Skewed associativity enhances performance predictability

François Bodin, André Seznec

▶ To cite this version:

François Bodin, André Seznec. Skewed associativity enhances performance predictability. [Research Report] RR-2499, INRIA. 1995. inria-00074177

HAL Id: inria-00074177
https://inria.hal.science/inria-00074177

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Skewed associativity enhances performance predictability

François Bodin, André Seznec

N° 2499

Février 1995

PROGRAMME 1

ISSN 0249-6399

Skewed associativity enhances performance predictability

François Bodin, André Seznec* **

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Caps

Rapport de recherche n° 2499 — Février 1995 — 25 pages

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 numerical data is 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. As a result of its better comportment, it is possible to use larger blocks sizes with blocked algorithms, which will furthermore reduces blocking overhead costs.

Key-words: cache, predictable performance, loop blocking, skewed-associative caches

(Résumé : tsvp)

*e-mail :bodin, seznec@irisa.fr

**This report is an extended version of the article "Skewed associativity enhances performance predictability" which will appear in Proceedings of the 22th International Symposium on Computer Architecture (June 1995)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Le comportement des caches associatifs brouillés est prévisible

Résumé : Les performances des ordinateurs d'aujourd'hui sont devenues très difficiles à exploiter et à prévoir. Un des facteurs rendant cette prévision extrêmement complexe est l'utilisation de hiérarchies mémoires, et en particulier de mémoire caches. Des transformations de programmes telles que le blocage de boucles peuvent être utilisées pour améliorer la localité des applications dans les codes numériques. Malheureusement, le comportement des caches à correspondance directes et des caches associatifs par ensemble sont très sensibles à des paramètres tels que le placement des tableaux en mémoire ; ceci entraine parfois des chutes de performances imprédictibles et catastrophiques même sur des codes bloqués. La plupart des utilisateurs ne peuvent pas être conscients de tels phénomènes.

Dans cet article, nous montrons que le cache associatif brouillé 4 voies que nous avons récemment proposé est à peu près insensible au placement relatif des tableaux en mémoire et qu'il fournit ainsi une performance prédictible et stable à l'usager.

De plus, sur les algorithmes bloqués, une partie importante de l'espace d'un cache associatif brouillé peut être utilisé pour les données réutilisables ; ceci limite le surcoût lié au blocage de boucles.

Mots-clé : cache, performance prévisible, caches associatifs brouillés

1 Introduction

Performance tuning on today's computers has become very complex. One factor of this complexity is the use of memory hierarchies, and particularly of cache memories. As the miss penalty is becoming higher and higher, performance becomes very sensitive to the cache performance. Unfortunately, the behaviors of direct-mapped and set-associative caches are very sensitive to small variations of the application's parameters. Since the caches are not perfect (limited associativity, non-optimal replacement strategy), performance may suffer unpredictably from conflict misses even with blocked loops [6]. For instance, in a recent study, Schlansker et al [8] showed that, even with a very regular memory access patterns such as iterating on the read of a fixed size memory subblock, 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 result in a hit or nearly all the access result in a miss.

For most users, such unpredictable behaviors can not be accepted. Getting predictable and stable performance is a major issue.

Recently, the skewed-associative cache, a new associative cache structure has been proposed in [9, 10]. In this paper, we investigate the sensitivity of the skewed-associative cache to parameters such as size of arrays or relative placements of the arrays in numerical kernels on dense structures. Unlike usual set-associative caches, a 4-way skewed-associative cache is quite insensitive to those application's parameters. This leads to better average miss ratio than set-associative cache and more predictable performance.

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

Our simulations also established that, even when using restructuring technique such as blocking and copying, performance of direct-mapped caches are 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 in 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. Experiments have also been conducted which indicates that, when using a 4-way skewed-associative cache, a larger fraction of the cache may be used for blocking.

The remainder of the paper is organized as follows. In Section 2, we recall the principles of a skewed-associative cache. In Section 3 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 4, we present the simulation tools which have been used in the paper. In Section 5, the impact of various array placements is studied on a few numerical kernels. Original, blocked and blocked copied algorithms are studied. In Section 6, we study the impact of the blocking factor on performance and try to characterize the fraction of the

cache size that is available for blocking (resp. blocking & copying) on set-associative caches as and on skewed-associative caches. Section 7 summarizes this study.

Related work

Improving performance by reducing capacity and conflict misses in numerical applications by software technique has been addressed in many studies. First studies [4, 12, 13, 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 any 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 cache configurations. In order to overcome this difficulty, blocks exhibiting high level of reuse may be copied in order to control the placement of data in memory and avoid placement conflicts in the cache [11, 4, 6]. But copying may induce large overhead on many numerical kernels. Techniques for determining whether copying is needed or not (e.g. [11, 6]) address direct-mapped caches and are still very conservative. Moreover 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 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. Nevertheless their proposal suffers from two major drawbacks: first a high degree of associativity is needed (in the 16-32 range) and second, some complex hardware mechanism is needed to pseudo-randomize the set selection in the cache.

2 Skewed-associative caches

2.1 Principle

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

For a skewed associative cache (Figure 2), different mapping functions are used for each cache banks i.e., a memory block at address D may be mapped on physical line $f_0(D)$ in cache bank 0 or in physical line $f_1(D)$ in cache 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.

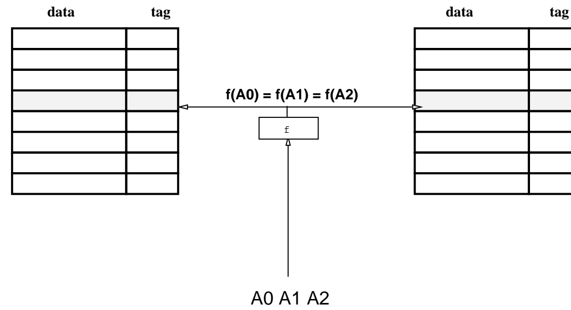

Figure 1: 3 data blocks conflicting for a single set on a two-way set-associative cache. When A0 and A1 are stored, A2 has to kick out one of the two

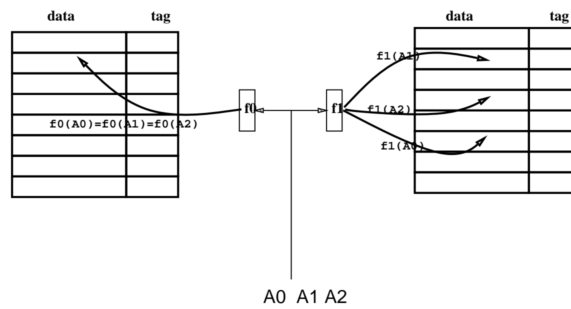

Figure 2: A0, A1 and A2 compete for the same location in bank 0, but can be present at the same time, as they do not map to the same location in bank 1

2.2 Choosing skewing functions

When designing a skewed associative cache, the mapping functions may be chosen in order to minimize conflict misses and hardware cost. We list some of these properties [9, 10].

Inter-bank dispersion In a usual X-way set-associative cache, when (X+1) data blocks contend for the same set in the cache, there is a conflict and one of the blocks must be rejected from the cache (Figure 1).

Skewed-associative caches avoid such a situation by scattering the data. Mapping functions can be chosen such that whenever two data blocks conflict for a single location in cache bank i, they have a very low probability of conflicting for a location in cache bank j (Figure 2).

Local dispersion in a single bank Many applications exhibit spatial locality, therefore the mapping functions must be chosen such as to avoid two "almost" neighboring data blocks conflicting for the same physical cache lines in bank i. The mapping functions f_i must be chosen in order to limit mapping conflicts such as the mapping of consecutive data blocks in a single cache bank i.

Simple hardware implementation A key issue for the overall performance of a processor is the pipeline length. Using distinct mapping functions on the distinct cache banks should have no effects on performance, as long as the computations of the mapping functions can be implemented into a non critical stage in the pipeline as not to lengthen the pipeline cycle.

2.3 An example of skewing functions

We present here the skewing functions which were used in the simulations that illustrate this paper. These skewing functions are obtained by XORing a few bits in the address of a memory block (as in [9, 10]).

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, 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 $A_1 \oplus A_2$ in cache bank 0
2. *or* on cache line $\sigma(A_1) \oplus A_2$ in cache bank 1
3. *or*[1] on cache line $\sigma^2(A_1) \oplus A_2$ in cache bank 2
4. *or* on cache line $\sigma^3(A_1) \oplus A_2$ in cache bank 3

[1] on a 4-way skewed associative cache

These functions satisfy the criterion for "good" skewing functions defined in [9] (inter-bank dispersion, local dispersion and low hardware costs).

A pseudo-LRU replacement policy similar to that described in [10] was used in simulations.

3 How a skewed-associative cache handles conflict misses

We have conducted a very simple experiment 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. A pseudo-LRU replacement policy was used for the skewed-associative cache[2]. The experiment was repeated on 100 different collections for every sequence size.

3.1 Data dispersion

Figure 3a 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.

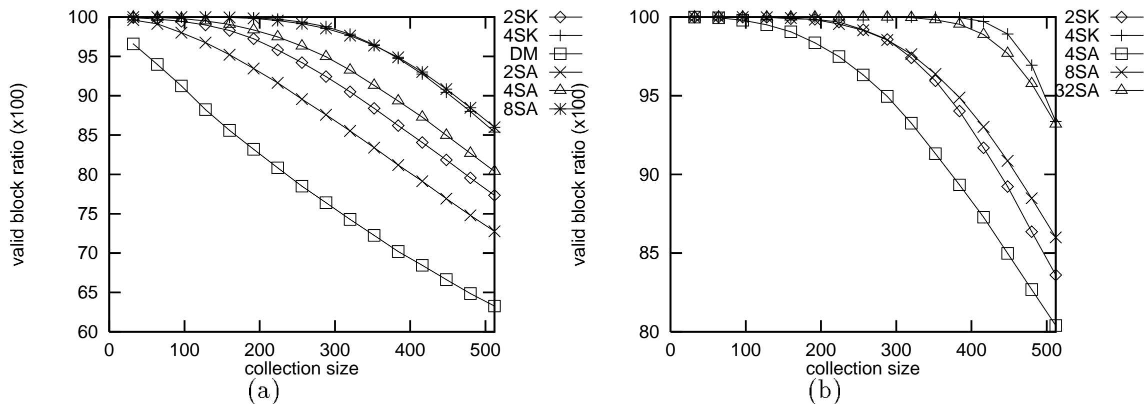

Figure 3: (a) Ratio of valid blocks after one read, (b) Ratio of valid blocks after ten reads

The number of valid blocks in the 2-way skewed-associative cache is greater than the number of valid blocks in the 2-way set-associative cache and slightly less than the number of valid blocks in the 4-way set-associative cache.

[2]For the set-associative cache, the parameter measured in this experiment does not depend in any way of the replacement policy

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

After a single read sequence, for an equal associativity degree, more data will be on a skewed-associative cache than on a set-associative cache.

3.2 Self data reorganization

A second phenomenon accentuates the advantage of the skewed-associative over the set-associative cache.

Figure 3b illustrates the average ratio of the blocks in the sequence that remain valid in the cache after ten successive reads of the whole sequence (sequence varying from 32 to 512 blocks). For direct-mapped and set-associative caches, the number of valid blocks does not evolve after the first sequence read: if a block is missing its target set, loading the block will invalidate another block in the cache.

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. Among the other possible locations for a data block D present in the cache at time t, there may be an empty location. Block D may be removed from the cache by a miss on an other block D' in bank i, but 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 sequence reading, the number of valid blocks in the 4-way skewed-associative cache (resp. 2-way) is in the same range as that of the 32-way set-associative (resp. 8-way) cache.

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 alow greater block size.

Notice that this example does not show sure performance gain. 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.

4 Evaluation methodology

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

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 binary code to be analyzed was required. User code of a single application can be completely traced except for the OS kernel code. As we studied the

behavior of numerical kernels consisting of a few nested loops, only data references were piped to a cache simulator.

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 (and simulation time), a 8Kbyte cache was simulated. The cache line size chosen for the simulation was 32 bytes.

Sensitivity to Parameter Variations 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).

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 bits branch target buffer was implemented. For these simulations, 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 roughly modelize the execution time of a kernel by:

$$T_{exec} = \text{ ideal execution time } + 10 * N_{miss} \quad (1)$$

5 Cache associativity and blocking/copying

As previously mentioned, the impact on cache behavior of the software technique proposed to improve data locality is not fully understood.

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 vary 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 6, this happens to be a good approximation.

5.1 Matrix-matrix multiply

The three versions of the programs which were simulated are illustrated in Figure 4. 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 then 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 more memory accesses (see Figure 4).
- 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 200000 extra memory references.

The number of instructions in 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 unrolls the inner most loop 4 times. This explains the large difference between ideal execution time of the original version of the algorithm and that of blocked, and blocked & copied algorithms.

	Original	Blocked	Bl & Co
floating point ref	2020000	2100000	2220000
data ref	2024880	2120751	2243408
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, then leading

to 400 reads of a single cache 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 5 and Figure 6, 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 quite unpredictable behaviors on the original version as well as on the blocked version of the algorithm. For instance, execution time of the blocked algorithm, the execution time vary from from 3 200 000 cycles to 16 000 000 cycles, even with a 4-way set-associative cache.

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 better on a 4-way skewed-associative cache than on anyother cache structure: for the blocked version, the average miss number is around 62 000 for the 4-way skewed-associative cache against 126 000 for the 4-way set-associative cache.

Blocked and Copied: Associating blocking and copying brings relatively stable numbers of misses for the matrix multiply.

With a 4-way set-associative cache, the execution time varies between 3 180 000 to 4 180 000 cycles.

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

The quite stable but relatively poor performance of the direct mapped cache must also be pointed out.

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.

5.2 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 7. Due to lack of space , blocked and blocked copied versions of the kernel are not illustrated. In this kernel, the data reuse is more limited than in the matrix-multiply: 5 reuses per data on arrays vx0, vy0, 3 reuses per data in array po[3] , and only one access to each element of arrays ivx, ivy, vxn and vyn. ivx and ivy are integer

[3] 4 reuses are present, but analysis of the assembler code confirms that one of this reuse is captured by registers

```
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 iT6_1=0,min(100-j,22)
tmp=a(i,j+iT6_1)
do iT6_2=0,min(100-k,22)
tmp=tmp+b(i,k+iT6_2)
     *c(k+iT6_2,j+iT6_1)
enddo
a(i,j+iT6_1)=tmp
enddo
enddo
enddo
enddo
```

(b)

```
do j=1,100,23
do k=1,100,23
do iw21_0=0,min(100-j,22)
do iw21_1=0,min(100-k,22)
c52(iw21_0+23*iw21_1)=
          c(k+iw21_1,j+iw21_0)
enddo
enddo
do i=1,100
do iw21_0=0,min(100-k,22)
b47(iw21_0)=b(i,k+iw21_0)
enddo
do iT6_1=0,min(100-j,22)
a37=a(i,j+iT6_1)
doiT6_2=0,min(100-k,22)
a37=a37+b47(iT6_2)*
    c52(iT6_1+23*iT6_2)
enddo
a(i,j+iT6_1)=a37
enddo
enddo
enddo
enddo
```

(c)

Figure 4: (a) Original matrix-matrix multiplication, (b) blocked matrix-matrix multiplication, (c) blocked matrix-matrix multiplication with copies

	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

arrays. Since there are 4 floating point words or 8 integer words per cache block, the first reference 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.

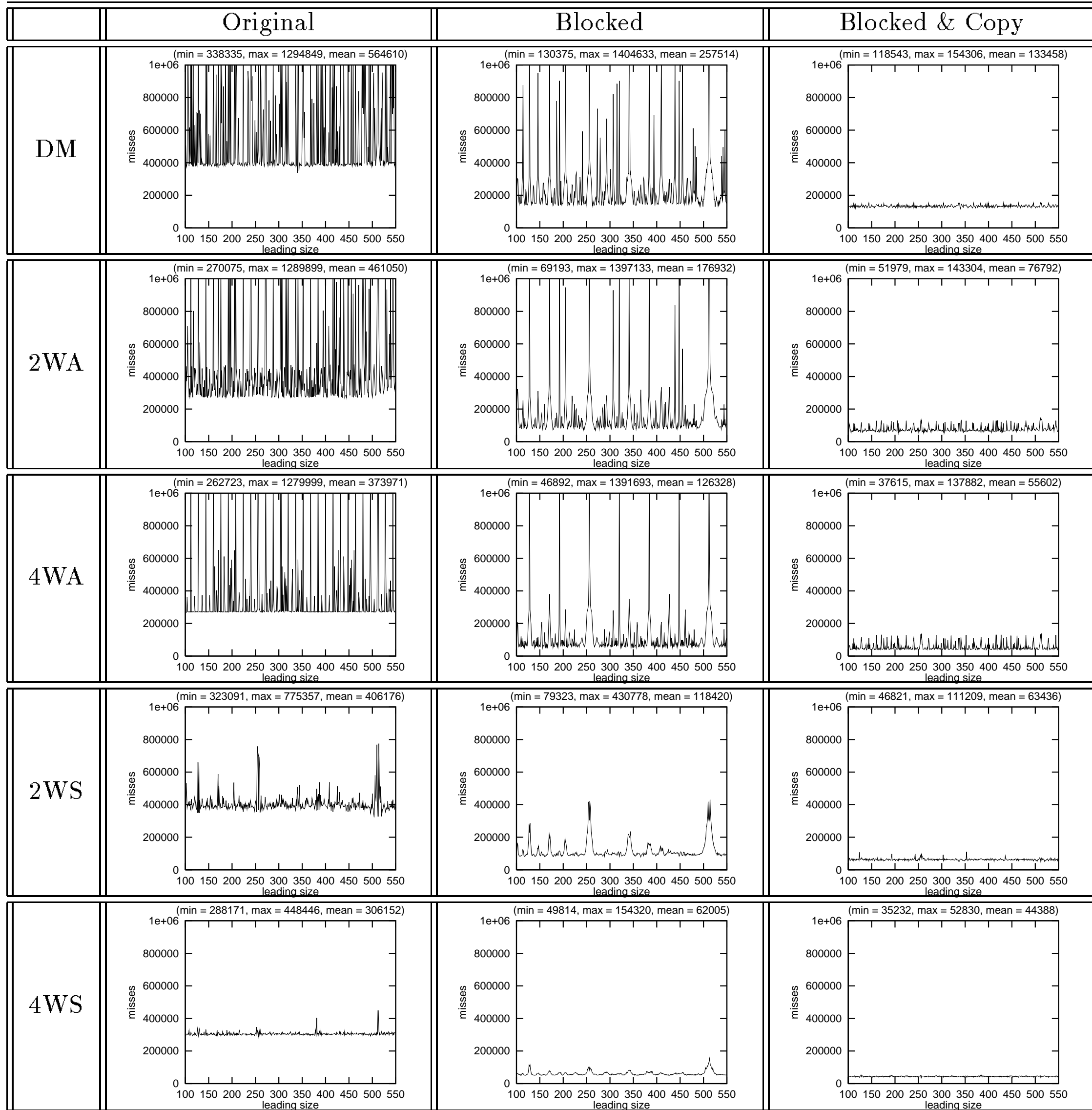

Figure 5: Matrix-matrix multiply. DM stands for Direct Mapped, 2WA for 2 way set-associative cache, 4WA for 4 way set-associative cache, 2WS for 2 way skewed-associative cache, 4WS for 4 way skewed-associative cache.

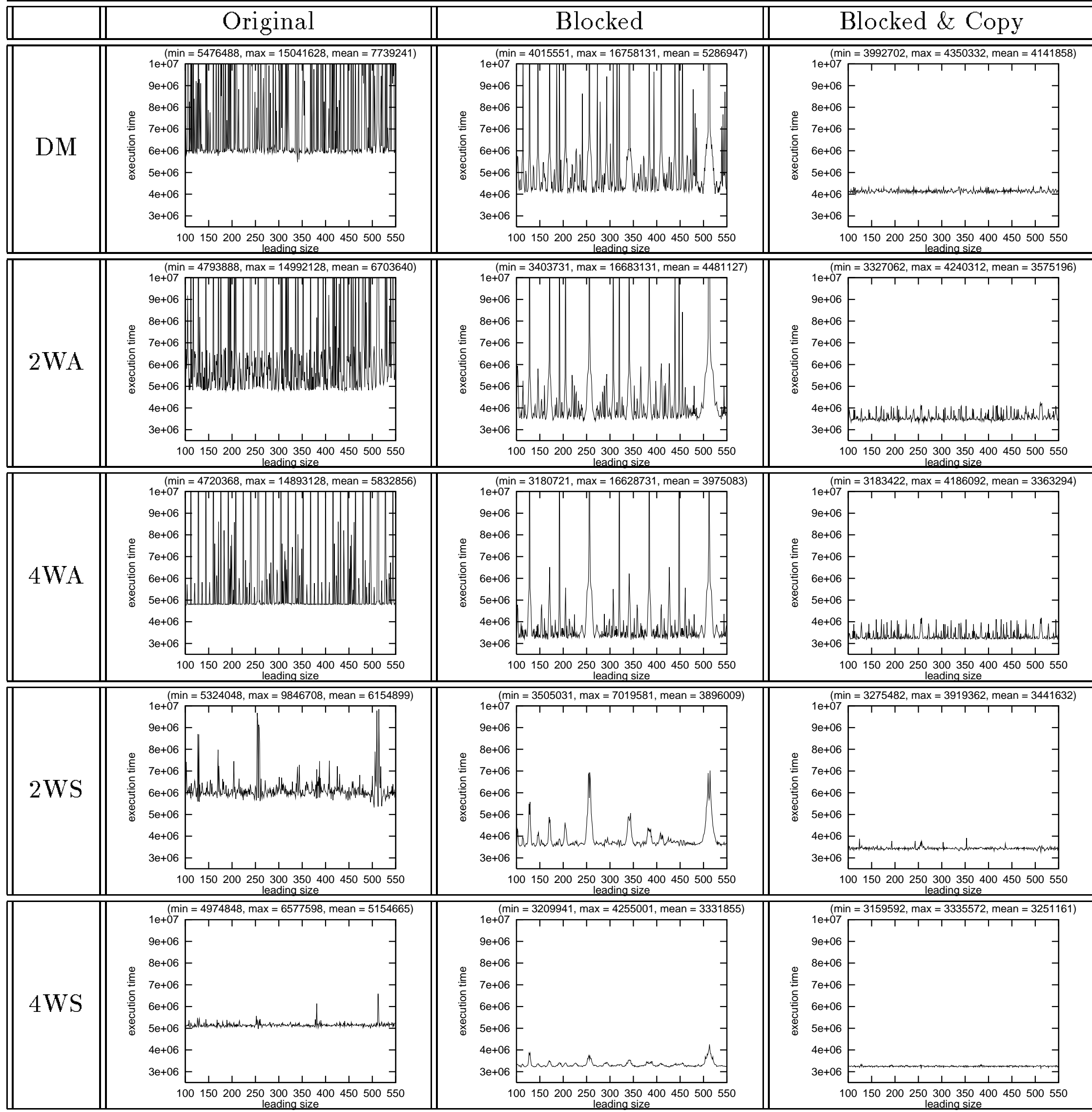

Figure 6: Execution times for 100*100 matrix-matrix multiply

```
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 7: 2D Jacobi loop nest

The numbers of misses are given in Figure 8 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. Then 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 8.

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 and Copy: 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 cache, the average number of misses is 50% higher than for the blocked loop.

5.3 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 9:

- if $N \leq 22$, direct LU factorization of matrix A else
  1. LU factorization of submatrix A0,0

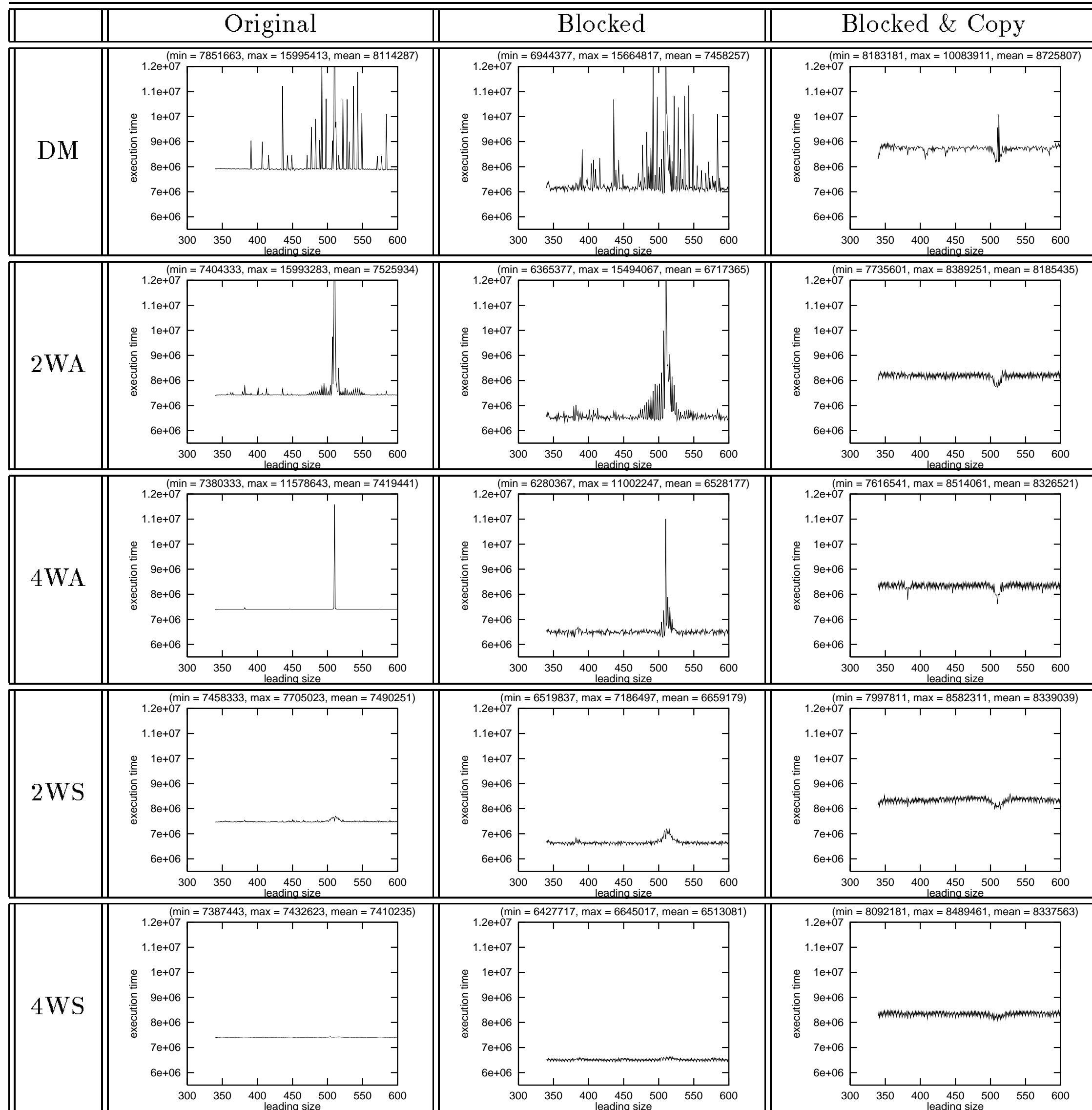

Figure 8: Simulation results for the 2D Jacobi loop