Skewed associativity enhances performance predictability*

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

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 four-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.

Keywords:

cache, predictable performance, loop blocking, skewed associative caches

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

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 [7]. For instance, in a recent study, Schlansker et al [9] showed that, even with a very regular memory access patterns such as iterating on the read of a fixed size memory sub-block, 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 [10, 11]. 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 four-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 four-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 four-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 [5, 13, 14, 8, 4] 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 [12, 5, 7]. But copying may induce large overhead on many numerical kernels. Techniques for determining whether copying is needed or not (e.g. [12, 7]) 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 [9] 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 [10, 11]. A X-way set associative cache 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 1), 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 [10, 11] that, for general applications, skewed associative caches exhibit an average lower miss ratio than set associative caches.

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 [10, 11].

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.

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 1).

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 [10, 11]).

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* 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

These functions satisfy the criterion for "good" skewing functions defined in [10] (inter-bank dispersion, local dispersion and low hardware costs).

A pseudo-LRU replacement policy similar to that described in [11] 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 four-way skewed associative were simulated. A pseudo-LRU replacement policy was used for the skewed associative cache[1]. The experiment was repeated on 100 different collections for every sequence size.

3.1 Data dispersion

Figure 2a 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 than the number of valid blocks in the 2-way set associative cache and slightly less than the number of valid blocks in the four-way set associative cache.

The number of valid blocks in the four-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.

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

3.2 Self data reorganization

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

Figure 2b 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 four-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 [7, 9] 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 [6] 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 four-way set associative, 2-way and four-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:

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

5 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 x 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

23 was used as the blocking factor for the **blocked** and **blocked & copied** versions. Statistics on the execution of the three simulated algorithms are reported in table 1. The number of instructions in the different versions depends highly on the quality of the F77 compiler. 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.

Figure 4 illustrates 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 four-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 four-way set associative cache.

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

sociative cache against 126 000 for the four-way set associative cache.

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

With a four-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 four-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.

Summary For all cache organizations, except the four-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 [1], that computes permeability in porous media using a finite difference method. The original loop nest is shown in Figure 3. 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[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 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 905938 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 execution times are given in Figure 5 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 5.

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

	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

Blocked loop As, for the matrix multiply, some pathological behaviors can be observed for usual cache structures, while the behavior of the four-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 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 x 100 LU factorization without pivoting. The blocked and blocked copied versions were derived by hand [3]. The characteristics of the three codes are given in Table 3. On the blocked and blocked copy versions of the algorithms, the numbers of memory accesses (and also *the ideal execution times*) are significantly lower than in the original LU factorization [3]. The number of misses for the three codes are given in Figure 6.

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

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

Blocked & Copied loop Copying is effective in reducing peak miss rate. Nevertheless significant behavior differences for different parameters are encountered for direct-mapped and set associative caches. For instance on the four-way set associative cache, the execution time vary from 1255328 to 1910138 cycles.

With the previous experiments, four-way skewed associative cache exhibits a good and stable behavior for the blocked version of the algorithm, 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

6 Use of cache space

In the previous experiments, the block size was computed so 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 3 seems to indicate that skewed associative caches allows a 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 when the blocking factor varies from 8 to 32[3].

Statistics on the blocked and blocked & copied matrix-matrix multiplies for the different blocking factors are reported in Table 4 (resp. Table 4). When the block size increases, the overhead due to loop blocking decreases the number of floating point data references, the number of instructions and the *ideal execution time*, thus using a large block size is desirable, if it does not increase too much the miss numbers.

Figure 7 illustrates the minimum, maximum and average execution times in function of the block sizes for both blocked and blocked & copied loops[4]. Experiences were conducted varying the row numbers in the matrix from 120 to 220.

Blocked loop For all cache structures, the minimum execution times are obtained for a block size of 20 and are in the same range (except for direct-mapped caches).

A 20 % difference between the minimum execution time and the average execution time on a four-way set associative cache must be noted while on the four-way skewed associative cache, the difference between minimum execution time and average execution time is of only 3%. Then difference between best average performance for four-way skewed associative caches and four-way set associative caches is about 18% in favor of skewed associative caches.

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

[4] For blocked loops, 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!

When using direct-mapped and set associative caches, the execution times for particular values of the leading dimension reach clearly unacceptable values (over 20 000000 cycles) and this for all blocking factors. When using a four-way skewed associative cache, the difference between maximum execution time and average execution time remains in the 20% range.

Blocked & copied loop 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 7 clearly illustrates that for direct-mapped caches as well as set associative caches, larger blocking factors may be used when blocking and copy method than with simple blocking thus resulting in better average performance. This average performance remains relatively poor for direct-mapped cache.

For direct-mapped and set associative caches, peak execution times are also dramatically lower when blocking and copying data than when only blocking it. Nevertheless differences on performance depending on the leading size of the arrays may be quite significant; for instance for the four-way set associative cache, for a blocking factor of 20, the execution time for the blocked and copy matrix multiply varies between 4934514 cycles and 6767734 cycles: more than 30 % !.

This has to be compared with the remarkable stability of the execution time for the four-way skewed associative cache; with a blocking factor of 24, the execution time varies here between 4960040 cycles and 5208650 cycles (only a 5% difference).

7 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 emphasized 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 [7, 9]. Our experiments have also shown that blocking and copying, however costly, allows to reach a better level of performance when using these cache organizations. It has also pointed out that :

- 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 for blocked & copied versions of the algorithms.

The high variations of execution times 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 performance of the processor.

The experiments presented in this paper show that the skewed associative cache recently proposed in [10, 11] has lower miss ratio and has a more stable behavior than the corresponding set associative cache.

Our experiments have shown that the behavior of the four way skewed associative cache is quite insensitive to variations of the array placements in memory. It may provide to the user a quite predictable cache behavior and then a predictable performance. This is particularly true on blocked and on blocked & copied versions of the algorithms. While copying is not necessary to improve average performance of skewed associative caches, when direct-mapped and set associative caches all necessitate copying to improve 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 remaining of the computation, (as in 5.2) or when copying is impossible. If a skewed associative cache is used and if copying is possible, it may be used in order to guarantee a very stable behavior.

At last, we have shown in section 3, that, using skewed associativity allows to effectively use more cache space than with set associative cache. As a result, it is possible to use a larger blocking factor with a skewed associative cache than with a set associative cache (see Section 6). This leads to a reduced blocking overhead.

On today's processors, set associative caches or direct-mapped caches are used. To track performance, compiler designers and application programmers must improve data locality (i.e. limiting capacity misses) and limit conflicts misses at a reasonable cost (i.e. avoiding copying when possible). Using a four-way skewed associative in a processor would allow the user to focuss his efforts on only improving data locality.

Acknowledgements

The authors wish to thank Dan Truong from IRISA for polishing their poor english style.

References

[1] 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.

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

[3] F. Bodin, A. Seznec, "Skewed associativity enhances performance predictability", PI IRISA 909, feb. 1995, available by ftp at irisa.irisa.fr as /techreports/1995/PI-909.ps (extended version of this paper)

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

[5] 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.

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

[7] 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.

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

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

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

[11] A. Seznec, F. Bodin, "Skewed associative caches", Proceedings of PARLE' 93, Munich, june 1993

[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), Portland, nov. 1993

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

[14] M. Wolf, M. Lam, "A Data Locality Optimizing Algorithm", ACM Conference on Programming Language Design and Implementation, June 26-28, 1991.

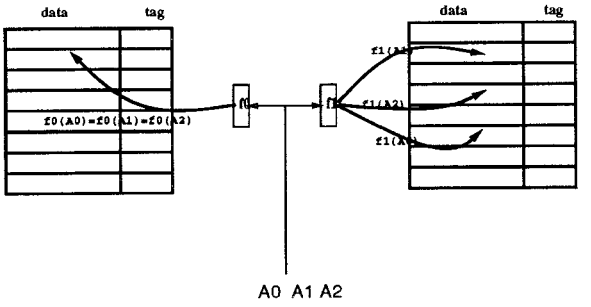

Figure 1: A two-way skewed-associative cache: A0, A1 and A2 compete for the same location in bank 0, but can be present at the same time, as they do not map to the same location in bank 1

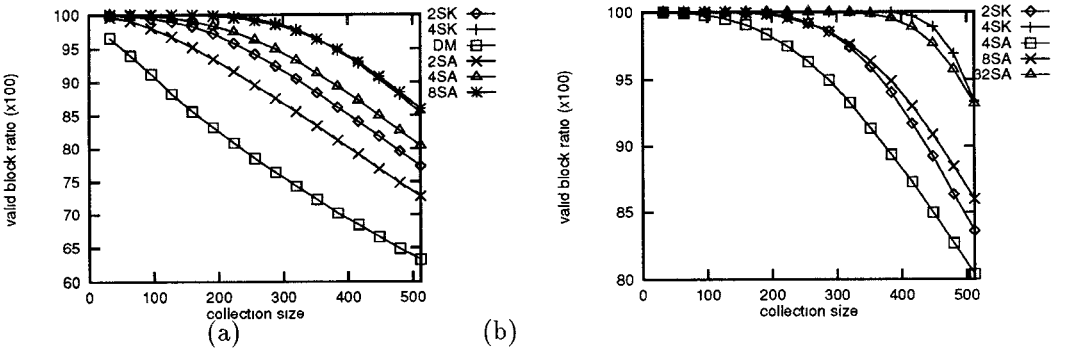

Figure 2: (a) Ratio of valid blocks after one read, (b) Ratio of valid blocks after ten reads

```
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 3: 2D Jacobi loop nest

	block size	8	12	16	20	24
Blocked	data ref.	4054635	3818480	3734266	3655908	3618925
	Ideal execution time	5867338	4961618	4627352	4309654	4157018
Bl & Co	data ref.	4543841	4147926	4001572	3862082	3794911
	Ideal execution time	6341012	5237272	4838252	4463504	4285230

Table 4: Statistics on 120*120 blocked and blocked & copied matrix multiplications for different block sizes

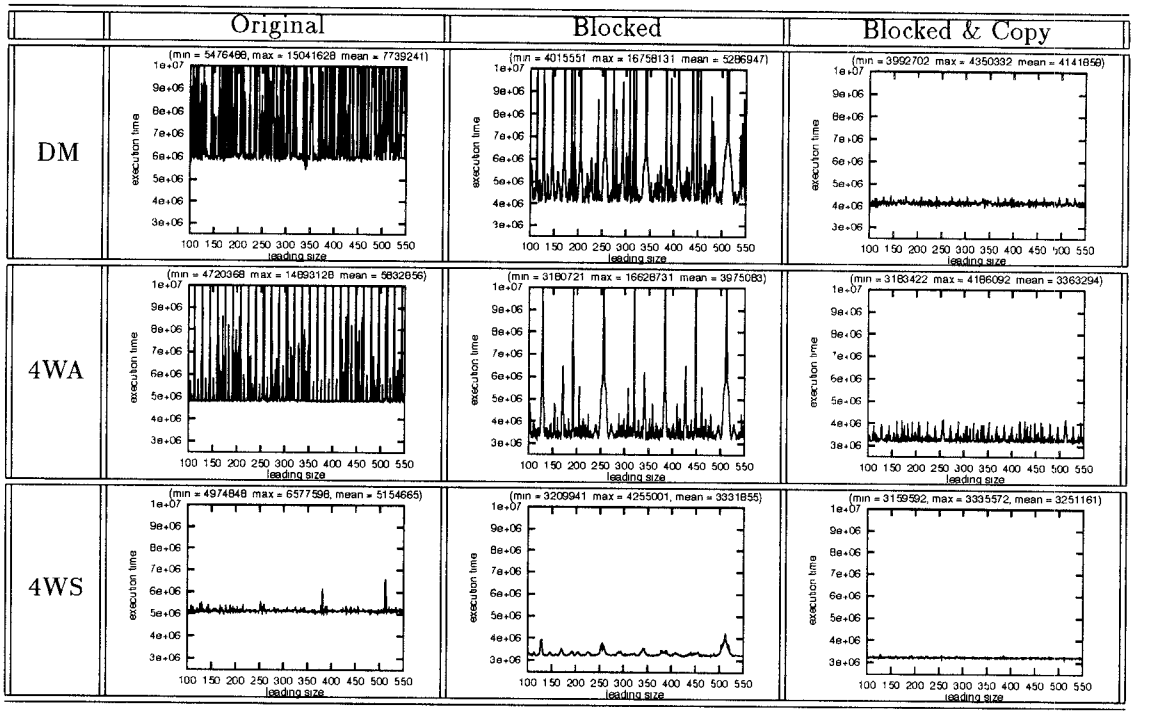

Figure 4: Execution times for 100*100 matrix-matrix multiply

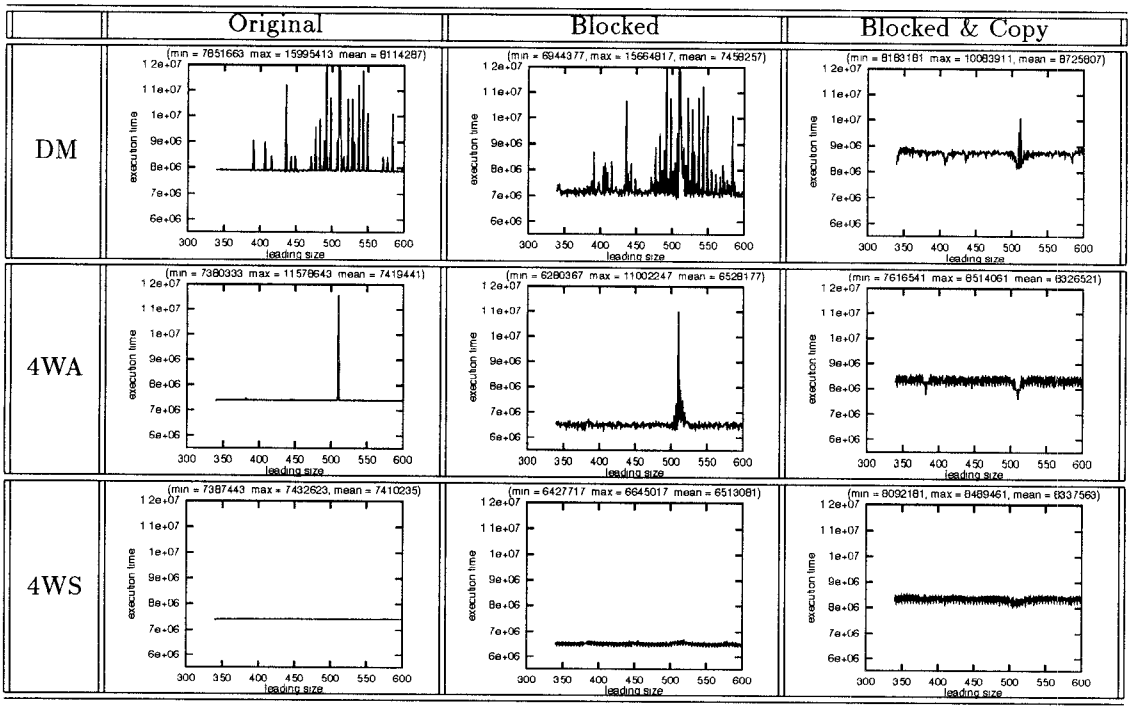

Figure 5: Simulation results for the 2D Jacobi loop

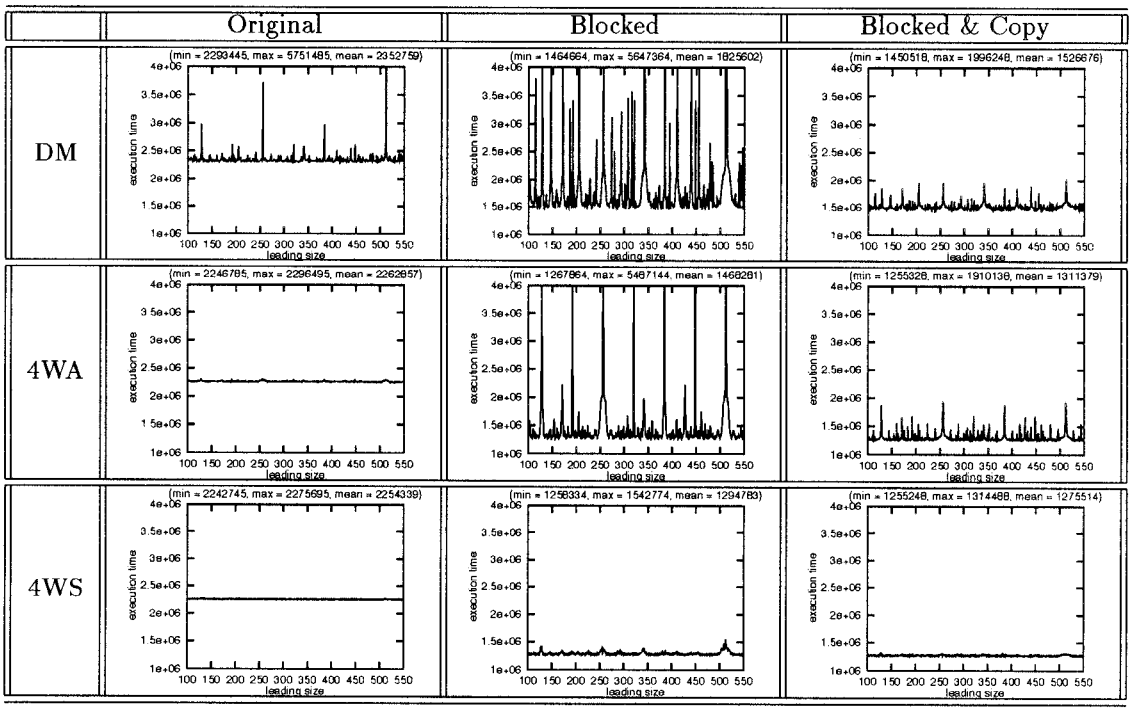

Figure 6: Simulation results for LU

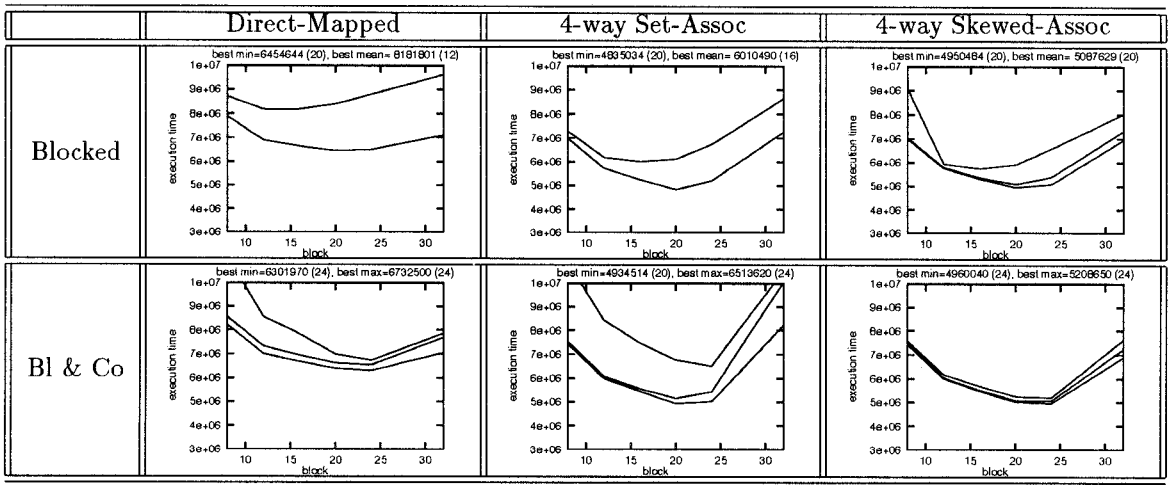

Figure 7: Minimum, average and maximum execution times for various block sizes (8 to 32) and leading sizes (120 to 220) on blocked and blocked & copied matrix multiply