leading size the best execution time from the

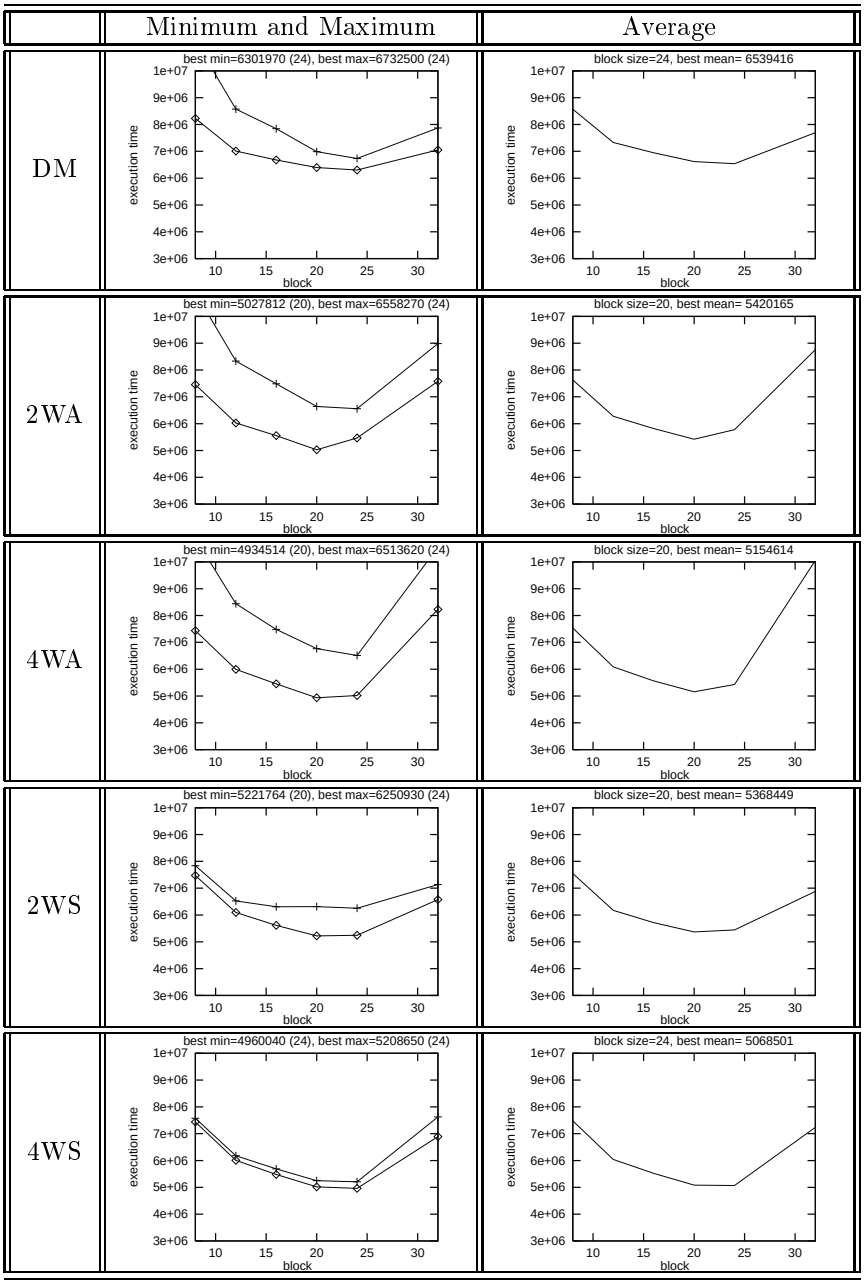
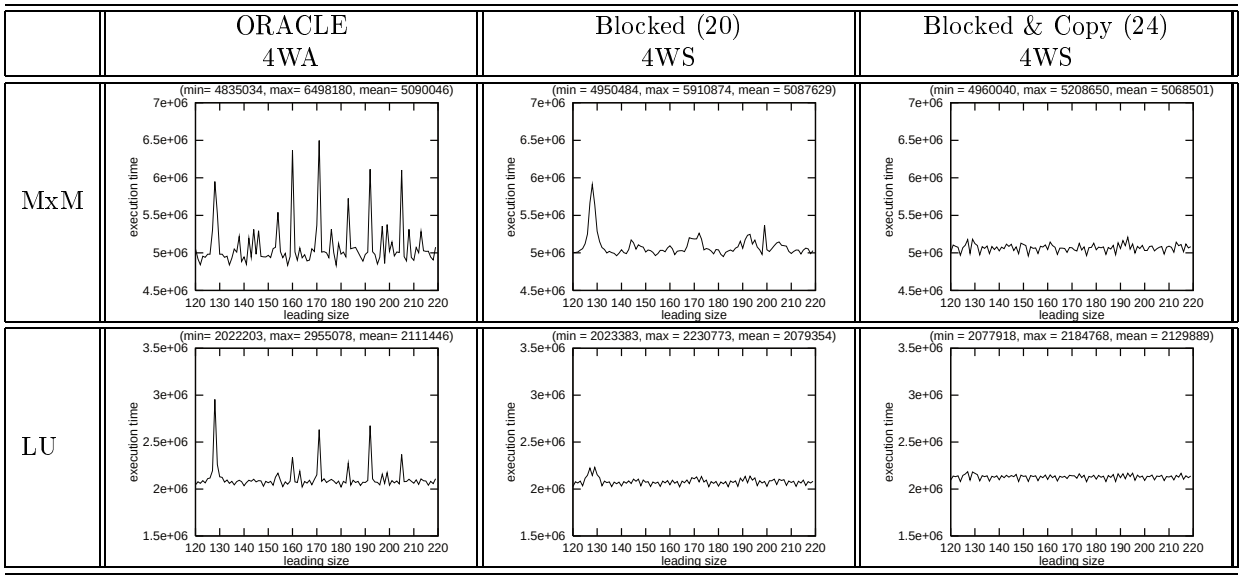


Figure 13: Blocked & copy matrix multiply with various block sizes (4 to 32) and different leading sizes (120 to 220);

block size	8	12	16	20	24	32
array ref.	4348800	4060800	3945600	3830400	3772800	3715200
data ref.	4543841	4147926	4001572	3862082	3794911	3729456
instructions	14176073	12247763	11544353	10879751	10562004	10253958
Ideal execution time	6341012	5237272	4838252	4463504	4285230	4113026

Table 5: Statistics on 120*120 blocked & copied matrix multiplications



12 executed simulation runs (i.e blocking factors 8,12,16,20,24 or 32 and copying or not). In reality, such performance can only be reached if an "oracle" algorithm is used before execution to determine the optimal parameters (block size and copy or not).

Two curves are given for the 4-way skewed-associative cache, the first one with the blocked algorithm using 20 used as the blocking factor, the second one with the blocked & copied algorithm and a blocking factor of 24.

The three curves exhibit almost the same mean performance. The set-associative cache curve is the best that can be obtained, nevertheless it exhibits variations in performance that may be very troublesome for many users: over 30 % variation for the matrix multiplication and around 45 % for the LU decomposition.

On the other hand, curves with the 4-way skewed-associative cache exhibit stable behavior without requiring any complex software analysis at compile time.

8 Conclusion

In this paper, we have studied the behavior of original, blocked and blocked & copied versions of three numerical kernels while varying the relative placement of the arrays in memory on skewed-associative caches and on conventionnal set-associative and direct-mapped caches.

Our experiments have shown that, when using a set-associative cache or a direct-mapped cache, blocking is not sufficient to guarantee a correct level of performance. This is coherent with previous studies [6, 8]. Our experiments have also shown that blocking and copying, however costly allows a better level of performance when using these cache organizations. Furthermore:

- Direct-mapped caches deliver poor average performance compared to other cache configurations.

- When using set-associative caches, significant execution time differences exist for different array placements, even with blocked & copied versions of the algorithms.

The high execution time variations shows that for many numerical applications, set-associative caches are not adequate structures and any performance numbers are very suspect as a representation of the real processor performance.

The experiments presented in this paper show that the skewed associative cache recently proposed in [9, 10] has lower and more stable miss ratio than the corresponding set-associative cache.

Our experiments have shown that the behavior of the 4 way skewed-associative cache is quite insensitive to variations of array placements in memory. It may provide to the user predictable cache behavior and therefore a predictable performance. This is particularly true for blocked and for blocked & copied versions of the algorithms. While copying is not necessary to improve average performance of skewed-associative caches, direct-mapped and set-associative caches all necessitate copying to improve and stabilize the cache behavior and get relatively predictable performance. This is a clear advantage over set-associative caches when the reuse factor is small and the cost of copying is too high compared to the cost of computation, (as in 6.2) or when copying is impossible. If a skewed-associative cache is used and copying is possible, it may be used to guarantee a very stable behavior.

At last, we have shown in section 4, that, skewed-associative caches are likely to allow a better use of the cache space than set-associative cache. Simulations presented in Section 7 have confirmed that a larger fraction of the cache may be used for blocking on a skewed-associative cache than with a set-associative cache. This leads to reduced blocking overhead costs.

Current processors use set-associative caches or direct-mapped caches. To track performance, compiler designers and application programmers must improve data locality (i.e. limiting capacity misses) and limit conflict misses for a reasonable cost (avoiding copying when possible). Using a 4-way skewed-associative in a processor would allow the users to focuss their efforts only on improving data locality.

References

[1] F. Bodin, C. Eisenbeis, W. Jalby, D. Windheiser, "A quantitative algorithm for data locality optimization" In *Code Generation-Concepts, Tools, Techniques*, Springer Verlag, pp 119-145 1992.

[2] D. Bernard, F. Bodin, A. Goasguen, C. Fechant, "Implementing a two dimensional pore-scale flow model on different parallel machines", Proceedings of X international Conference on Computational Methods in Water Resources, June 1994.

[3] D. Callahan, S. Carr, K. Kennedy, "Improving Register Allocation for Subscripted Variables", Proceedings of the Conference on Programming Language Design and Implementation, pp 53-65, 1990.

[4] C. Eisenbeis, W. Jalby, D. Windheiser, F. Bodin, "A Strategy for Array Management in Local Memory", Mathematical Programming, Special issue on Applications of Discrete Optimization in Computer Science, 1994.

[5] G.Irlam "Spa" personal communication 1992; the Spa package is available from gordoni@cs.adelaide.edu.au

[6] M. Lam,, E. Rothberg, M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", Proceedings of the Fourth ACM ASPLOS conference, April 91, pp 63-75.

[7] A. Porterfield, "Compiler management of program locality", Technical Report, Rice University, Houston, Texas, January 1988.

[8] M. Schlansker, R. Shaw, A. Sivaramakrishnan "Randomization and Associativity in the Design of Placement-Insensitive Caches" HP Laboratories Technical Report 93-41, June 1993

[9] A. Seznec, "A case for two-way skewed associative caches", Proceedings of the 20th International Symposium on Computer Architecture, pp 169-178, May 1993

[10] A. Seznec, F. Bodin, "Skewed-associative caches", Proceedings of PARLE' 93, Munich, pp 305-316 June 1993

[11] H.S. Stone, "Parallel processing with the perfect-shuffle", IEEE Transactions on Computers, pp 153-161, Feb. 1971

[12] O. Temam, E. Granston, W. Jalby "To Copy or Not to Copy : A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts" Proceedings of Supercomputing'93 (ACM), pp 410-419, Nov. 1993

[13] M. Wolf, M. Lam, "An algorithm to generate sequential and parallel code with improved data locality", Technical Report, Stanford University 1990.

[14] M. Wolf, M. Lam, "A Data Locality Optimizing Algorithm", ACM Conference on Programming Language Design and Implementation, pp 30-44 June 1991.