# Optimizing Program Locality Through CMEs and GAs

Xavier Vera
Institutionen för Datateknik
Mälardalens Högskola
Västerås, Sweden
xavier.vera@mdh.se

Jaume Abella, Antonio González, Josep Llosa
Computer Architecture Department
Universitat Politècnica de Catalunya-Barcelona
Barcelona, Spain
{jabella, antonio, josepll}@ac.upc.es

## Abstract

*Caches have become increasingly important with the widening gap between main memory and processor speeds. Small and fast cache memories are designed to bridge this discrepancy. However, they are only effective when programs exhibit sufficient data locality.*

*Performance of memory hierarchy can be improved by means of data and loop transformations. Tiling is a loop transformation that aims at reducing capacity misses by exploiting reuse at the lower levels of cache. Padding is a data transformation targeted to reduce conflict misses.*

*This paper presents an accurate cost model which makes use of the Cache Miss Equations (CMEs) to guide tiling and padding transformations. It describes misses across different hierarchy levels and considers the effects of other hardware components such as branch predictors. We combine the cost model with a genetic algorithm (GA) to select the tile and pad factors that enhance the program.*

*Our results show that this scheme is useful to optimize programs' performance. When compared to previous works, we observe that with a reasonable compile-time overhead, our approach obtains significant performance improvements for all studied kernels on a variety of architectures.*

## 1. Introduction

With ever-increasing clock rates and the use of new architectural features, the speed of processors increases dramatically every year. Unfortunately, memory latency does not decrease at the same pace, being a key obstacle to achieve high IPC. The basic solution that almost all systems rely on is the cache hierarchy.

While caches are useful, they are effective only when programs exhibit sufficient data locality in their memory accesses. Numerical applications tend to operate on large data sets, and usually present a large amount of reuse. However, this reuse may not translate to locality since caches can only hold a small fraction of the data accessed.

### 1.1. Cache Compiler Optimizations

Memory is organized hierarchically in such a way that the lower levels are smaller and faster. In order to fully exploit the memory hierarchy, one has to ensure that most memory references are handled by the lowest levels of cache. Various hardware and software approaches have been proposed lately for increasing the effectiveness of memory hierarchy. Software-controlled prefetching [23] hides the memory latency by overlapping a memory access with computation and other accesses. Other useful optimizations are loop transformations such as tiling [4, 7, 20, 35] and data transformations such as padding [6, 17, 25, 27, 30]. In all cases, a fast and accurate assessment of a program's cache behavior at compile time is needed to make an appropriate choice of transformation parameters.

Unfortunately, cache memory behavior is very hard to predict. Thus, current approaches are based on simple models (heuristics) for estimating locality [5, 7, 20, 25, 27]. However, modern architectures have very complex internal organization, with different levels of cache, branch predictors, etc. Such models provide very rough performance estimates, and in practice, are too simplistic to statically select the best optimizations.

Tiling has been shown to be useful for many algorithms in linear algebra. By restructuring the loop and changing the order in which memory references are executed, it reuses data in the faster levels of the hierarchy; thus it reduces the average latency. Nevertheless, finding the optimal tile sizes is a very complex task. The solution space is huge, and exploring all possible solutions is infeasible. A number of algorithms have been proposed [4, 7, 20, 27, 35], which are based on simple cost models that only consider one array access and the first level of cache.

We introduce a method that aims at optimizing overall performance via tiling and padding. It combines an approach to choose the best tile and pad factors in concert. Our approach makes use of a very precise cost model that allows us to consider all different levels of the memory hierarchy. Furthermore, we consider the performance cost of the miss-predicted branches. We present results for a collection of kernels drawn from several related papers [4, 7, 20, 27, 35]. For the sake of concreteness, we report results for a set of modern processors that represent current architectural paradigms. We have chosen the Pentium-4 (CISC), Alpha-21264 (RISC), and Itanium[1] (EPIC). The centerpiece of the proposed method is an accurate cost model combined with a genetic algorithm. Although it is not shown in this paper, this approach can be used to drive other program transformations oriented to enhance data locality.

### 1.2. An Overview

This paper proposes a new method to perform loop tiling combined with padding for numeric codes. We use a static data cache analysis that considers different levels of cache. Moreover, it considers the cost of the branch instructions accordingly to the outcome from the branch predictor.

We first start describing data reuse using the well-known concept of reuse vectors [35]. We implemented Ghosh *et al's* Cache Miss Equations (CMEs) [10] to compute the locality of a program, extending its applicability to deal with multi-level caches. This allows us to have a precise model that describes cache memory behavior across different levels. Once information about read and write misses for all different levels is obtained, we set up the cost model function. We consider the relative costs of each memory level and the cost of miss-predicted branches, which tunes the cost function for improving execution time. Finally, we use a genetic algorithm (GA) for traversing the solution space in order to determine all tile and pad factors at the same time.

We have implemented our system in the SUIF2 compiler. We identify high level information (such as array accesses and loop constructs), that is used to model the cache behavior. Then, the GA generates different possible combinations of tile and pad factors, that are analyzed by the cost model function. Finally, the best parameters are fed back and used for generating the optimized code.

The rest of the paper is organized as follows. Section 2 reviews our method to describe data locality, and introduces tiling and padding techniques. Section 3 describes our cost model for estimating performance. Section 4 presents the experimental framework, and Section 5 compares our results against the state-of-the-art techniques. Section 6 contains some related work that aims at optimizing the cache behavior statically. Finally, we conclude and outline a road map to future extensions in Section 7.

[1]We do not consider the 16KB L1 cache since it only holds integers.

## 2. Improving Locality

In this section, we review the concepts of data reuse and data locality. We first discuss some important concepts related to data cache analysis and how we model different cache levels. Finally, we introduce loop tiling and padding, and explain how they can be used to improve data locality.

Understanding data reuse is essential to predict cache behavior. *Reuse* happens whenever the same data item is referenced multiple times. This reuse results in *locality* if it is actually realized; reuse will result in a cache hit if no intervening reference flushes out the datum.

Given that, a static data cache analysis can be split into the following steps:

1. *Reuse Analysis* describes the intrinsic data reuse among all different memory references.[2]

2. *Data Locality Analysis* describes the subset of reuses that actually results in locality.

In the following, we describe each step in detail.

### 2.1. Reuse Vectors

In order to describe data reuse, we use the well-known concept of reuse vectors [35]. They provide a mechanism for summarizing repeated memory accesses within a perfect loop nest.

Trying to determine all iterations that use the same data is extremely expensive. Thus, we use a concrete mathematical representation that describes the direction as well as the distance of the reuse in a methodical way. The shape of the set of iterations that uses the same data is represented by a *reuse vector space* [35]. Whereas self reuse (both spatial and temporal) and group-temporal reuse is computed in an exact way, group-spatial reuse is only considered among uniformly generated references (UGRs), this is, references whose array index expressions differ at most in the constant term [9].

Figure 1(b) presents the reuse vectors for the references in our running example shown in Figure 1(a). The reference $a(i,j)(W)$ may reuse from the same datum (hence, temporal reuse) that $a(i,j)(R)$ (hence, group reuse) accessed at the same iteration. Reference $c(k,j)$ is associated with the self-spatial reuse vector (0,0,1), since it may reuse the same cache line (thus, spatial reuse) that it accessed one iteration before of the innermost loop nest. The other reuse vectors can be understood in a similar way.

In the case that a reuse vector is not present, we assume there is not reuse, thus, there is not locality.

[2]We use *memory reference* to note a static read or write in the program. A particular execution of that read or write at run-time is a *memory access*.

```
REAL*8 A(N,N), B(N,N), C(N,N)

DO 2 i=1,N
   DO 2 j=1,N
      DO 2 k=1,N
         a(i,j)=a(i,j)+b(i,k)*c(k,j)
2 CONTINUE
```

(a) FORTRAN code.

| Reusing reference | Reused reference | | Reuse Vector |
|---|---|---|---|
| a(i,j) (*R*) | Self Spatial | | (1,0,0) |
| | Self Temporal | | (0,0,1) |
| b(i,k) | Self Spatial | | (1,0,0) |
| | Self Temporal | | (0,1,0) |
| c(k,j) | Self Spatial | | (0,0,1) |
| | Self Temporal | | (1,0,0) |
| a(i,j) (*W*) | a(i,j) (*R*) | Group Spatial | (0,0,0) |
| | Self Spatial | | (1,0,0) |
| | Self Temporal | | (0,0,1) |

(b) Computed reuse vectors. *R* stands for READ, *W* for WRITE.

**Figure 1. The IJK matrix multiplication and the reuse vectors that describe its reuse.**

## 2.2. Data Locality

Data locality is the subset of reuse that is realized; i.e., reuse where the subsequent use of data results in a hit in the considered cache level. To discover whether a reuse translates to locality we need to know all data brought to the cache between the two accesses (this implies knowledge about loop bounds and memory access addresses) and the particular cache architecture we are analyzing.

CMEs [10] are mathematical formulas that provide a precise characterization of the cache behavior for perfect nested loops consisting of straight-line assignments. Based on the description of reuse given by reuse vectors, some equations are set up that describe those iteration points where the reuse is not realized. Recently, Vera and Xue [34] have proposed a new CMEs that are capable of analyzing whole programs. It takes into account imperfect loop nests, data-independent conditionals and subroutines.

For every reuse vector, two types of CMEs are generated:

- **Compulsory equations.** Compulsory equations represent the first time a memory line is brought into the cache.

- **Replacement equations.** Given a reference, replacement equations represent the interferences with any other reference. For each pair of references ($R_A$ and $R_B$), the following expression gives the condition that determines whether they are mapped onto the same cache set:

$$\begin{aligned} Cache\_Set(\vec{\imath})_{R_A} &= Cache\_Set(\vec{\jmath})_{R_B} \\ \vec{\jmath} &\in \mathcal{I} \end{aligned}$$

where $\mathcal{I}$ represents the iteration points between $\vec{\imath}$ (the current one) and the iteration point from which $R_A$ reuses. This condition is expanded into a set of equations for each reuse vector.

Each equation represents a real polyhedron, whose integer points are potential cache misses (the number of points is the number of potential cache misses).

Given a reference, all the iteration points can be tested independently [10]. In order to know if an iteration point $\vec{\imath}_0$ results in a miss we need to know when it fulfills the CMEs. This problem is equivalent to finding out whether, after substituting the iteration point in the CMEs, the resulting polyhedron is non-empty, which is an NP problem.

Even though generating the equations is linear in the number of references, solving them can be very time consuming. We use our previous work, which presents probabilistic methods based on sampling [33] to solve the equations in a fast and accurate way.

### 2.2.1 CMEs for Multi-level Caches

We now briefly discuss how we extend our analysis to memory hierarchies with more than one level.

For these architectures, we have to analyze differently memory references depending on the cache level they are accessing. For that purpose, a set of equations that describe precisely the relationship among the iteration space, array sizes and cache parameters is set up for each of the cache levels.

When analyzing potential cache set contentions, only memory accesses that miss in lower cache levels are considered. Thus, we can see the equations for each level as filters, where only those memory accesses that miss are analyzed in further levels.

## 2.3. Tiling and Padding Overview

In addition to the hardware organization, it is well known that performance of memory hierarchy is very sensitive to the particular memory reference patterns of each program. In order to enhance locality, transformations that do not alter the semantics of the program attempt to modify the order

```
REAL*8 A(N,N)
REAL*8 B(N,N)
REAL*8 C(N,N)

DO 2 ii=1,N,T1
   DO 2 jj=1,N,T2
      DO 2 k=1,N
         DO 2 i=ii, min(ii+T1-1, N)
            DO 2 j=jj, min(jj+T2-1, N)
               a(i,j)=a(i,j)+b(i,k)*c(k,j)
2 CONTINUE
```

(a) Tiled Matrix Multiply.

```
REAL*8 A(N+Pad1,N+Pad2)
REAL*8 B(N+Pad3,N+Pad4)
REAL*8 C(N+Pad5,N+Pad6)

DO 2 ii=1,N,=T1
   DO 2 jj=1,N,T2
      DO 2 k=1,N
         DO 2 i=ii, min(ii+T1-1, N)
            DO 2 j=jj, min(jj+T2-1, N)
               a(i,j)=a(i,j)+b(i,k)*c(k,j)
2 CONTINUE
```

(b) Tiled and Padded Matrix Multiply.

**Figure 2. Matrix multiply algorithm after applying tiling and padding.**

in which computations are performed, or simply change the data layout.

Loop tiling combines strip-mining with loop interchange for increasing the effectiveness of memory hierarchy. Figure 1(a) shows the code for the IJK matrix multiplication of `NxN` arrays kernel, which we use as our running example. We present the tiled version, with tile sizes `T1` and `T2`, in Figure 2(a). Loop tiling basically consists of two steps [35]. The first one consists in restructuring the code to enable tiling those loops that carry reuse. The second one is to select the tile factors that maximize locality. It is the latter step that is sensitive to the characteristics of the cache memory considered. Due to hardware constraints, caches have limited associativity, which may cause cache lines to be flushed out of the cache before they are reused despite sufficient capacity in the overall cache.

Unlike loop tiling, padding modifies the data layout to eliminate conflict misses. It changes the data layout in two different ways. Inter-padding modifies the base address of the arrays, whereas intra-padding changes the size of array dimensions. Figure 2(b) shows our running example after tiling and intra-padding all array dimensions.

## 3. Performance Modeling

In this section, we introduce our cost model. We first describe how we model loop tiling, padding and branch predictor behavior. Then, we describe our cost function to estimate performance. Finally, we reason the use of a GA to traverse the solution space.

### 3.1. Tiling and Padding Model

We want to improve data locality through loop tiling and padding. We focus on removing capacity misses by means of loop tiling, whereas we use padding to eliminate those conflict misses that loop tiling cannot remove.

We present a compiler strategy that combines both optimizations at the same time by implementing the CMEs in a parameterized way. The domains of the parameters are set as follows. Assuming normalized loops, each tile factor will range between 1 and the upper bound of the respective loop nest. For the pad factors, we use the cache size of the smallest cache in the hierarchy (which in practice is L1) as the domain [32].

Our measure of locality is the number of read and write misses for each cache level. More formally stated, given a loop nest $L$ with $n$ normalized enclosing loops $L = \{l_1, \ldots, l_n\}$, and a set of tile and pad factors $F = \{T_1, \ldots T_k, P_1, \ldots P_t\}$, we define a function $MCost(L, F)$:

$$MCost(L, F) \quad \longmapsto \quad (rm_{L1}, wm_{L1}, \ldots, rm_{Lu}, wm_{Lu})$$

where $rm$ ($wm$) stands for read (write) misses and $Li$ for the i-th level of cache.

The following function characterizes the locality of the optimized version of our running example (see Figure 2(b)), where $C_s$ stands for the first level cache size of the architecture being analyzed:

$$MCost(L, \{\underbrace{[1, N]}_{T_1}, \underbrace{[1, N]}_{T_2}, \underbrace{[0, C_s - 1]}_{P_1}, \ldots, \underbrace{[0, C_s - 1]}_{P_6}\})$$

### 3.2. Branch Model

As the issue rate and depth of pipelining of high performance superscalar processors increase, the importance of control-flow speculation becomes more vital to achieve the potential performance of current processors. There is a serious performance degradation in deep-pipelined machines caused by branch miss-predictions [3]. Modern branch predictors work fairly well for loops, since they usually miss at most once over all iterations of a loop. However, current processors still miss-predict in this situation frequently.

Tiling must be applied carefully because it may increase overhead due to the tiled code complexity. Besides, the

```
INPUT
  $L$ = $\{l_1, \ldots, l_n\}$ a loop nest
  $M$ = $\{M_1, \ldots, M_u\}$ a memory hierarchy with $u$ levels
  $F$ = $\{T_1, \ldots, T_n, P_1, \ldots, P_t\}$ a set of tile and pad factors
OUTPUT
  LoopCost($L$,$M$,$F$) = number of estimated cycles

ALGORITHM
  $< rm_{M1}, wm_{M1}, \ldots, rm_{Mu}, wm_{Mu} >$ = $MCost(L, F)$
                                  miss_predictions = $MissPred(L)$
                               LoopCost($L$,$M$,$F$) = $\sum_{l=1}^{l \leq u} (\mu_{Rl} * rm_{Ml} + \mu_{Wl} * wm_{Ml})$
                                                   + $\mu_{MP}$*miss_predictions
WHERE
  $\mu_{Rl}$ = cost of a read miss in level $l$
  $\mu_{Wl}$ = cost of a write miss in level $l$
  $\mu_{MP}$ = cost of a miss-predicted branch
```

**Figure 3. LoopCost algorithm.**

extra levels of loops may lead to larger number of miss-predicted branches. In order to avoid a large performance degradation due to branch miss-predictions, we incorporate into our model the number of possible miss-predicted branches.

Let $L$ be a loop nest with $n$ normalized enclosing loops $L = \{l_1, \ldots, l_n\}$, with upper bounds $\{U_1, \ldots, U_n\}$ respectively. If there is a branch miss-prediction for each loop execution, the number of miss-predicted branches is:

$$MissPred(L) = 1 + \sum_{j=2}^{j \leq n} \prod_{i=1}^{i<j} U_i$$

**Example.** Let us consider our running example when {`N=100, T1=20, T2=20`} (the pad factors are irrelevant for computing the number of miss-predicted jumps). The number of miss-predicted branches for the non-tiled version (see Figure 1(a)) is $1 + \sum_{j=2}^{j \leq 3} \prod_{i=1}^{i<j} 100 = 1 + 100 + 10^4$.

The number of miss-predictions for the tiled version (see Figure 2(a)) will be $1 + \sum_{j=2}^{j \leq 5} \prod_{i=1}^{i<j} U_i = 1 + \lceil\frac{100}{20}\rceil + \lceil\frac{100}{20}\rceil * \lceil\frac{100}{20}\rceil + \lceil\frac{100}{20}\rceil * \lceil\frac{100}{20}\rceil * 100 + \lceil\frac{100}{20}\rceil * \lceil\frac{100}{20}\rceil * 100 * 20 = 1 + 5 + 25 + 2500 + 5 * 10^4$.

This is, over five times as many as the original program. Thus, in order to have some speedup, we should have an important reduction in number of misses to compensate this overhead.

## 3.3. Cost Model

Once we account for misses across all different levels and loop tiling overhead due to miss-predicted branches, we can calculate the loop cost. In Figure 3, we give a detailed description of our cost model function, *LoopCost*.

| | $\mu_{R1}$ | $\mu_{W1}$ | $\mu_{R2}$ | $\mu_{W2}$ | $\mu_{MP}$ |
|---|---|---|---|---|---|
| Pentium-4 | 24 | 24 | 150 | 150 | 20 |
| Alpha-21264 | 6 | 6 | 84 | 84 | 7 |
| Itanium | 21 | 21 | 128 | 128 | 15 |

**Table 1. Weights to calculate *LoopCost* for a particular architecture.**

*MCost* calculates the locality for the loop nest $L$ given the set of tile and pad factors, i.e., the number of read and write misses for each cache level. *MissPred* estimates the number of miss-predicted branches that the branch predictor may incur in when executing loop nest $L$. *LoopCost* then calculates the total cost of executing $L$. It simply adds up all different misses and the number of miss-predictions, weighting each value by its relative cost.

In Table 1, we give the relative costs used to model the architectures we used in our study (see Table 2). Values have been obtained from vendors' specifications. We assume a memory latency of 160ns.

## 3.4. Compiler Strategy

The main objective of a compiler strategy is to determine which transformations to apply. In our case, our main concern is to decide which tile and pad factors yield the best results. In this subsection, we explain how we choose the parameter values guided by our cost model.

We want to find a set of tile and pad factors that minimize the *LoopCost* of a loop nest $L$. More formally stated:

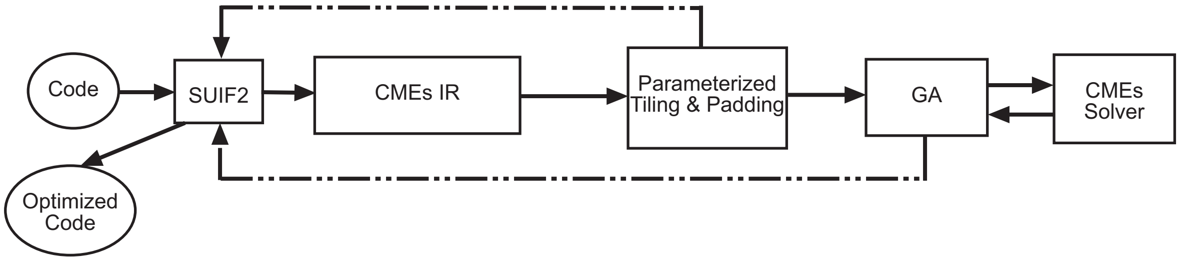


**Figure 4. Optimizing framework.**

$$MIN \quad LoopCost(L, M, F)$$
$$1 \leq T_k \leq U_k$$
$$0 \leq Pad_t \leq C_s - 1$$

where *LoopCost* is called the *objective function*.

The relationship between tiling, padding and the number of misses is not linear. Moreover, recently it has been proved that *"Unless P==NP there is no efficient optimal algorithm for data placement that minimizes the number of misses"* [24].

Since both tile and pad factors can take only integer values, our problem is considered as a nonlinear integer optimization problem (NLP).

One of the challenges in NLP is that some problems exhibit local minima. Proposed algorithms to overcome this problem are named *Global Optimization*. Real functions have been studied deeply [11, 16, 31]. Unfortunately, integer functions are hard to optimize. There are some studies based on {0,1} valued integer functions [14], but in general, this is a hard and time-consuming problem. Hence, the use of heuristics that traverse the solution space is necessary. Tabu search [12] obtains promising theoretical results, but only partial implementations have been reported so far. On the other hand, simulated annealing [18] and genetic algorithms [13, 15] have been used for years with very good results.

### 3.4.1 Why a Genetic Algorithm?

The majority of research in optimization via high-level restructuring has relied on smart heuristics and very simple models [4, 7, 20, 25, 30, 35], which manage to improve program performance significantly. However, current results in compiler theory [24] point out two important practical issues: (i) the use of heuristics is a must, and (ii) the loss of information is critical to find a good solution.

Our proposal is based on the use of a very accurate cost model, thus reducing the loss of information. Then, we use a heuristic, in this case a genetic algorithm, to optimize function *LoopCost*. According to Petrank and Rawitz [24], the only way to evaluate the potential of our method is comparing it with previous ones. Our experimental results show that with a small and reasonable compile-time overhead, our method outperforms all other previous approaches, for all benchmarks running on a variety of modern architectures.

For a detailed description of the decisions taken for implementing the GA, we refer the interested reader to our technical reports [1, 32].

| Processor | Freq. | L1 (*Cs*,*Ls*,*K*) | L2 (*Cs*,*Ls*,*K*) |
|---|---|---|---|
| Pentium-4 | 1.6GHz | (8,64,4) | (512,128,8) |
| Alpha-21264 | 525MHz | (64,64,2) | (4096,64,4) |
| Itanium | 800MHz | (96,64,6) | (2048,64,4) |

**Table 2. Processors used for the experimentation. *Cs* stands for cache size in KB, *Ls* stands for cache line size in bytes, and *K* stands for the degree of associativity.**

## 4. Experimental Framework

Figure 4 depicts the framework used in our experiments. We implement the analysis as general as possible, so the compiler is written using the SUIF2 internal representation, which can be generated from different front-ends. We use SUIF2 to collect all information about memory accesses and control flow.

The core block is the one that computes the equations and solves them, which describes the cache behavior. We have implemented the CMEs following the techniques outlined in our previous work [2, 33], choosing a confidence interval width of 0.1 and a 90% confidence which proved to be enough to guide our optimizations.

The genetic algorithm has been implemented following techniques described in our technical reports [1, 32]. Our experimental results show that an initial population of 30 individuals, with a crossover and mutation probability of 0.9 and 0.005 respectively, give near-optimal results after 15 generations.

| Name | Description |
|---|---|
| MATMUL | Matrix multiplication |
| MATVEC | Matrix vector multiplication |
| T2D | 2D matrix transposition |
| ADI | 2D ADI integration |
| VPENTA | Invert 3 pentadiagonals |

(a) Description of the kernels.

| Name | Size 1 | Size 2 |
|---|---|---|
| MATMUL | 400+50i | 1000+50i |
| MATVEC | 500+43i | 1000+43i |
| T2D | 2000+53i | 4000+53i |
| ADI | 2000+53i | 4000+53i |
| VPENTA | 1028+47i | 2056+47i |

(b) Problem sizes ($i = 0 \dots 14$).

**Table 3. Evaluated kernels.**

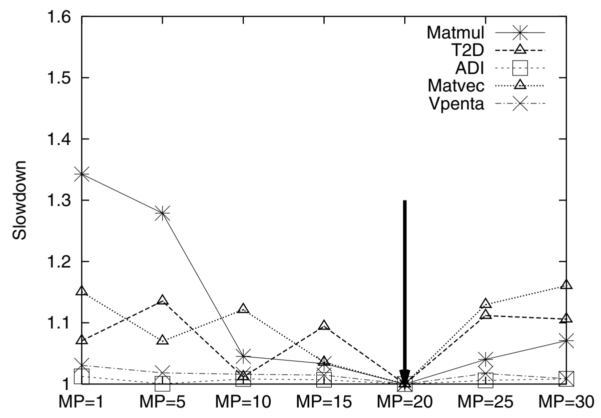


**Figure 5. Impact of branch miss-prediction overhead for the Pentium-4 processor. Results are normalized to our estimated penalty, $\mu_{MP} = 20$.**

Performance is reported for three modern machines. Table 2 shows their memory configurations. An overview of the five evaluated kernels can be seen in Table 3(a). For all of them, we have studied 15 different sizes. The second column of Table 3(b) shows the sizes considered for the Pentium-4 and the Alpha-21264, whereas the third column shows the sizes used for experimenting on Itanium. All kernels are written in FORTRAN, drawn from different benchmarks (NAS , LIVERMORE). We chose these kernels because they exhibit high number of capacity misses.

To evaluate our method, we compare our results with two other methods which represent the state-of-the-art:

- **lrw**: Lam *et al.* [20] choose the largest non-conflicting square tile.
- **tss**: Rivera and Tseng [27] extend Coleman and McKinley's [7] Euclidean GCD algorithm.

For the sake of comparison, all kernels are compiled with "g77 -O3".

## 5. Experimental Results

In this section we present performance results. We start studying the importance of considering the loop tiling overhead. We show that without an accurate estimate of the penalty of miss-predicted branches, the degradation in performance can be severe. Then, we analyze the efficiency of our approach. We present results of applying our method, and we compare them with previous works.

### 5.1. Loop Tiling Overhead

The results of our first set of experiments are shown in Figure 5. In order to show the importance of considering branch predictor behavior, we have analyzed different penalty values for the Pentium-4 processor. Since padding does not change the shape of the loop nest, we only consider the effects of tiling. We have run our approach for obtaining the best tile sizes considering different values of $\mu_{MP}$, and compared the execution times to that of the estimated penalty ($\mu_{MP} = 20$). We report results in terms of slowdowns. We can see that in general, execution time converges smoothly to our solution, which confirms our intuition. When the penalty is set to small values, the degradation in performance may be very important (up to 34% for MATMUL). This is because we generate tiles that are very small in order to minimize memory penalty, though incurring in a high overhead due to the increased number of miss-predicted jumps. On the other hand, if we set large penalty values, we prioritize branches overhead, thus, tiles are bigger but we incur in more misses that can degrade performance.

### 5.2. Performance Evaluation

Now, we show the effectiveness of our method compared to **lrw** and **tss** (see Section 4).

#### 5.2.1 Tiling the Matrix Multiplication Kernel

While **lrw** and **tss** can be applied to any loop nest, they were originally thought for programs involving matrix operations, and especially, for tiling matrix multiplication.

In this subsection we compare our tile selection approach with them for the matrix multiplication kernel. We present results for 5 possible loop orders, IKJ, JIK, JKI, KIJ and KJI (the remaining IJK order is used in the next subsection). We

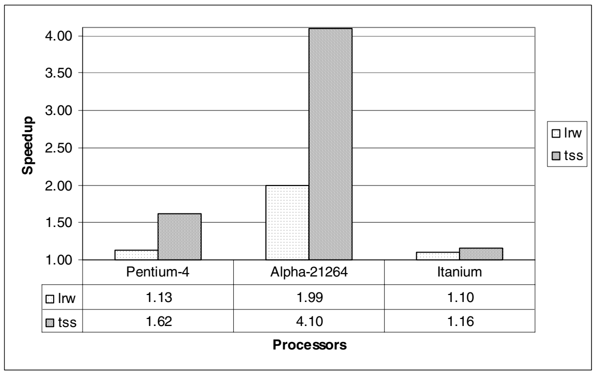


**Figure 6. Speedups obtained by our approach compared with lrw and tss algorithms for 5 different loop orders of the MATMUL kernel.**

use 15 different matrix sizes:

$$\mathrm{N} = 1000 + 53i,\ 0 \leq i \leq 14$$

Figure 6 shows the average speedups of our method (only tiling is applied) compared to **lrw** and **tss**. For obtaining these results, we have run all different approaches to select the best tile sizes for each platform. Then, we have executed the tiled version measuring the actual execution time. The results show that our approach outperforms these two techniques significantly, up to 310% for **tss** on the Alpha processor. We also show that our approach is better than both techniques for all platforms, with improvements ranging between 10% and 310%.

### 5.2.2 Tiling and Padding

We now present results for a set of common kernels that may benefit from tiling and padding. Figure 7 shows, for each machine, to what extent our method is better in terms of execution time of the optimized codes.

We first consider the results where only loop tiling is applied. For each program, the first two bars report the speedup compared to **lrw** and **tss** respectively. In all cases our method yields better results than previous approaches. Our ability to select tile factors results in important run-time improvements; on average, our transformed code runs 8% and 49% faster on a Pentium-4 compared to **lrw** and **tss**. On the Alpha machine results are even more impressive, with average speedups of 63% and 195% respectively.

Now, we consider results where both tiling and padding techniques are applied. The second set of bars reports the speedup compared to **lrwPad** and **tssPad**, enhanced versions of **lrw** and **tss** where padding is allowed [27]. The

| Processor | MIN | MAX | AVG |
|---|---|---|---|
| Pentium-4 | 1.8 | 14.5 | **4.63** |
| Alpha-21264 | 0.1 | 11.9 | **3.61** |
| Itanium | 0.4 | 17.0 | **5.50** |

**Table 4. Compile-time overhead when selecting tile and pad factors on a Pentium-4 running at 1.6GHz. Time is measured in seconds.**

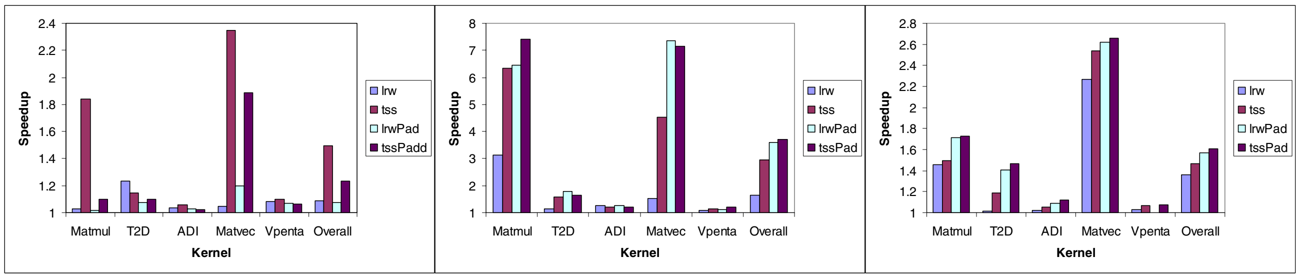
memory requirement was roughly the same for all three approaches. We can see that the speedup is smaller on the Pentium-4, where our transformed code runs 7.7% and 26% faster than **lrwPad** and **tssPad** respectively. However, the difference increases on the Alpha (260% and 271%).

Finally, in order to see to what extent tiling and padding help enhancing the program, we show in Figure 8 the speedups that the different approaches (with and without padding) obtain w.r.t. the original kernel. The application of padding on **lrw** and **tss** does not always translate to a better performance. Padding improves especially **tss** on the Pentium-4, but it yields worse results on the Alpha machine. On the other hand, our approach applies padding in concert with tiling using the same cost model; if padding is not useful, our cost model will predict a performance degradation, so pad factors will be set to 0. Overall, our approach obtains (98%, 204%, 49%) average speedups on the Pentium-4, Alpha and Itanium respectively. Combining the other methods with padding, **lrwPad** obtains (69%, 19%, 16%) and **tssPad** (20%, 80%, 11%). Otherwise, their speedups are (74%, 78%, 9%) and (30%, 20%, 3%) for **lrw** and **tss** respectively.

Note that the use of an accurate model allows us to obtain always a version of the code that it is not worse than the original one. For instance, when optimizing MATVEC for the Itanium platform our cost model determines that tiling is not useful, thus we do not apply it. However, the other approaches do not have an accurate model that guides the transformations, which results, some times, in an optimized code that runs slower than the original version.

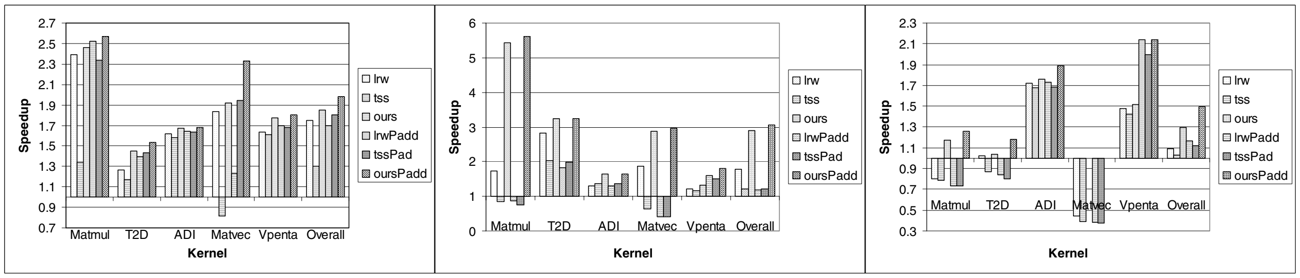
## 5.3. Compile-Time Overhead

Clearly, for our method to be considered a realistic optimization approach, it must be shown that the compile time required is small enough to be practical. In order to investigate this, we have collected the execution time needed to obtain tile and pad factors for all our experiments. We account for 15 problem sizes for each of the 5 kernels. We also include the time needed to optimize the different versions of the matrix multiplication kernel. Overall, we account for 450 experiments.



(a) Pentium-4 (b) Alpha-21264 (c) Itanium

**Figure 7. Speedup obtained by our approach compared with lrw and tss algorithms.**



(a) Pentium-4 (b) Alpha-21264 (c) Itanium

**Figure 8. Speedup of all approaches w.r.t. the original program.**

Table 4 shows the average times needed to generate the optimized version (including both tiling and padding) for each architecture. We see that in the worst case, it takes an average of 5.5 seconds to optimize a code, which we believe is reasonable for a static compiler.

## 5.4. Summary

Overall, we have shown the effectiveness of our method to select tile and pad factors. We first have presented results that highlight the importance of modeling the branch predictor behavior. Later, we have seen that our approach outperforms state-of-the-art techniques to select tile sizes for all analyzed kernels, for all platforms. We have shown how our cost model selects tile and pad factors in concert, which translates to consistent speedups.

From these results, we conclude that accurate cost models that consider not only cache behavior but other hardware components are necessary. A simple cost model may hinder compiler's ability to generate good code that improves overall performance. For instance, it is not clear when padding should be combined with tiling for the **lrw** and **tss** algorithms.

## 6. Related Work

Caches are an essential part of processors for reducing memory latency and increasing memory bandwidth. By reducing the number of accesses to the slow upper levels of the memory hierarchy, significant speedups can be achieved. Conflict misses may represent the majority of intra-nest misses and about half of all cache misses for typical programs and cache architectures [22].

Researchers working on locality optimizations have considered re-ordering techniques such as loop interchange [9, 21, 35, 36], loop fission/fusion [21] and loop tiling [4, 7, 20, 27, 29, 35, 37, 38]. Recently, GAs have been used to reduce code size [8] or select the best phase order [19].

The success of loop tiling depends on the tile size and shape selection. Lam *et al.* [20] present an algorithm that chooses the largest non-conflicting square tile, considering caches with low associativity. Coleman and McKinley [7] try to maximize the tile size while minimizing the cross-interferences. Their cost model is based on computing the footprints of the array references. Rivera and Tseng [27] further extend the Euclidean algorithm [7] by computing tile widths using a recurrence. They realize that there may be some pathological problem sizes where tile selection does not work very well. They propose padding the first dimension of all arrays with the same pad to eliminate such cases.

Array padding can help eliminate conflict misses. Rivera and Tseng [25, 26] propose several simple heuristics that are addressed to eliminate conflicts in some particular cases.

They mainly focus on conflicts that occur on every loop iteration, addressing only inter-padding for uniformly generated references (so they cannot 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 cannot 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 into account that interferences arise in different situations for different cache architectures. They presented an extension of this work targeting multi-level caches [28], where they study the effects of optimizing L1 cache on L2 cache behavior.

Ghosh *et al.* [10] use the CMEs to propose a tiling and padding technique. Padding works on direct-mapped caches, optimizing 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. Tiling is based on maximizing the tile size for every self-interference equation, which selects a tile that has no conflicts for the given equation. However, they do not give insights about how to combine the different tile sizes obtained. Furthermore, tiling is not applied to cross-interferences.

Our approach has several advantages over previous research. First, our cost model describes accurately the cache behavior of any memory hierarchy, and considers all array accesses within a loop nest. Moreover, we model tiling overhead due to miss-predicted branches. Second, our padding considers different pad factors for each array dimension, increasing the chances of finding a better optimized code. Finally, we perform tiling and padding at the same time, hence considering a global solution.

## 7. Conclusions

This paper presents a new approach to improve execution time of programs by improving data locality. It combines tiling and padding to remove both capacity and conflict misses. First, we present a very accurate model that describes cache locality across different levels. Moreover, our cost model takes into account the possible tiling overhead due to the added miss-predicted branches. We discuss how this model can be tuned to describe accurately the performance cost for different modern architectures.

Second, we introduce the use of genetic algorithms to traverse the solution space. We show how our approach can guide compiler optimizations efficiently; with a small compile-time overhead (average of 4.58 seconds per kernel), we obtain very important run-time improvements. Our results show that, compared to the best of the state-of-the-art approaches for each particular architecture, we have 7.7%, 63.2% and 35.7% average speedup for Pentium-4, Alpha-21264 and Itanium respectively.

Overall, this paper contributes with a new technique that makes a case for the use of accurate models to guide compilers in order to improve execution time. Moreover, it does not only model cache behavior but hardware components such as branch predictors, which shows the possibility of having complex and accurate models for actual architectures.

While this work represents an important attempt for improving global optimization, there are still some issues that can be investigated further. Future work will first investigate the application of padding and tiling for whole programs. For that purpose, we plan to plug our scheme to the new analytical model that studies whole programs [34]. In addition, we want to explore the use of other compiler techniques such as loop fusion, loop interchange and loop unrolling.

## Acknowledgements

This work has been supported by the ESPRIT project MHAOTEU (EP 24942). Xavier Vera was supported in part by VR grant no. 2001–2575 and CICYT project 511/98. Jaume Abella, Josep Llosa and Antonio González were supported by CICYT project 995/2001.

We wish to thank Erik Hagersten for helpful discussions on the importance of modeling “more than L1 caches” to improve performance on current machines.

## References

[1] J. Abella, A. González, J. Llosa, and X. Vera. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. Technical Report UPC-DAC-2001-20, Universitat Politècnica de Catalunya, June 2001.

[2] N. Bermudo, X. Vera, A. González, and J. Llosa. An efficient solver for cache miss equations. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’00)*, 2000.

[3] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Sales, and M. Shebanow. Instruction level parallelism is greater than two. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA’91)*, pages 276–286, 1991.

[4] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing (SC’92)*, pages 114–124, Nov. 1992.

[5] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of VI Int.*

*Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, pages 252–262, Oct. 1994.

[6] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layout for hierarchical memory systems. In *Proceedings of ACM International Conference on Supercomputing (ICS'99)*, pages 444–453, Rhodes, Greece, Jun. 1999.

[7] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 279–290, Jun. 1995.

[8] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9. ACM Press, 1999.

[9] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[10] 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.

[11] Gill, Murray, and Wright. *Practical optimization*. Academic Press, 1981.

[12] Glover and Laguna. *Tabu search*. Kluwer, 1997.

[13] D. Goldberg. *Genetic algorithms in search, optimizations and machine learning*. Addison-Wesley, 1989.

[14] Hansen, Jaumard, and Mathon. Constrained nonlinear 0-1 programming. *ORSA Journal on Computing*, 1995.

[15] J. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, 1975.

[16] R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*. Kluwer Academic Publishers, 1995.

[17] M. Kandemir, A. Choudhary, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, Feb. 1999.

[18] Kirkpatrick, Gelatt, and Vecchi. Optimization by simulated annealing. *Science 220*, 1983.

[19] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23. ACM Press, 2003.

[20] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance of blocked algorithms. In *Proceedings of IV International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, Apr. 1991.

[21] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul. 1996.

[22] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of VII Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996.

[23] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of V Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 62–73, Oct. 1992.

[24] E. Petrank and D. Rawitz. Hardness of cache conscious data placement. In *Proceedings of International Conference on Principles of Programming Languages (POPL'02)*, 2002.

[25] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, 1998.

[26] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of ACM Internacional Conference on Supercomputing (ICS'98)*, 1998.

[27] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99)*, 1999.

[28] G. Rivera and C.-W. Tseng. Locality optimizations for multi-level caches. In *Proceedings of Supercomputing (SC'99)*, 1999.

[29] V. Sarkar and N. Meggido. An analytical model for loop tiling and its solution. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, 2000.

[30] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for accessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing (SC'93)*, pages 410–419, 1993.

[31] Torn and Zilinskas. *Global optimization*. Springer-Verlag, 1989.

[32] X. Vera, J. Llosa, and A. González. Near-optimal padding for removing conflict misses. Technical Report UPC-DAC-2000-71, Universitat Politècnica de Catalunya, Nov. 2000.

[33] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior. In *Proceedings of European Conference on Parallel Computing (Europar'00)*, 2000.

[34] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA 8)*, Cambridge, Feb. 2002.

[35] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Jun. 1991.

[36] M. Wolfe. Advanced loop interchanging. In *Proceedings of International Conference on Parallel Processing (ICPP'96)*, 1996.

[37] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[38] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. *International Journal of Parallel Programming*, 26(6):671–696, 1998.

IEEE COMPUTER SOCIETY