Optimal Tile Size Selection Guided by Analytical Models*

Basilio B. Fraguela[a], Martín G. Carmueja[a], Diego Andrade[a]

[a] Dept. de Electrónica e Sistemas, Universidade da Coruña, Facultade de Informática, Campus de Elviña, 15071 A Coruña, Spain. {basilio,carmueja,dcanosa}@udc.es

As the memory bottleneck problem continues to grow, so does the relevance of the techniques that help improve program locality. A well-known technique in this category is tiling, which decomposes data sets to be used several times in a computation into a series of tiles that are reused before proceeding to process the next tile. This way, capacity misses are avoided. Finding the optimal tile size is a complex task. In this paper we present and compare a series of strategies to search the optimal tile size guided by an analytical model of the whole memory hierarchy and the CPU behavior. Our experiments show that our strategies find better tile sizes than traditional heuristic approaches proposed in the literature while requiring a small compile-time overhead. Iterative compilation can yield better results, but at the expense of very large overheads.

1. Introduction

The success of memory hierarchies in bridging the gap between the processor and the main memory speeds depends on the locality of the data accesses. Such locality can be improved by a number of software optimizations [2]. One of the most effective techniques is tiling [8,10,4,13,7], which combines strip-mining and loop permutation to create small tiles of loop iterations that concentrate the accesses to a given data set, thus maximizing access locality. The tile sizes are chosen so that the associated data sets fit in the cache; which results in a reduction of the capacity misses. The effect of tiling on multiple processors is even more significant, since it not only reduces average data access latency, but also the required memory bandwidth. Still, the fact that caches have a limited associativity and that normally during a portion of the execution of a program several data structures need to be accessed implies that despite the application of this technique, many capacity and conflict misses can take place. Besides, the number of misses may vary widely depending on the chosen tile size, since cache behavior is unstable and very sensitive to small changes on a large number of parameters [10,14].

Most of the approaches proposed so far to choose an optimal tile size are based on relatively simple heuristics [10,4,13,7]. These algorithms have many restrictions. The most important ones are that they adopt a very simplified view of the cache behavior, they do not consider the additional CPU time required to manage the tiles, they are restricted to a single cache level, and they only consider accesses to a single data structure. As a result, very often the tiles proposed by these techniques are far from being optimal. Iterative compilation techniques [9] to explore the solution space yield better results. Unfortunately these techniques are too costly to be applied repetitively and/or in big applications except in very particular cases.

An optimal tile size search technique introduced in [5] lies in exploring the solution space guided by a precise analytical model that considers both the CPU and the whole memory hierarchy cost of a tile size. This paper compared the model predictions for a fixed set of tile sizes that were divisors (or values nearby) of the sizes of the loops in the considered nests and chose the one with the lower cost.

*This work has been supported in part by the Ministry of Science and Technology of Spain under contract TIN2004-07797-C02-02, and by the Xunta de Galicia under contract PGIDIT03-TIC10502PR.

Later, [15] applied similar ideas with other models using a genetic search algorithm. Both works lacked a study of which of the possible search strategies would yield the best results. In this paper we compare both search techniques as well as a novel hybrid one we propose here which turns out to yield the best results. We also compare the results of search guided by models with those of the traditional heuristic approaches and iterative compilation.

The rest of the paper is organized as follows. An introduction to our model of the memory hierarchy is provided in Sect. 2. Our approach to estimate the cost associated to a tile size is explained in Sect. 3. Then, Sect. 4 presents the optimal tile size search strategies that we consider. The experiments in Sect. 5 shows the relative advantages of the different tile size selection techniques. Section 6 contains a review of related work. Finally, Sect. 7 is devoted to our conclusions.

2. The Probabilistic Miss Equations (PME) Model

The PME model [5] provides fast and accurate predictions of the memory behavior of codes with regular access patterns in direct-mapped or set-associative caches with a LRU replacement policy. The model is based on the idea that cache misses take place the first time a memory line is accessed (compulsory miss) and each time a new access does not find the line in the cache because it has been replaced since the previous time it was accessed (interference miss). This way, the model generates a formula for each reference and loop that encloses it that classifies the accesses of the reference within the loop according to their reuse distance, this is, the portion of code executed since the latest access to each line accessed by the reference. The reuse distances are measured in terms of loop iterations and they have an associated miss probability that corresponds to the impact on the cache of the footprint of the data accessed during the execution of the code within the reuse distance. In particular, if K is the degree of associativity, the miss probability is given by the ratio of cache sets that have received K or more lines during the execution of the code within the reuse distance. The reason is that if each cache set holds K lines, a line whose behavior is being observed will have been replaced if and only if K or more different lines mapped to its cache set have been accessed since the previous access to the line. The formula estimates the number of misses generated by the reference in the loop by adding the number of accesses with each given reuse distance weighted by their associated miss probability.

The accesses that cannot exploit reuse in a loop are compulsory misses from the point of view of that loop. Still, they may enjoy reuse in outer loops, so they must be carried outwards to estimate their potential miss probability. This is why the PME model begins the construction of its probabilistic formulas in the innermost loop that contains each reference and proceeds outwards. The formulas are built recursively, with the formula for each loop level being built in terms of the formula obtained for the immediately inner level. In each nesting level the set of accesses whose reuse distances correspond to iterations of that loop are detected, and the associated miss probabilities are estimated. Those accesses for which no reuse distance has been found when the outermost loop is reached correspond to the absolute compulsory misses, whose miss probability is one.

The usage of miss probabilities, generated by estimators provided by the model of the impact on the cache of the accesses to data regions during the reuse distances, gives place to the model's probabilistic nature, which distinguishes it from all the other cache models in the bibliography.

3. Computer Modeling

The PME model provides accurate estimations of the behavior of a cache, but current computers have a memory hierarchy with several levels of caches, and the CPU plays of course an essential role

in the system performance too. The first problem is solved by extending the PME model to consider a hierarchy of cache levels. Each level i is characterized by a total cache size C_{s_i}, a line size L_{s_i}, and an associativity K_i. Given these parameters and an arbitrary code, the model can predict the number of misses generated in each level of memory hierarchy. The property of inclusion that multilevel memory hierarchies fulfill allows the independent modeling of the different levels, as each access that were a hit in a given cache level, should also be a hit in the lower levels as a consequence of this property. Still, the caches in the different levels do not experience the same pattern of accesses, since accesses are filtered by the successive levels as they proceed down the hierarchy. This fact can generate some deviations with respect to this ideal behavior, but they are minimal and so they affect little the accuracy of the estimations obtained following this approach.

Since we aim to predict the performance of the whole memory hierarchy, misses in the different levels receive different weights W_i that are given by the miss penalty in cache level i measured in processor cycles. This way, the cost of the execution of loop nest L using the set of tile sizes T in a system with N cache levels can be estimated as

$$\text{MemCost}(L, T, H) = \sum_{i=1}^{N} W_i M(C_{s_i}, L_{s_i}, K_i, L, T) \tag{1}$$

where function M yields the number of misses estimated by the model for loop nest L for a given cache level and set of tile sizes; and H stands for the memory hierarchy represented as a set of tuples $H = \{(C_{s_1}, L_{s_1}, K_1, W_1), \ldots, (C_{s_N}, L_{s_N}, K_N, W_N)\}$.

We use the Delphi [3] CPU model in order to estimate the number of cycles $\text{CPUCost}(L, T, C)$ that code L spends in the CPU C depending on the tile sizes T. This model simply counts the number of operations of each type found within the code and it adds them weighting them according to their respective latencies. Since current processors are superscalars, Delphi uses some simple heuristics to take into account the overlapped execution of instructions so as not to overestimate the CPU time.

This way our final estimation of the cost of executing loop nest L with the set of tile sizes T in a computer with a CPU C and a memory hierarchy H is given by

$$\text{Cost}(L, T, C, H) = \text{MemCost}(L, T, H) + \text{CPUCost}(L, T, C) \tag{2}$$

The Delphi and the PME models, as well as our optimal tile size search module are implemented in the Polaris platform [1], which we use to analyze the codes and generate the optimized versions with the tile sizes chosen by our module.

4. Search Strategies

Our approach to find the tile sizes that minimize the execution time of a loop nest L lies in exploring the solution space (the combinations of possible tile sizes) guided by our analytical model. Such exploration can be performed using several strategies. In our experiments we have explored the viability of two completely different strategies: the search on divisors, an ad-hoc algorithm specifically designed for this problem that searches only in a set of predefined values, and a genetic algorithm, which is a general search approach applicable to any solution space. Then, we have developed a hybrid strategy that combines them.

4.1. Search on Divisors

In each loop nest in which tiling has been applied, this algorithm searches a well-defined subset of all the combinations of possible tile sizes. Concretely, for each tiled loop it chooses the initial

Table 1
Cache and TLB parameters in the architectures used (sizes in Bytes)

Architecture	L1 Parameters $(C_{s_1}, L_{s_1}, K_1, W_1)$	L2 Parameters $(C_{s_2}, L_{s_2}, K_2, W_2)$	L3 Parameters $(C_{s_3}, L_{s_3}, K_3, W_3)$	TLB Parameters $(C_{s_4}, L_{s_4}, K_4, W_4)$
Pentium 4	(8K,64,4,24)	(512K,128,8,150)	-	(256K,4K,64,30)
Itanium 2	Irrelevant	(256K,128,8,24)	(6MB,128,24,120)	(8MB,64K,128,25)

tile size specified by the program, if any, and the series of values $T_i = \lceil N/i \rceil$, where N is the total loop size. The intention of choosing divisors of the loop sizes is to maximize the work made in each iteration of the loops that traverse the tiles, and to simplify the control conditions of the loops. In our experiments we used $0 < i \leq 128$, and whenever two values of T_i differed in less than three units, one of them was discarded. This algorithm, applied in [5], searches the best combination of values chosen from these sets for the different tiles in a loop nest. This approach explores quickly the solution space paying more attention to small tile sizes.

4.2. Genetic Algorithm

Genetic algorithms are stochastic methods applicable to problems for which no specific resolution method is available. These algorithms simulate the genetic and natural selection processes. Namely, an initial set or *population* of candidate solutions is encoded as bit strings arbitrarily codified. These strings are modified and recombined to produce new solutions. The solutions are evaluated by means of a *fitness function* that chooses the best ones and promotes them as the base for a new *generation* of solutions. Crossover and mutation are applied on minimal units of information called *genes* in order to recombine and alter the strings. The process of generation and selection of solutions is applied repetitively for several generations till certain convergence criteria are met. The adaptability of these algorithms, a good coverage of the solution space and their inherent parallelism make them suitable for the search of the optimal tile size.

4.3. Hybrid Algorithm

Both the search on divisors and the genetic algorithm have interesting properties to solve our problem. It is possible to combine their advantages following a hybrid strategy. The hybrid algorithm we propose has two phases. In the first step, an approach to the solution is found using the search on divisors. In the second phase, the result is refined using a genetic algorithm. The n best solutions found in the first phase constitute the initial population for the second phase. This population is completed using versions with noise of elements randomly chosen among the n first ones.

5. Evaluation

For each search strategy we implemented two versions in our tool: one based on the exploration of the solution space guided by the PME analytical model, and another one based on iterative compilation [9]. We also implemented two of the most popular traditional techniques based on heuristics in order to compare them with our strategies. The first one, which we call *lrw* [10], chooses the biggest square block that does not collide with itself in the cache. The second technique, which we name *euc*, was developed in [13] by extending the work in [4], based on the Euclidean GCD algorithm.

We run our experiments in two popular platforms: a Pentium 4 at 2 GHz with g77 3.3, which is representative of the most widely extended architecture nowadays, and an Itanium 2 at 1.5 GHz with a 6MB third level cache and the HP F90 2.7.3 compiler. The `O3` optimization level, and flags to preclude the compiler for applying additional loop transformations that would distort the experiments were used to compile the codes in both systems. Table 1 shows the configuration of the TLB

Table 2
Description of the kernels and sizes used for the experiments, where $i = 0, 1, \ldots, 19$.

Kernel	Description	Pentium 4 sizes	Itanium 2 sizes
MXM	Matrix product (IJK)	$300 + 50 \times i$	$1000 + 50 \times i$
MV	Matrix-vector product	$300 + 50 \times i$	$2000 + 50 \times i$
TRANSP	Bidimensional matrix transposition	$1700 + 50 \times i$	$4000 + 50 \times i$
VPENTA	Inversion of three pentadiagonals	$800 + 50 \times i$	$2000 + 50 \times i$

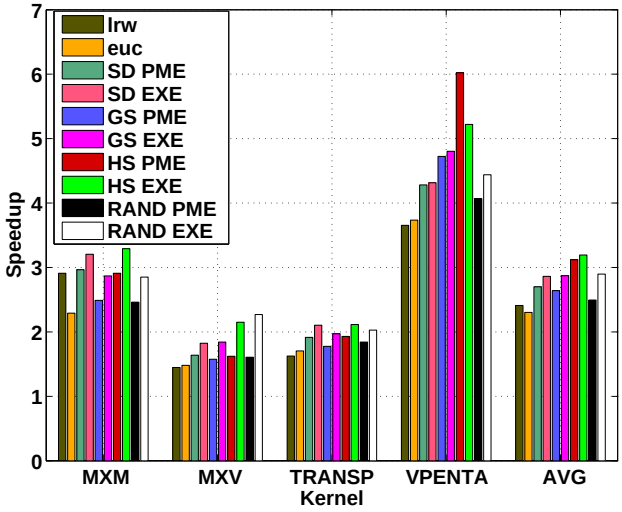

Figure 1. Tiling speedup in the Pentium 4 as a function of the strategy followed to choose the tile sizes

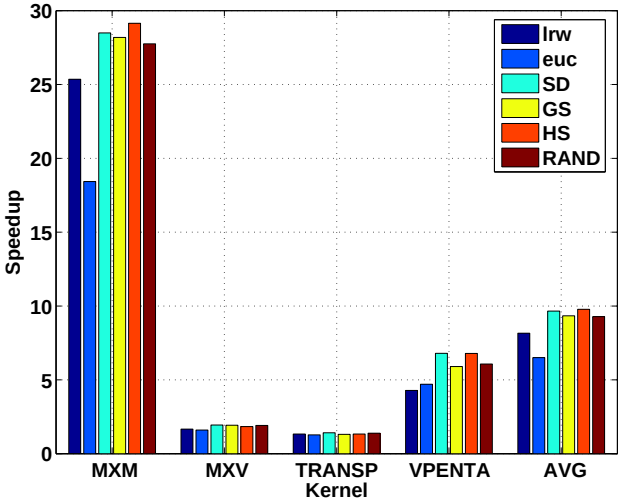

Figure 2. Tiling speedup in the Itanium 2 as a function of the strategy followed to choose the tile sizes

and the cache levels of both systems. The TLB acts as a level of the memory hierarchy from the point of view of the execution time, and this is how it is regarded by the analytical model. In fact it can be characterized with the same parameters as a cache, the page size being the line size, and the product of the page size by the number of entries of the TLB being the total size of the level. Let us notice that the Pentium 4 has only two level of caches, while the first level cache of the Itanium 2 is irrelevant for our experiments, since it does not store floating point data; which is the kind of data we use in our experiments. As for the CPU parameters, the most relevant one from the point of view of the tile size selection is the penalty associated to misspredicted branches, which is 20 cycles in the Pentium 4 architecture and 8 for the Itanium 2.

We used for our experiments four representative kernels taken from [15]. Table 2 describes them briefly and shows the sizes considered for our experiments in both platforms. The sizes used in the Itanium 2 are much larger than those used in the Pentium 4, since while the first level of cache to consider in the Itanium 2 has 256 KBytes, the first level cache of the Pentium 4 has only 8 KBytes. Something similar happens when we compare the size of their corresponding next level caches and the TLBs.

Figures 1 and 2 show the speedups achieved by each tiled algorithm with respect to its non-tiled version depending on the strategy followed to choose the tile size on the Pentium 4 and the Itanium 2 architectures, respectively. The last set of bars, labeled AVG, represents the arithmetic mean of

the speedup achieved over the four kernels for each strategy. Columns *lrw* and *euc* correspond to the traditional techniques described in [10] and [13] respectively, as explained before. Bars SD, GS, HS and RAND correspond to the Search on Divisors (Sect. 4.1), the pure Genetic Algorithm Search (Sect. 4.2), the Hybrid Search (Sect. 4.3) and a Random Search, respectively. The parameters used for the genetic algorithms in GS and HS are very similar to those used in [15] and [9]. As for the hybrid search, its genetic search started with a population formed by the 30 best tiles found by the search on divisors. The random search tried 184 randomly generated tile sizes, which is a value similar to the average number of tiles considered by the genetic algorithm search. In the Pentium 4 architecture we tested each search strategy by guiding it either by the PME analytical model static predictions or by means of real executions (iterative compilation). Both columns are labeled PME and EXE in Fig. 1 respectively. We could only run experiments using the static predictions in our Itanium 2 because of its limited availability and the large amount of time that iterative compilation requires.

We can see that while the tiles chosen by the genetic search (GS) and the random search (RAND) sometimes performed worse than those of the traditional approaches, the search on divisors (SD) and the hybrid search (HS) always generated results at least as good as those of the heuristics. As expected, iterative compilation generates almost always better results than the static model, but in general the difference is small: on average the tiles chosen by the iterative compilation are 8.3% faster than those chosen by the static search, and just 5.7% if we exclude the blind random search. Interestingly, HS PME chooses a much better tile than HS EXE in VPENTA. This may be due to the random factor in any genetic algorithm. The average speedup of the static SD, GS, HS and RAND PME searches over the best traditional approach (*lrw*) is 12%, 9.6%, 29.5% and 3.53% in the Pentium 4, respectively; and 18.4%, 14.4%, 19.8% and 13.8% in the Itanium 2, respectively. This way, the novel hybrid search we propose in this paper seems to be the best overall strategy, followed by the search on divisors.

Tile selection guided by the PME analytical model is completely feasible for current compilers, given that the average time to choose the tile sizes was about 0.82 seconds per code in the Pentium 4 and 2.15 seconds in the Itanium 2, with maximum times below 7 seconds in both architectures. In contrast, iterative compilation times are usually in the order of the thousands of seconds in the Pentium 4, with an average search time of 2709 seconds, and a maximum time of 20440 seconds.

6. Related Work

Traditional approaches to choose the optimal tile size [10,4,13,7] have many limitations: they disregard the CPU time, they focus on a single level of the memory hierarchy and they rely on simple heuristics that dismiss important cache parameters like the associativity, that only consider the reuse in a single array, and that pay little or no attention to the interactions in the cache among several data structures, which could include several tiles generated by the dimension partitioning implied by tiling. For example, the fact that traditional approaches only consider a single tile while our general analytical model considers arbitrary inter-tile interactions is one of the reasons why the speedups achieved by our approach with respect to the traditional ones is much more noticeable in VPENTA than in the other kernels: this code has two tiled loop nests in which tiling defines at least three different tiles that interact in the cache.

The importance of taking into account the interactions in several levels of the memory hierarchy and using global metrics rather than focusing on local cost functions when choosing tile sizes has been proved in [11]. Still, this work does not propose any general framework for this purpose: it is based on a specific model with many simplifications for a particular code. A framework for the op-

timal multi-level orthogonal tiling problem for fully permutable, perfectly nested, rectangular loops that are compute bound, i.e., in which the amount of computation is at least one order greater than the amount of memory operations, is presented in [12]. This high-level model does not consider may factors like cache associativity, caches hits/misses, etc., but the authors suggest applying simple heuristics to include them in the model. Interestingly, the paper shows that the model tracks approximately the execution time of simple loops, but there are no measurements on the quality of the tiles the framework is supposed to choose. Multi-level semi-oblique tiling, which is more general than the orthogonal tiling considered by our paper and most works, has been studied by [16] and [6] in the context of multiprocessors. The applicability of [16] is restricted to a class of iterative stencil calculations, and it does not address the problem of tile size optimization. The approach in [6] is much more general, but it only handles perfectly nested loops and it does not provide any model for the memory memory behavior; rather it focuses on improving parallelism via minimization of the longest path of dependent tiles in the iteration space.

Iterative compilation [9] is based on the real execution on the machine, so it is the most reliable strategy to choose optimal tile sizes. Still, it is only applicable when long times can be devoted to the optimization process, and in our experiments the tiles it chooses are only fractionally better on average than those chosen by an exploration of the solution space guided by a good analytical model. In fact, the analytical model can even choose better tiles than the iterative compilation, since random factors may affect the search, as we have seen in our validation. The hybrid search strategy proposed in this paper has proved to be of interest for both static and iterative tile selection approaches.

Tile and pad factors selection guided by analytical models using a genetic algorithm search is explored in [15]. The relative speedups over *lrw* they achieve on a Pentium 4 with the same characteristics as the one used in our experiments (8%) are similar to those measured in our validation when we use the same kind of search (9.6%), while the times they require to drive the optimization process are several times longer than ours. A motivation for our work was to compare the genetic algorithm search approach applied in [15] with our search on divisors [5]. The latter yields better results in our experiments, but the optimal search strategy turns out to be the mixture of both strategies in the hybrid approach we present in Sect. 4.3.

7. Conclusions

In this paper we propose a new search strategy for the optimal tile size that combines the search on divisors of the loop sizes with a pure genetic algorithm approach. We also compare the traditional heuristic-based approaches to choose tile sizes with different algorithms that search the solution space guided by either estimations of an analytical model that predicts the performance of the computer, or measurements of actual executions (iterative compilation). Our experiments show that while iterative compilation requires three to five orders of magnitude more time than analytical model based search, on average it only delivers relatively small performance improvements over search guided by analytical models and in fact it can (seldom) deliver worse results. The search based on the PME analytical model estimations is suitable to be used in production compilers, since it typically requires one or two seconds; and never more than 7 seconds in our experiments. As for the quality of the search, the algorithms based on the exploration of the solution space almost always generate better results than the heuristical approaches, with our novel hybrid search being the best strategy, followed by the search on divisors. Both approaches are always better than the traditional heuristics.

Future work includes extending our tool to apply more optimizations guided by our analytical model and generalizing the PME model to cope with codes with irregular access patterns.

Acknowledgments

We want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers to get the data related to the Itanium 2 architecture.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[2] S. Carr, K. S. McKinley, and C-W. Tseng. Compiler Optimizations for Improving Data Locality. In *Proc. of the Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), Computer Architecture News*, volume 22, pages 252–262, Boston, MA, Oct. 1994. ACM SIGARCH/SIGOPS/SIGPLAN.

[3] G.C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Aug 2000.

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

[5] B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic Miss Equations: Evaluating Memory Hierarchy Performance. *IEEE Transactions on Computers*, 52(3):321–336, March 2003.

[6] K. Hogstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):307–321, 2003.

[7] C-H. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *Journal of Supercomputing*, 27(3):279–294, 2004.

[8] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California, 1998.

[9] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–248, 2000.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. of the Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 63–74, Santa Clara, California, Apr 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.

[11] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Int. J. Parallel Programming*, 26(6):641–670, 1998.

[12] L. Renganarayana and S. Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Proc. 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, page 18, Washington, DC, USA, Nov. 2004. IEEE Computer Society.

[13] G. Rivera and C.-W. Tseng. A Comparison of Compiler Tiling Algorithms. In *Proc. of 8th Int'l. Conf. on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 168–182, 1999.

[14] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomena. In *Proc. Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 261–271. ACM Press, May 1994.

[15] X. Vera, J. Abella, A. Gonzalez, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'03)*, pages 68–78, New Orleans, Louisiana, October 2003.

[16] David Wonnacott. Time skewing for parallel computers. In *Proc. of the 12th Intll. Workshop on Languages and Compilers for Parallel Computing (LCPC '99), LNCS vol. 1863*, pages 477–480, London, UK, 2000. Springer-Verlag.