Near-Optimal Padding for Removing Conflict Misses

Xavier Vera[1], Josep Llosa[2], and Antonio González[2]

[1] Institutionen för Datateknik, Mälardalens Högskola
P.O. BOX 883, Västerås, 721 23, Sweden
`xavier.vera@mdh.se`
[2] Computer Architecture Department, Universitat Politècnica de Catalunya-Barcelona
Jordi Girona 1-3, Barcelona, 08034, Spain
{`josepll,antonio`}`@ac.upc.es`

Abstract. The effectiveness of the memory hierarchy is critical for the performance of current processors. The performance of the memory hierarchy can be improved by means of program transformations such as padding, which is a code transformation targeted to reduce conflict misses. This paper presents a novel approach to perform near-optimal padding for multi-level caches. It analyzes programs, detecting conflict misses by means of the Cache Miss Equations. A genetic algorithm is used to compute the parameter values that enhance the program. Our results show that it can remove practically all conflicts among variables in the SPECfp95, targeting all the different cache levels simultaneously.

1 Introduction

Memory performance is critical for the performance of current computers. Memory is organized hierarchically in such a way that the upper levels are smaller and faster. The uppermost level typically has a very short latency (e.g. 1-2 cycles) but the latency of the lower levels may be a few orders of magnitude longer (e.g. main memory latency may be around 100 cycles). Thus, techniques to keep as much data as possible in the uppermost levels are key to performance.

In addition to the hardware organization, it is well known that the performance of the memory hierarchy is very sensitive to the particular memory reference patterns of each program. The reference patterns of a given program can be changed by means of transformations that do not alter the semantics of the program. These program transformations can modify the order in which some computations are performed or can simply change the data layout. *Padding* is an example of the latter family of techniques. Padding is based on adding some dummy elements between variables (inter-variable padding) or between elements of the same variable (intra-variable padding).

Padding has a significant potential to remove cache misses. In fact, it can remove most conflict misses by changing the addresses of conflicting data, and some compulsory misses by aligning data with cache lines. However, finding the

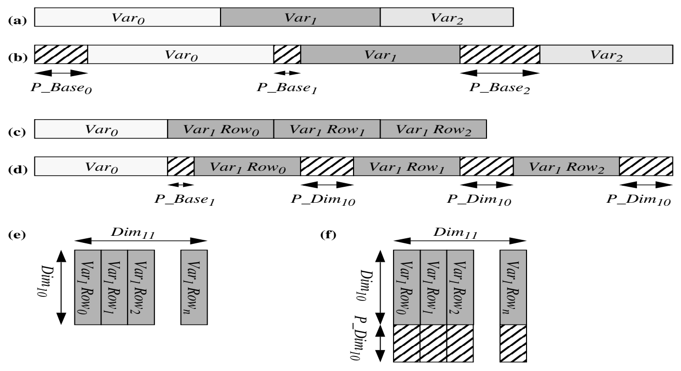


Fig. 1. Data layout: (a) before inter-variable padding, (b) after inter-variable padding (c) before padding, (d) after padding, (e) 2-D array, (f) 2-D array after intra-variable padding

optimal padding for a given program is a very complex task, since the options are almost unlimited and exploring all of them is infeasible. For very simple programs, the programmer intuition may help but in general, a systematic approach that can be integrated into a compiler and can deal with any type of program and cache architecture is desirable. This systematic approach requires the support of a locality analysis method in order to assess the performance of different alternatives.

In this paper, we propose an automatic approach to perform both inter- and intra-variable padding in numeric codes, targeting any kind of multi-level caches. It is based on a very accurate technique to analyze the locality of a program that is known as Cache Miss Equations (CMEs) [6] and a genetic algorithm in order to search the solution space. Earlier, we have proposed techniques to estimate the locality of a possible solution in a very few seconds [2, 21], in spite of the fact that a direct solution to the CMEs is an NP problem. The proposed genetic algorithm converges very fast and although it does not guarantee that the optimal solution is found, we show that after padding, the conflict miss ratio of the evaluated benchmarks is almost negligible. Besides, comparing our method with previous frameworks that address padding [17, 19], it turns out that in 91% of the cases our approach yields better results.

The rest of this paper is organized as follows. Section 2 presents the padding technique and its performance is evaluated in Section 3. Section 4 outlines some related work and compares our method with previous approaches. Finally, Section 5 summarizes the main conclusions of this work.

2 Padding

This section presents our method for guiding both inter- and intra-variable padding. In this paper we refer to the cache size of L1 (primary) cache as C_s. mem_i is the original base address of variable number i (Var_i) and P_Base_i stands for the inter-variable padding between Var_i and Var_{i-1}. dim_{ij} stands for the size of the dimension j of Var_i (D_i is the number of dimensions) and S_i is its size. P_Dim_{ij} is the intra-variable padding applied to dim_{ij}, and P_S_i is the size of Var_i after padding (see Figure 1). We define Δ_i as $P\_S_i - S_i$.

2.1 Inter-variable padding

When inter-variable padding is applied only the base addresses of the variables are changed. Thus, padding is performed in a simple way. Memory variable base addresses are initially defined using the values given by the compiler. Then, we define for each memory variable Var_i, a variable $P\_Base_i,\ i = 0 \ldots k$:

$$0 \leq P\_Base_i \leq C_s - 1$$

Note that padding a variable is equivalent to modifying the initial addresses of the other variables (see Figure 1). Thus, after padding, the memory variable base addresses are computed as follows:

$$BaseAddr(Var_i) = mem_i + \sum_{k=0}^{k \leq i} P\_Base_k$$

2.2 Adding intra-variable padding

The result of applying both inter- and intra-variable padding is that all base addresses and sizes of every dimension of each memory variable may change. They are initially set according to the values given by the compiler. For each memory variable $Var_i, i = 0 \ldots k$ we define a set of variables $\{P\_Base_i, P\_Dim_{ij}\}$, $j = 0 \ldots D_i$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

After padding, memory variable base addresses are computed in the following way (see Figure 1):

$$BaseAddr(Var_i) = mem_i +$$
$$+ \textstyle\sum_{k=0}^{k<i} (P\_Base_k + \Delta_k) + P\_Base_i$$

and the size of the dimensions are:

$$Dim_i(Var_j) = dim_{ji} + P\_Dim_{ji}$$

2.3 Model

For the sake of uniformity in the analysis presented here, we assume that both inter- and intra-variable padding are applied[3]. In presence of a multi-level cache, the cost function to minimize is the miss penalty, which can be estimated as follows:

$$miss\_penalty = \sum_{l} \mu_l * number\_misses_l$$

where μ_l is the latency of the cache level l. Our work focuses in obtaining the values of the variables

$$\{P\_Base_i, P\_Dim_{ij}\}$$

that minimizes the miss penalty. When having only a single level cache, minimizing the miss penalty is the same as minimizing the number of misses.

Let f be the function that represents the miss penalty for each possible value of the padding variables:

$$f \longmapsto miss\_penalty \tag{1}$$

$$f(\underbrace{[0, C_s - 1]}_{P\_Base_0} \times \underbrace{[0, C_s - 1]^{D_0}}_{P\_Dim_{0j}} \times \ldots \times \underbrace{[0, C_s - 1]}_{P\_Base_k} \times \underbrace{[0, C_s - 1]^{D_k}}_{P\_Dim_{kj}}) =$$
$$= f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0}, \ldots, P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

Note that $[0, C_s - 1]^{D_i}$ represents the domain of the different P_Dim_{ij} of the variable Var_i. There is no need to consider larger domains: if two references do not conflict on a cache of size S, they will not conflict on a cache of size nS (larger by a factor of n). Therefore, we use the cache size of the smallest cache in the hierarchy (which in practice is L1).

Our problem can be expressed as follows:

$$MIN \quad f(P\_Base_0, \underbrace{P\_Dim_{0j}}_{D_0} \ldots, P\_Base_k, \underbrace{P\_Dim_{kj}}_{D_k})$$

$$0 \leq P\_Base_i, P\_Dim_{ij} \leq C_s - 1$$

$$i = 0 \ldots k$$

where f is called the *objective function.*

Since f is a pseudo-polynomial function [4], the relationship between padding and the number of misses is nonlinear. P_Base_i and P_Dim_{ij} can take only integer values, thus, our problem can be seen as a nonlinear integer optimization (NLP) one.

One of the challenges in NLP is that some problems exhibit local minima. Algorithms that propose to overcome this problem are named *Global Optimization.* Real functions have been studied deeply [20, 12, 7]. Unfortunately, integer functions are hard to optimize. There are some studies based on {0,1} valued

[3] To apply only inter-variable padding, set all P_Dim_{ij} to 0

```
ALGORITHM:
Supply a population $P_0$
i=1
while (not finish)
   $P_i$=Selection($P_{i-1}$)
   $P_i$=Reproduce($P_i$)
   i=i+1
end
```

Fig. 2. Simple Genetic Algorithm

integer functions [10], but in general, this is a hard and time-consuming problem. Hence, the use of heuristics is necessary. Tabu search [8] obtains promising theoretical results, but only partial implementations have been reported so far. On the other hand, simulated annealing [13] and genetic algorithms [9, 11] have been used for years with very good results.

Our proposal is based on the use of a genetic algorithm to optimize function f. We implemented a direct-search that makes the same number of evaluations as our approach for the sake of comparison. In none of the cases did it yield better results than the genetic algorithm and the miss penalty was 26.9% larger on average.

2.4 Genetic Algorithm

Algorithms for function optimization are generally limited to convex regular functions. However, there are lots of functions that are not continuous, non differentiable or multi-modal. It is common to solve this problem by means of stochastic sampling.

Genetic Algorithms (GAs) [9] are a particular type of stochastic methods, that simulate the evolution of a population. Figure 2 shows the simplest GA. It starts from a random generated population, and it makes the population evolve by means of basic genetic operators (selection, mutation and crossover) [9] applied to individuals of the current population, to produce an improved next generation. The probabilities for crossover and mutation, as well as the size of the initial population, are set experimentally.

Genetic Algorithm Parameters The use of GAs requires the determination of the following issues: chromosome representation, selection function, genetic operators and the termination criteria.

Each individual is made up of a set of chromosomes, which represents the variables. In our work, each individual is one configuration of padding (identified by all the inter- and intra-variable padding factors), and the chromosomes represent one single padding factor. The fitness of those individuals is computed using the objective function (eq. 1). The fittest individual is the one that has a set of padding factors that results in a smallest miss penalty.

Genetic algorithms require the natural parameter set of the optimization problem to be coded as a finite-length string over some finite alphabet such as alphabet {0,1}. Therefore, each chromosome is made up of a sequence of genes from a certain alphabet.

It has been shown that using large alphabets gives better results [15]. Thus, we have used the alphabet $\{0, \ldots, 2^k - 1\}$, where k is the greatest divisor of the $log_2 C_s$ that is lower than $log_2 C_s$. This is the largest value of k that guarantees that a single padding factor consists of at least two genes for every cache size. This is not a restriction because the compilers know the cache size. Thus, this computation can be done automatically.

Example. Let us assume a cache of 32KB. Thus, $log_2(32 \times 2^{10}) = 15$. The set of divisors is $divisors = \{1, 3, 5, 15\}$. Hence, the greatest divisor less than 15 is 5, and we will use the alphabet $\{0, \ldots, 31\}$, representing each single padding with 3 genes. For instance, a padding factor of 10017 is represented by the following three genes:

$$\underbrace{\underbrace{\underline{01001}}_{gene_0=9}\;\underbrace{\underline{11001}}_{gene_1=25}\;\underbrace{\underline{00001}}_{gene_2=1}}_{chromosome}$$

Genetic operators provide the basic search mechanism of the GAs, creating new solutions based on the solutions that exist. The *selection* of individuals to produce successive generations plays an extremely important role. We have adopted one of the selection schemes that gives better results, which is known as *remainder stochastic selection without replacement* [9].

2.5 Implementation of Padding

Given a loop nest, our objective function (f in eq. 1) consists of the CMEs generated in a parameterized way, weighted with the latencies of each cache level. We generate a set of parameterized equations for each cache level, where the parameters are the padding factors. We have developed some techniques that exploit the special characteristics of the CMEs [2] in order to speed-up the process of counting solutions in them. To further reduce the computation cost, we propose to study a subset of the iteration space instead of the whole iteration space [21]. This subset is used to study the L1 cache, and the resulting misses are passed to the following cache levels.

Our experiments have shown that an initial population of size equal to 30 is enough to achieve a good solution. We find that if we set crossover probability to 0.9 and we choose a mutation probability of 0.001, the genetic algorithm gives near-optimal results after 15 generations.

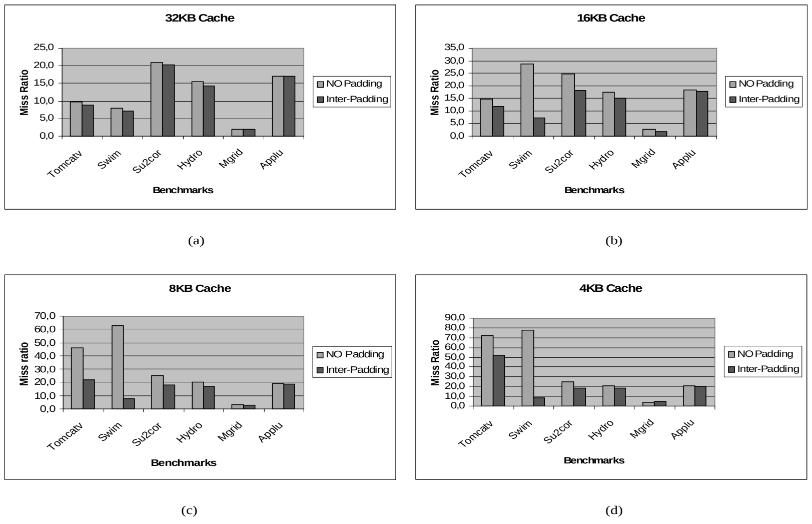


Fig. 3. Miss ratio before and after inter-variable padding for different cache sizes.

3 Performance Evaluation

3.1 Experimental Framework

We have implemented our padding technique for Fortran codes through the Polaris Compiler [16] and the Ictineo library [1].

We evaluate the CMEs using our own polyhedra representation [2]. The size of the sample is set according to a confidence interval of width 0.05 and a 95% confidence [21]. We use the central point of this interval as an estimation of the actual miss ratio.

We have optimized several applications taken from the SPECfp95 that give an insight into how our tool can remove conflict misses. For each application, we have chosen the most time-consuming loop nests that in total represent between the 60-70% of the whole execution time, using the reference input data. Results for different cache architectures, including multi-level caches, are reported. A fully-associative cache has been evaluated as a reference point to estimate the amount of conflict misses that are not removed by the padding technique.

3.2 Experimental Evaluation

Figure 3 shows, for the 6 SPECfp95 programs analyzed, the miss ratio of a direct-mapped cache before and after applying inter-variable padding. Note that the figures for the different cache sizes (32KB, 16KB, 8KB, and 4KB) have different scales. Note also that the SPECfp95 applications have a relatively small working set with respect to current applications. Thus, the results for the smaller cache

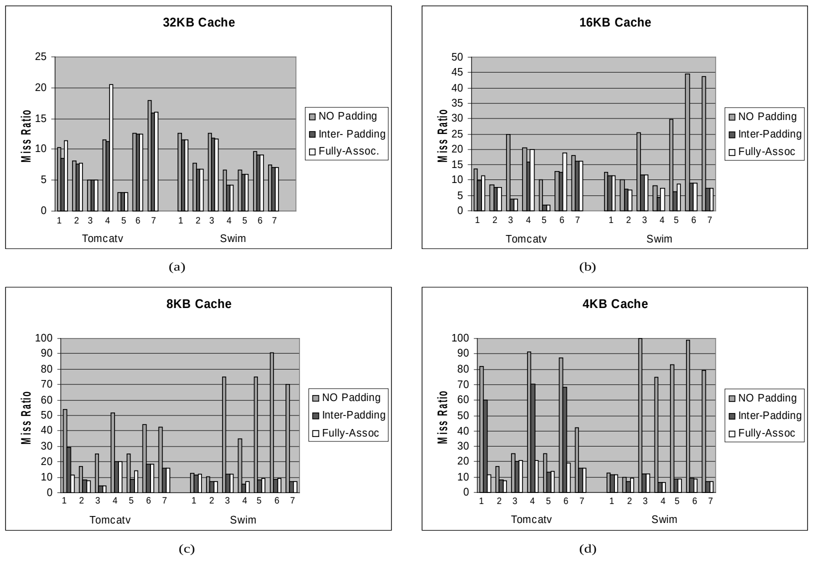


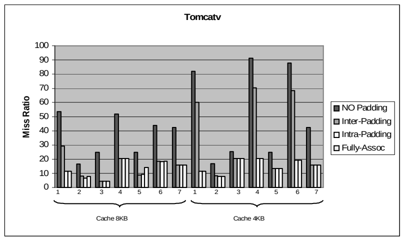
Fig. 4. Miss ratio for the Tomcatv and Swim loop nests before and after inter-variable padding for different cache sizes.

sizes may be more representative of what we can expect today for larger caches and bigger applications. Two sets of programs can be distinguished:

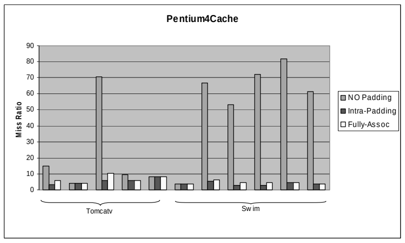
– **Set1** is composed of programs Tomcatv and Swim. The miss ratio of this set of programs is highly affected by cache size. In addition many of the misses are due to conflicts [5].
– **Set2** is composed of programs Su2cor, Hydro, Mgrid, and Applu. The miss ratio of this set of programs is quite insensitive to the cache size. In addition all the programs of this set have practically no conflict misses [5].

Inter-Variable Padding Since the objective of padding is to eliminate conflict misses, for **Set2** we obtain a small improvement when applying inter-variable padding due to the low number of conflicts. Su2cor, which is the program with the highest conflict miss ratio in this set, experiences the highest improvement (e.g 27% miss rate reduction for a 16KB cache). In addition, another source of improvement is that the proposed inter-variable padding technique also aligns the data structures with cache lines, which reduces compulsory misses.

On the other hand, inter-variable padding provides a huge improvement in miss ratio for **Set1**. Note that for both programs, a small improvement is obtained for a 32KB cache (Figure 3.a). This is caused by the fact that almost no conflicts arise for 32KB caches or bigger for these programs due to the relatively small working set of the SPECfp95 applications. However, the smaller the cache



(a)



(b)

Fig. 5. (a) Miss ratio for different Tomcatv loop nests before and after inter- and intra-variable padding (b) Miss ratio for the Tomcatv and Swim loop nests for the Pentium 4 L1 cache

the bigger the miss ratio and the bigger the improvement that inter-variable padding obtains.

For the Swim program, the miss ratio grows from 8.1% to 24.8%, 62.9%, and 77.9% when the cache is reduced from 32KB to 16KB (Figure 3.b), 8KB (Figure 3.c), and 4KB (Figure 3.d) respectively. However, when we apply inter-variable padding, the miss ratio is kept almost constant (7.1%, 7.2%, 7.8% and 8.2% respectively). This is because most of the misses of this program are caused by conflicts between different data structures (inter-variable conflict misses) and the algorithm practically obtains the optimal padding among them.

For the Tomcatv program, the miss ratio also grows significantly when the cache size is reduced (9.5%, 14.8%, 46.0%, and 72.1% respectively for the different cache sizes). In this program, we also obtain a considerable improvement when applying inter-variable padding for caches smaller than 32KB. However, the miss ratio after inter-variable padding varies significantly with the cache size (8.8%, 11.8%, 21.6%, and 52%). This variation is caused by capacity misses that grow when the cache is reduced, and by intra-variable conflict misses (e.g. conflicts among distinct rows and columns of the same array) whose frequency also grows when the cache is reduced. Inter-variable padding does not remove the latter type of conflicts, which are the target of intra-variable padding.

Figure 4 details the miss ratio for the main loop nests of the programs in **Set1** (note again the different scales for the different cache sizes). The figure shows the miss ratio for each loop before and after applying inter-variable padding. It also shows the miss ratio for a fully-associative cache after inter-variable padding.

For the Swim program loop nests 1 and 2 have practically no improvement due to inter-variable padding (excepting a slight improvement due to alignment) because they have no conflict misses. Note also that these two loop nests have almost the same miss rate regardless of the cache size. On the other hand, loop nests 3 to 7 have an extremely large miss ratio. As an extreme case, loop nest 3 has a miss ratio close to 100% for a 4KB cache, which after inter-variable padding is reduced to 11.8%. Note that inter-variable padding removes all the conflict misses for all Swim loops since the miss rate after inter-variable padding and the fully-associative miss rate are practically identical.

The Tomcatv program has several loop nests that deserve special comments. For the 32KB and 16KB, the proposed inter-variable padding technique practically removes all conflict misses. For the 8KB cache, inter-variable padding removes all conflict misses from all loop nests except for loop 1. In this case, inter-variable padding reduces the miss ratio from 53.6% to 29.2% but not all conflict misses are removed since the fully-associative miss ratio is 11.4%. An analysis of this loop shows that there are also intra-conflict misses.

In the case of a 4KB cache, inter-variable padding achieves about the same miss rate as a fully-associative cache for loop nests 2, 3, 5, and 7. As a noticeable case, the miss ratio of loop 7 has been reduced from 42.3% to 15.8%. For the other loop nests there is a significant improvement but the miss ratio is still far from that of the fully-associative cache. An analysis of these three loop nests revealed that most of the remaining misses are intra-variable conflict misses.

Intra-Variable Padding Inter-variable padding cannot remove intra-variable conflict misses. The objective of intra-variable padding is to eliminate them.

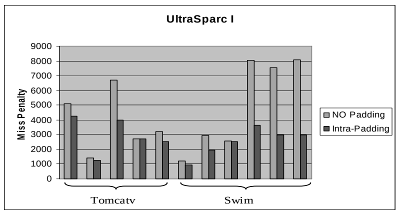
We have shown in the previous section that Tomcatv is the only program of our benchmarks that has a significant intra-variable conflict miss ratio, in particular for caches of 4KB and 8KB. Figure 5.a shows the miss ratio for the different loop nests of the Tomcatv program. The figure shows the miss ratio for each loop after applying inter- and intra-variable padding. It also shows the miss ratio before padding and that of a fully-associative. As we observed before, inter-variable padding does not remove all conflict misses because there are intra-conflict misses. Intra-variable padding achieves about the same miss rate as the fully-associative cache, which means that the proposed padding algorithm removes practically all conflict misses.

Figure 5.b details the miss ratio for the main loop nests of the programs in **Set1** for a 8KB 4-way set associative cache with 64B lines, which is the L1 cache architecture of the new Pentium 4 processor [3]. Intra-variable padding achieves about the same miss ratio as the fully associative cache, reducing the average miss ratio from 62.5% to 4.18% for the Swim program, and from 23.6% to 4.6% for the Tomcatv.

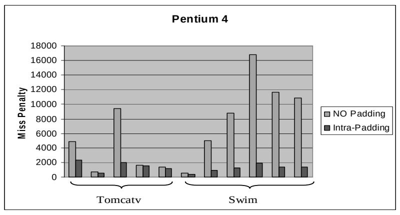
3.3 Multi-level Caches

We experimentally evaluated multi-level padding for uniprocessors. Cache analyses were made for two different configurations:

- UltraSparc I:
  - 16KB, 32B line direct-mapped L1 cache
  - 512KB, 64B line direct-mapped L2 cache
- Pentium 4
  - 8KB, 64B line 4-way set-associative L1 cache
  - 256KB, 128B line 8-way set-associative L2 cache



(a)



(b)

Fig. 6. Miss penalty before and after intra-padding for (a) UltraSparc I (b) Pentium 4 cache architectures.

For both processors, the L2 latency is approximately 3 times the latency of L1, so for computing the cost function, we define the miss penalty in multiples of L1 latency (e.g. a hit has no penalty, a L1 miss adds a penalty of 1, and a L2 miss adds a penalty of 3). We analyzed the most significant loop nests from Tomcatv and Swim, applying intra-variable padding. Figure 6.a shows the miss penalty for the different loop nests assuming a cache architecture such as UltraSparc I. Intra-variable padding reduces 21.7% the average miss penalty for the Tomcatv program, and it reduces the average miss penalty by 50.7% for the Swim program. Figure 6.b details the same information for the Pentium 4 architecture. Again, intra-padding reduces drastically the miss penalty for both programs. In the case of Tomcatv, average miss penalty is reduced by 57.2%, whereas it drops 86.6% in the case of Swim.

Optimization Time Finally, padding has to be performed in a reasonable amount of time in order to be included as an optimization step of a compiler. In our case, it took about 3 minutes to optimize each program[4]. This amount of time can be significantly reduced if the technique is guided by a locality analysis in order to apply padding only to those loop nests that can benefit from it. The locality analysis developed in this work could easily be extended to provide such information.

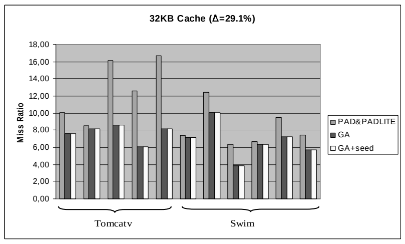
4 Related work

Caches improve the speed of programs by reducing the number of accesses to the slow upper levels of the memory hierarchy. Conflict misses may represent the majority of intra-nest misses and about half of all cache misses for typical programs and cache architectures [14].

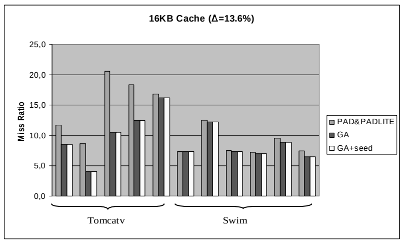
Some padding techniques have been previously proposed by other authors.

Rivera and Tseng [17, 18] propose several simple heuristics that are addressed to eliminate conflicts in some particular cases. They mainly focus on conflicts

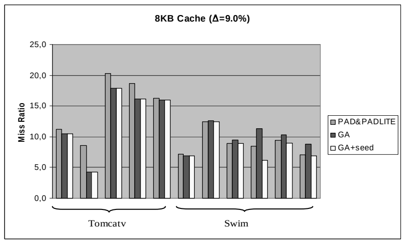
[4] In a Pentium III at 600 MHZ



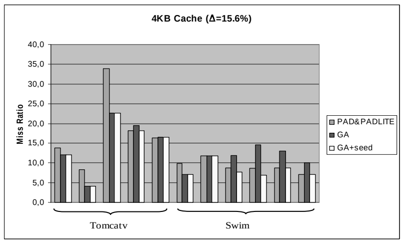
(a)



(b)



(c)



(d)

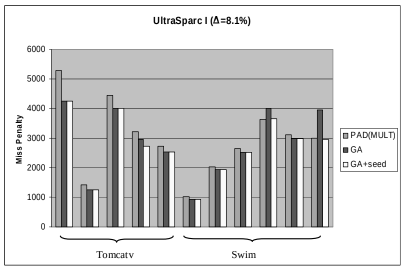
Fig. 7. Comparison with Rivera et al's method for direct-mapped caches. Δ stands for the relative decrease in miss ratios our method achieves compared to theirs.

that occur on every loop iteration, addressing only inter-padding for uniformly generated references (so they can not remove conflict misses for references such as B(i,j) and C(k,j)). On the other hand, they do not use intra-padding to remove cross-interferences. In the case they can not remove all the conflicts, no changes are done to the data layout. Besides, they use the padding algorithm devised to avoid conflict misses for direct-mapped caches to remove conflict misses for set-associative caches, without taking in account that interferences arise in different situations for different cache architectures. A set contention in a set-associative cache does not mean there is an interference. They presented an extension of this work targeting multi-level caches [19].

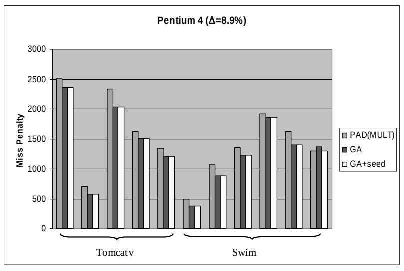
Figure 7 and Figure 8 compare their method with ours. We have studied all the main loop nests of the programs in **Set1** (see Section 3.2), which are the ones that suffer heavily from conflict misses.

Figure 7 compares both methods for 32KB, 16KB, 8KB and 4KB direct-mapped caches. Notice different scales for each chart. First column presents the miss ratios obtained running Rivera et al's method. We use the best result yielded by their two approaches PAD and PADLITE. The second column shows the miss ratios obtained by our approach. GA performs better in all the cases for 32KB and 16KB caches. However, we observe that in some cases Rivera et al's heuristics obtain better results when studying 8KB and 4KB caches.

In order to improve the population in successive iterations, the presence of good individuals in the first population may help. Thereby, we include in the initial solution two individuals (seeds) that represent the original solution



(a)



(b)

Fig. 8. Comparison with Rivera et al's method for multi-level caches. Δ stands for the relative decrease in miss penalty our method achieves compared to theirs.

provided by the compiler and the one obtained by running PADLITE [17]. The third column presents the results for this variant (called GA+seed). It gives better results for all cache configurations, yielding 29.1%, 13.6%, 9% and 15.6% smaller miss ratios for the 32KB, 16KB, 8KB and 4KB caches respectively.

Finally, we compare the different padding techniques for multi-level caches. Figure 8 shows the miss penalty for UltraSparc I and Pentium 4 cache architectures. Our method improves the miss penalty, compared to Rivera et al's method, by 8.1% and 8.9% for UltraSparc I and Pentium 4 architectures respectively.

Ghosh, Martonosi and Malik [6] propose a padding technique for direct-mapped caches based on using the CMEs for conflicting arrays that have the same column size. Their technique finds the optimal padding if there is a padding such that the total number of replacement misses after padding is zero. However, if such a padding does not exist, their technique does not provide any solution. Note that replacement misses include both conflict and capacity misses and one may expect the case where replacement misses cannot be decreased up to zero to be common. In their experiments, this only happens for one out of the seven loops examined but most of their benchmarks are small kernels.

Our technique differs and improves these two previous approaches in the fact that it is a technique to search the solution space for the optimal padding, for any type of reference pattern that corresponds to affine references. It always produces a padding scheme that reduces conflict misses and usually is very close to the optimal. It is not targeted to avoid conflicts in some particular cases but it considers any type of conflicts, using both inter- and intra-padding to remove self- and cross-conflicts. Besides, our algorithm works fine for both direct-mapped and set-associative caches, generating the best padding scheme for each kind of architecture.

Recently, Vera and Xue [22] have presented a method that extends the CMEs to further analyze whole programs. We believe that our padding approach can be easily adapted to this new analysis technique. In that way, the padding factors could be optimized at a global program level considering the interactions of the different loop nests.

5 Conclusions

Cache memory performance is critical for the efficient execution of numerical applications. Padding is a program transformation that reduces conflict misses. In this work, we have proposed the use of genetic algorithms in order to perform near-optimal padding.

The evaluations show that, for the programs that have conflict misses, we achieve a significant improvement. For instance, for a 8KB 4-way associative cache, which is the L1 cache of the new Pentium 4 processor [3], we can reduce the miss ratio of the Swim program from 62.5% to 4.18% and the miss ratio of the Tomcatv program from 23.6% to 4.6%. Furthermore, the miss penalty for Pentium 4 is reduced by 79.27%. Besides, for the programs without conflict misses padding slightly reduces the compulsory misses due to a better alignment of arrays with cache lines.

Finally, an exhaustive evaluation of the programs with a high number of conflict misses reveals that the proposed technique practically removes all the conflict misses for all the loops analyzed, both inter- and intra-variable conflicts.

6 Acknowledgments

This work has been supported by the ESPRIT project MHAOTEU (EP 24942) and the CICYT project 511/98. We would like to thank the anonymous referees for providing helpful comments in earlier drafts of this paper.

References

1. E. Ayguadé et al. *A uniform internal representation for high-level and instruction-level transformations.* UPC, 1995.
2. N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.
3. D. Carmean. *Inside the Pentium 4 Processor Micro-Architecture (www.intel.com/pentium4)*, 2000.
4. P. Clauss. Counting solutions to linear and non-linear constraints through Ehrhart polynomials. In *ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, Philadelphia, 1996.
5. A. Fernández. A quantitative analysis of the SPECfp95. Technical Report UPC-DAC-1999-12, Universitat Politècnica de Catalunya, March 1999.
6. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.
7. Gill, Murray, and Wright. *Practical optimization.* Academic Press, 1981.
8. Glover and Laguna. *Tabu search.* Kluwer, 1997.

9. D. Goldberg. *Genetic algorithms in search, optimizations and machine learning.* Addison-Wesley, 1989.
10. Hansen, Jaumard, and Mathon. Constrained nonlinear 0-1 programming. *ORSA Journal on Computing*, 1995.
11. J. Holland. *Adaptation in natural and artificial systems.* The University of Michigan Press, Ann Arbor, 1975.
12. Host, Pardalos, and Thoai. *Introduction to global optimization.* Kluwer, 1995.
13. Kirkpatrick, Gelatt, and Vecchi. Optimization by simulated annealing. *Science 220*, 1983.
14. K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. of VII Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996.
15. Z. Michalewicz. *Genetic algorithms+Data structures=Evolution Programs.* Springer-Verlag, 1994.
16. D. Padua et al. *Polaris developer's document*, 1994.
17. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.
18. G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *ACM Internacional Conference on Supercomputing (ICS'98)*, 1998.
19. G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Supercomputing (SC'99)*, 1999.
20. Torn and Zilinskas. *Global optimization.* Springer-Verlag, 1989.
21. X. Vera, J. Llosa, A. González, and C. Ciuraneta. A fast implementation of cache miss equations. In *8th International Workshop on Compilers for Parallel Computers (CPC'00)*, 2000.
22. X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *International Symposium on High-Performance Computer Architecture (HPCA 8)*, Cambridge, Feb. 2002.