Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation

T. Kisuki	P.M.W. Knijnenburg	M.F.P. O'Boyle
LIACS,	LIACS,	ICSA,
Leiden University,	Leiden University,	Edinburgh University,
the Netherlands	the Netherlands	United Kingdom
kisuki@liacs.nl	peterk@liacs.nl	mob@dcs.ed.ac.uk

Abstract

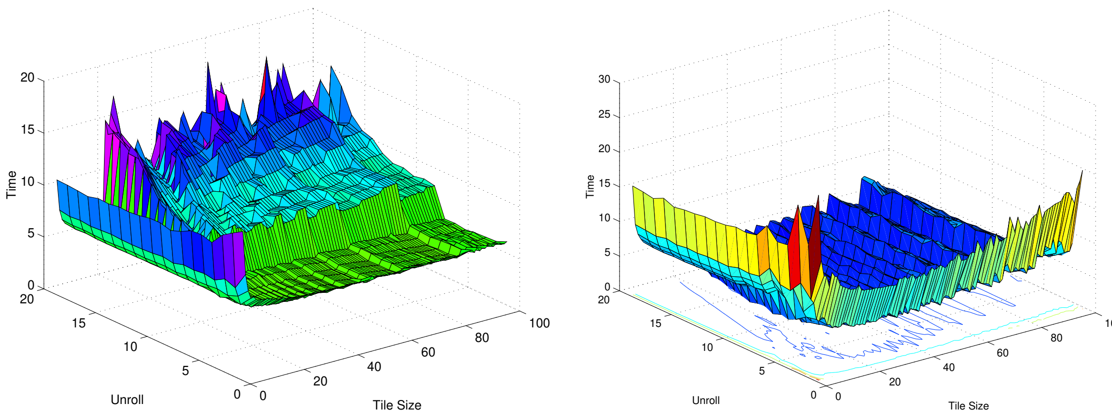
Loop tiling and unrolling are two important program transformations to exploit locality and expose instruction level parallelism, respectively. In this paper, we address the problem of how to select tile sizes and unroll factors simultaneously. We approach this problem in an architecturally adaptive manner by means of iterative compilation, where we generate many versions of a program and decide upon the best by actually executing them and measuring their execution time. We evaluate several iterative strategies. We compare the levels of optimization obtained by iterative compilation to several well-known static techniques and show that we outperform each of them on a range of benchmarks across a variety of architectures. Finally, we show how to quantitatively trade-off the number of profiles needed and the level of optimization that can be reached.

1. Introduction

Efficient use of the memory hierarchy is essential for good performance due to the ever increasing gap between processor and memory speed. Program transformations such as loop tiling or blocking have been shown to be an effective approach to improving locality and cache exploitation [7, 9, 14, 17]. In this approach one tries to divide the loop into smaller tiles in such a way that the working set of each tile fits in the cache thereby exploiting the available locality. It is also important to fully utilize the internal parallelism within modern processors which are capable of issuing several instructions per cycle [8]. Loop unrolling is an important transformation for this purpose [3] as it increases the size of the loop body, exposing more instructions for Instruction Level Parallelism (ILP). As we require effective utilization of the memory hierarchy and internal parallelism, we need to combine both of these transformations. In this paper, we address the problem of determining simultaneously optimal tile sizes and unroll factors for any given loop nest. Combining the best tiling transformation for locality with the best unrolling factor for ILP, however, does not necessarily give the best overall transformation as transformation application is not orthogonal in effect. Loop unrolling can adversely affect locality and tiling may restrict the available instruction level parallelism.

The close interaction between tiling and unrolling can be seen in Figure 1, which shows that a small deviation from 'good' tile sizes and unroll factors can cause a huge increase in execution time and even a slow down with respect to the original program. A static technique essentially tries to give an analytical expression for the minimum execution time. In [12] we have studied the characteristics of optimization spaces in detail for a variety of benchmarks and platforms and showed that different platforms give rise to widely varying optimization spaces (see Figure 1) and that a compiler, using a simplified static cost model, would have great difficulty to predict this minimum. Instead, we propose to actually search this space in a manner that is adaptable for different architectures. We call this approach *iterative compilation*, where we generate many versions of a program and try to determine their execution time. This can be done by actually executing the program on the target hardware, by employing one or more static models, or by a combination of both techniques. The version that performs best is selected and thus we determine the best combined tile size and unroll factor. In the present paper, we use real execution time to search the space for a minimal point and by using a generic iterative compilation strategy, we can find excellent optimizations across a range of architectures, outperforming static techniques significantly. Thus, we obtain highly architecture specific optimizations in an architecture independent manner.

This approach is highly attractive in situations which require high performance, such as embedded systems where



Pentium II UltraSparc

Figure 1. Execution Time MxM for Unrolling and Tiling

the compilation time can be amortized across the number of products shipped or in the case of vendor supplied library codes for which the same argument holds. It is also useful in contexts where the underlying architecture changes (e.g., additional memory, a new release of the low-level compiler or a completely new processor) as the iterative search strategy has no hardwired system dependent knowledge.

In section 2 the implementation of our iterative compilation system is briefly discussed. In order to assess the efficiency of our approach, we use a collection of small benchmark kernels and three target platforms in section 3. In section 4 we show that we can find good tile sizes and unroll factors by visiting only a tiny fraction of the entire optimization space. We assess the quality of the optimization found by comparing it to two well-known static tile size selection algorithms in section 5 and show that we outperform each of them in almost all cases. After analyzing the cost of compilation time in section 6, we limit the number of iterations to a realistic small and fixed amount and construct quantitative trade-off graphs, in section 7, that show that good performance can be reached with little cost. Finally, we discuss related work in section 8, a broader perspective in section 9 and draw some concluding remarks in section 10.

2. Implementation of Iterative Compilation System

Figure 2 shows an overview of the compiler system. For more details, consult [13]. The compilation system is centered around a global driver that reads a list of transformations that it needs to examine together with the range of their parameters. The driver keeps track of the different transformations evaluated so far and decides which transformations have to be applied next using a search algorithm to steer through the optimization space. We have implemented several search algorithms, including a genetic algorithm, simulated annealing, pyramid search, window search and random search (see section 3). The global driver invokes the source to source compiler MT1 [1] and instructs it which transformation to apply. MT1 has two mechanisms to control the application of transformations: a Transformation Definition Language (TDL) and a Strategy Specification Language (SSL) [1]. For each transformation included in the list of transformations, a transformation needs to be specified in the TDL-file. The global driver constructs an SSL file that specifies the order in which to apply certain transformations and outputs it to MT1. After a predetermined number of iterations, the global driver stops searching and outputs the transformed program with the shortest execution time.

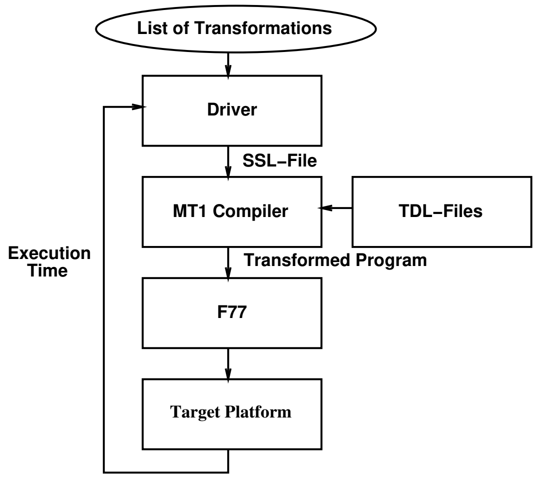


Figure 2. The Compilation Process

3. Experiment

In this section we discuss the parameters of our experiment, including the search algorithms employed by the global driver, the benchmarks and the target platforms.

3.1. Search Algorithms

We have implemented the following search algorithms.

Genetic algorithm First, an initial population of 20 programs is randomly selected. Second, in the crossover phase, for a number of individuals a crossover point is determined in the bit representation of the tile sizes and unroll factors. Different parts of the upper half and lower half of these 'chromosomes' are concatenated. Third, in the mutation phase, bits are flipped in the chromosomes. Finally, the entire new population is evaluated based on execution time and individuals are deleted until a new population of 20 is reached.

Simulated annealing Initially, a random point is selected and neighboring points are inspected. We move to the point with lowest execution time, or with a certain probability depending on the current temperature to a point with higher execution time. The temperature is subsequently decreased. We keep track of the best point visited so far.

Pyramid or Grid search We define a top level grid over the search space and evaluate each point on this grid. We order the points in a priority queue. Around the best points we refine the grid.

Window search We define windows over the search space. Initially, the window is the entire space. We take a number of samples and order them in a priority queue. Around the best points we define a smaller window.

Random search We randomly generate parameters.

We conducted a number of experiments to establish good values for the different parameters in the algorithms [10].

3.2. Benchmarks and Platforms

In order to assess the efficiency of iterative compilation for selecting tile sizes and unroll factors, we use many small kernel benchmarks from multimedia applications that exhibit a wide variety of memory access behavior. In this way, we are able to give a statistically relevant analysis of the results. Therefore, we chose the following benchmarks.

- Matrix-Matrix Multiplication (MxM). We use all 6 possible loop orders to generate 6 benchmarks with highly different memory access behavior. We use data input sizes of $N = 256$, $N = 300$ and $N = 301$.
- Matrix-Vector Multiplication (MxV). We use the two possible loop orders. We use data input sizes $N = 2048$, $N = 2300$ and $N = 2301$.
- Forward Discrete Cosine Transform. This benchmark is one of the most important routines from the low-level bit stream video encoder H263. It contains three initialization loops and two main loops: the first loop repeatedly calculates inner products and the other loop is a matrix-matrix multiplication. These loops are hand optimized in the reference implementation and we undid some of this optimization in order to remove a dependence that would prohibit some transformation. We use both the first main loop in isolation and the entire routine in our experiments. We use data input sizes of $N = 256$, $N = 300$ and $N = 301$.

We have conducted our experiments on three different platforms: Pentium II, Hewlett-Packard Precision Architecture (HP-PA 712/60) and UltraSparc. In order to compile a transformed version of the source program we used the native Fortran77 compiler with full optimization on.

We restricted attention to tile sizes from 1 to 100 and unroll factors from 1 to 20. Not every benchmark and data input size has been executed on every platform. Instead, we have considered a total of 82 different experiments.

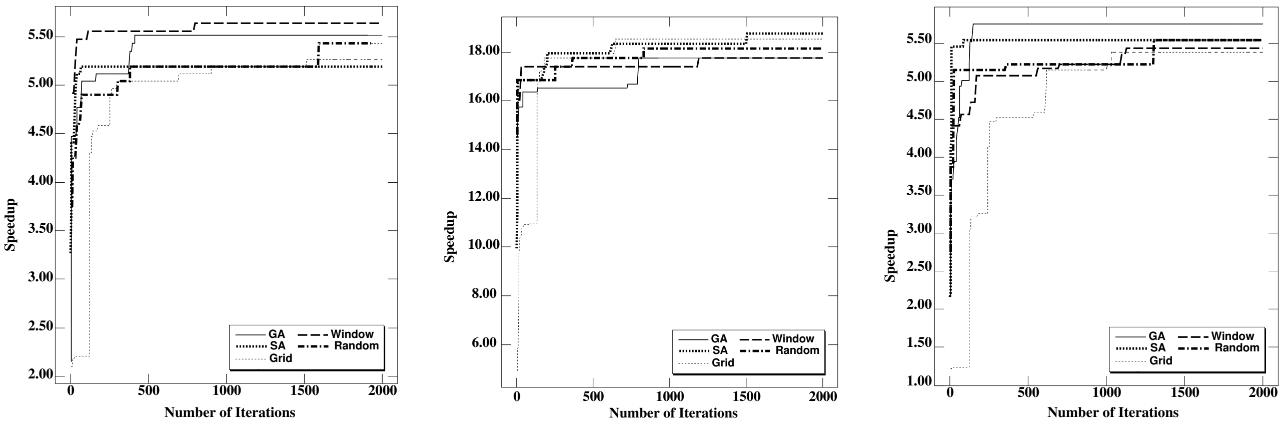
4. Results

In this section we show how much speedup we obtain as a function of the number of iterations, where we show the best speedup found so far. Due to space limitations we only show a few examples. See [10] for a more complete discussion.

4.1. Rectangular Tile Sizes

In this section we discuss rectangular tile sizes together with unroll factors. In this case, the search space consists of $20 \times 100 \times 100 = 200,000$ points. We let the search algorithm run for 2000 iterations. The results are given in Figure 3. The x-axis denotes the number of iterations, that is, the number of times a transformed version of the program is generated, compiled and executed. The y-axis denotes the speedup of the fastest version found so far.

The first observation is that iterative compilation indeed yields high levels of optimization. We have shown [10] that these high levels of optimization are found across all our benchmarks and platforms for all data input sizes. In section 5 we give a more quantified assessment of this by comparing iterative compilation to two well-known static techniques.



Pentium II HP-PA UltraSparc

Figure 3. Example: Speedup MxM – ikj version

The second observation we can make is that these search algorithms do not differ much in their efficiency. The different speedups found are within 5% on average of each other. We need on average between 750 and 1000 iterations to obtain the maximum speedup. However, iterative compilation reaches high levels of optimization much earlier and the last few hundred iterations are used for a small increase in the final outcome. SA and Random reach their maximum fastest. Pyramid search is slowest because initially we define a grid over the entire search space that consists of 500 points. We have shown [10] that we need only a fraction of the number of iterations for maximal speedup to reach 90% of this maximum: between 0.03% and 0.13% of the entire search space. However, the absolute number of iterations required is still high. Nevertheless, this observation enables us to propose a trade-off between the number of evaluations and the level of optimization in section 7.

We conclude that iterative compilation is capable of finding good unroll factors and tile sizes across a wide variety of benchmarks, data input sizes and platforms, visiting between 0.375% and 0.5% of the search space. However, rectangular tile sizes require a search space of 200,000 points. Therefore, in the next section, we consider the possibility to search for square tiles that reduces the size of the search space to 2000 points.

4.2. Square Tile Sizes

In this section we search for square tile sizes, reducing the size of the search space by a factor 100 to 2000 points. We have implemented four search algorithms: GA, Pyramid, Random and SA using 400 iterations and found that they reached their maximum improvement in about the same number of steps [10]. Therefore, we only present the results using Pyramid search.

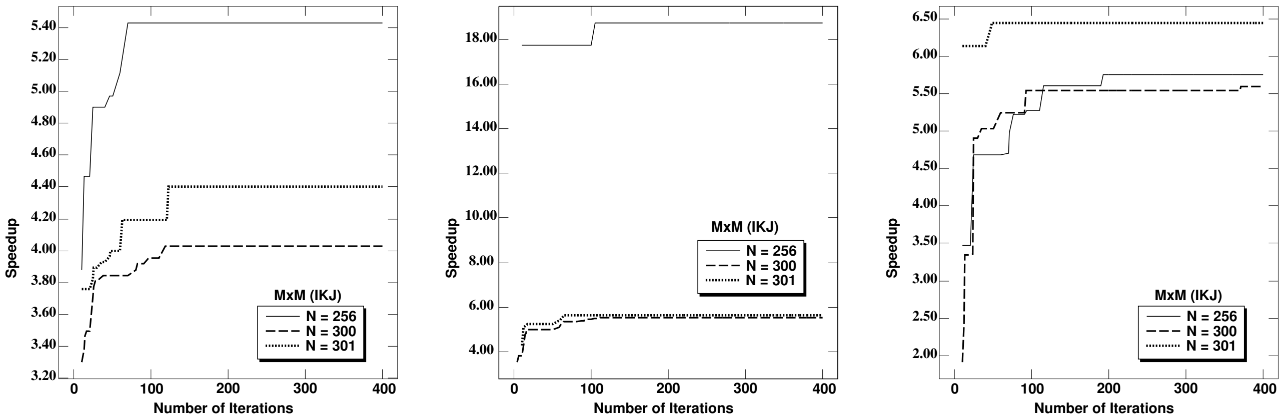
Comparing the speedups obtained using square tiles, as shown in Figure 4, with the speedups obtained using rectangular tiles shows that square tiling is within 5% on average from rectangular tiling [10]. Therefore, the difference between the different search algorithms for rectangular tiling is of the same order of magnitude as the difference between square and rectangular tiling. This shows that in our benchmark set square tiles can provide the same speedup as rectangular tiles do and therefore we can restrict attention to square tiles.

The next observation is that iterative compilation reaches high levels of optimization rapidly. The average number of iterations is 116.2. We have shown [10] that we improve by a factor of 8 over searching for rectangular tile sizes. We conclude that for the present benchmarks square tiles are almost as good as rectangular tiles, but much faster.

5. Comparison with Static Techniques

In this section we quantify the efficiency of iterative compilation by comparing the performance improvements to two static tile size selection algorithms. The first algorithm, TSS by Coleman and McKinley [7], considers the size of the working set in the loop body and requires that this working set is smaller than the cache size. It also takes into account an estimate of the cross interference between different arrays and tries to minimize this cross interference. We unrolled the loop a number of times and computed the tile size using TSS for the unrolled loop. The second algorithm, LRW by Lam, Rothberg and Wolf [14], does not consider the working set nor the cross interference rate. It computes a tile size based only on the size of the cache. We have used this tile size together with different unrolling factors.

The improvement of iterative compilation over TSS,



Pentium II HP-PA UltraSparc

Figure 4. Example: Speedup MxM – ikj version

I_{TSS}, is given by

$$I_{TSS} = \frac{S_{it} - S_{TSS}}{S_{TSS}} \cdot 100\%$$

where S_{it} is the speedup obtained by iterative compilation and S_{TSS} the speedup obtained by TSS. We compute the improvement over LRW likewise. The results are given in Figures 5 and 6.

We immediately observe that iterative compilation outperforms these static techniques significantly, up to 1800% for MxM-IKJ on the HP-PA. Note that we show that this improvement holds for each unroll factor less than or equal to 20 that the compiler might choose. In [10] it is shown that this holds for almost each benchmark, data size and platform. In the other cases only a very slight degradation of about 2% can be observed. We conclude that iterative compilation is a powerful optimization technique, outperforming existing static techniques significantly.

6. Compilation Time

In this section we discuss the compilation time required for iterative compilation that we show in Figure 7 as a function of the number of iterations, together with the breakdown in the times required for searching and program transformation in MT1, native Fortran77 compilation and execution. This relationship is almost linear. For 400 iterations, we need on average on Pentium 14.6 minutes, on HP-PA 65.23 minutes and on UltraSparc 64.49 minutes. In case of the Pentium and UltraSparc, most time is spent in executing the transformed program. However, for HP-PA, the time required for native Fortran77 compilation is the dominant factor and the time required for the global driver can be larger than the execution time of the transformed program. Hence most reduction in time required for iterative compilation can be obtained from reducing the number of actual executions, that need both native compilation and running the program.

7. Trade-off between Numbers of Iterations and Levels of Optimization

In this section we discuss how much improvement we can obtain from iterative compilation when limiting the number of iterations. Thus we investigate how much performance gain is available for a given compilation time "budget".

7.1. Comparing Improvements

In this subsection we discuss two possibilities for a metric to compare improvements. There are two properties we require this metric to have. First, if transformation T_1 has no effect, but transformation T_2 has a positive effect, we want to be able to say that T_1 reaches 0% of the improvement of T_2. Second, if T_1 results in the same running time as T_2, we want to be able to say that T_1 reaches 100% of the improvement of T_2. Both metrics below have these properties.

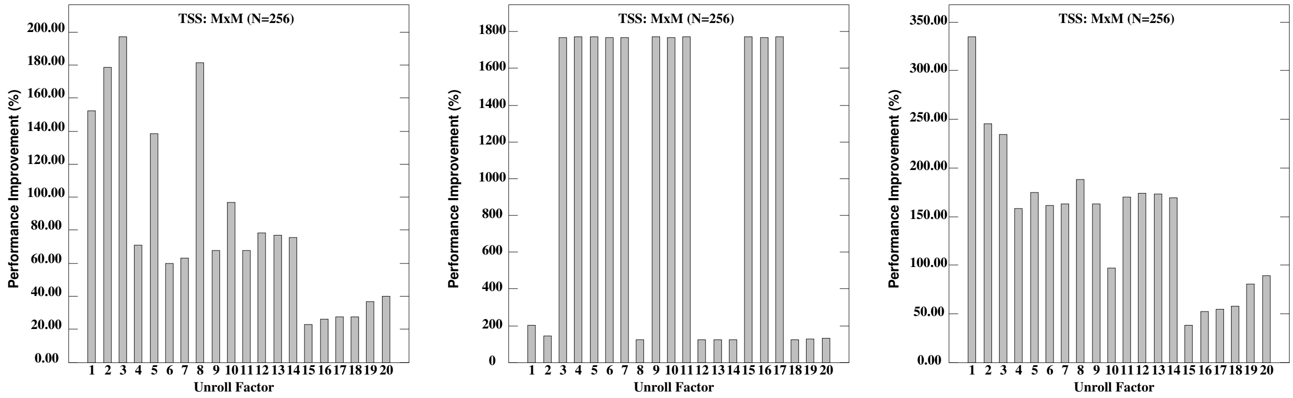
Speedup Since the minimal speedup we obtain from iterative compilation is 1, we define the metric M_S by

$$M_S(T_1, T_2) = \frac{S_1 - 1}{S_2 - 1} \cdot 100\%$$

where S_i is the speedup from transformation T_i.

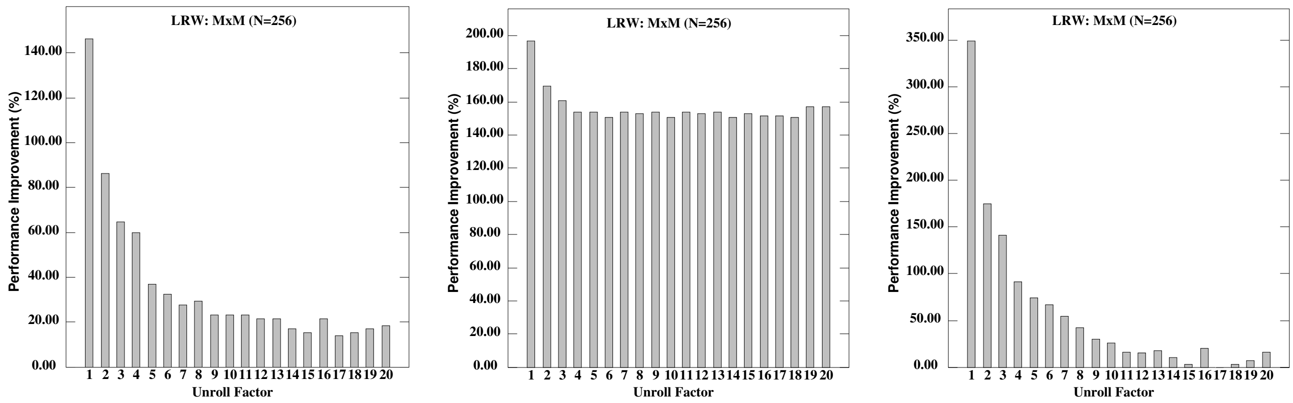
Execution time improvement The improvement I of a transformation is defined as

$$I = \frac{t_0 - t_1}{t_0} \cdot 100\%$$



Pentium II HP-PA UltraSparc

Figure 5. Example: Improvement over TSS for MxM – ikj version



Pentium II HP-PA UltraSparc

Figure 6. Example: Improvement over LRW for MxM – ikj version

where t_0 is the original execution time and t_i is the execution time resulting from transformation T_i. The metric M_I compares these improvements and is defined as

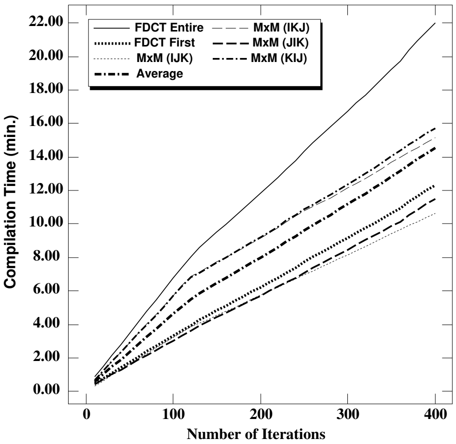
$$M_I(T_1, T_2) = \frac{t_0 - t_1}{t_0 - t_2} \cdot 100\%$$

For example, suppose we have a program that runs in 20 seconds. Suppose transformation T_1 reduces the running time to 4 seconds, and transformation T_2 reduces the running time to 2 seconds. Then $S_1 = 5$ and $S_2 = 10$, and $I_1 = 80\%$ and $I_2 = 90\%$. To compute comparisons, $M_S(T_1, T_2) = \frac{4}{9} \cdot 100\% \approx 44\%$ and $M_I(T_1, T_2) = \frac{20-4}{20-2} \cdot 100\% \approx 88\%$. Hence we see the difference between the two metrics.

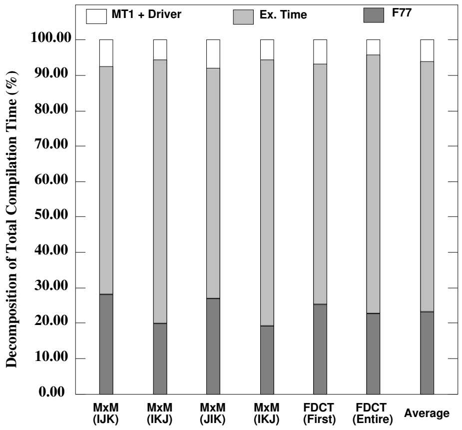
7.2. Quantitative Trade-off Graphs

We now want to compare the transformation T_i found in iteration i to the final transformation T_m with maximal effect that has been found after 400 iterations. We use the results of both the Pyramid and the Random search algorithm for square tiles, as discussed in section 4.2.

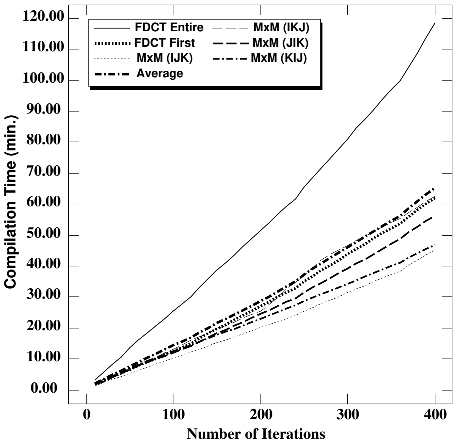
The graph in Figure 8 contains a number of *equi-optimization curves* that indicate the percentage of benchmarks that reach a certain level of optimization as a function of the number of program executions. This level is based on the two metrics from the previous subsection. It expresses how close the speedup (resp., execution time improvement) is to the maximal speedup (resp., execution time improvement) found by the iterative search procedure. From these figures we can deduce, for example, that using the Random algorithm after 100 iteration, 58% of the benchmarks were fully optimized and thus reached 100% of the speedup or execution time improvement. Likewise, we see that after 50 iterations, 85% of the benchmarks reached at least 90% of the maximal speedup and 79% of the benchmarks reached at least 97% of the execution time improvement. Conversely, given a budget of a certain amount of iterations, we can deduce the effect of iterative compilation in terms of how



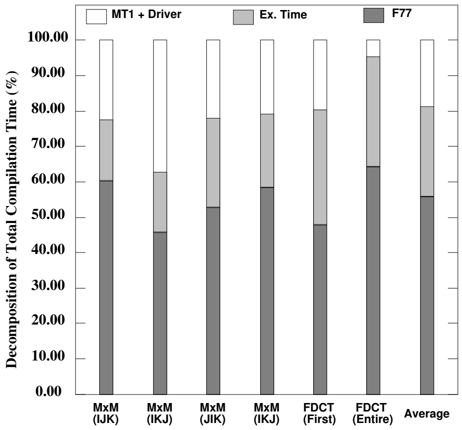
Pentium: Total time



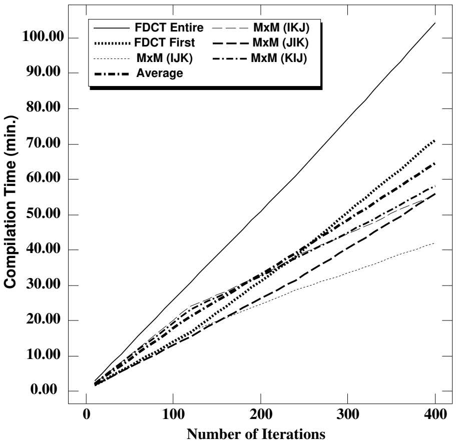
Pentium: Decomposition



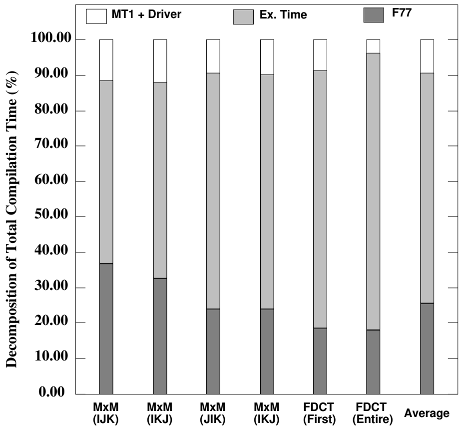
HP: Total time



HP: Decomposition

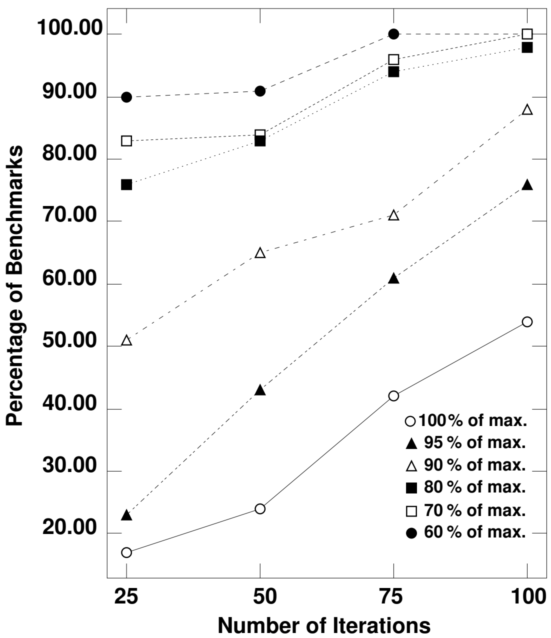


Ultra: Total time

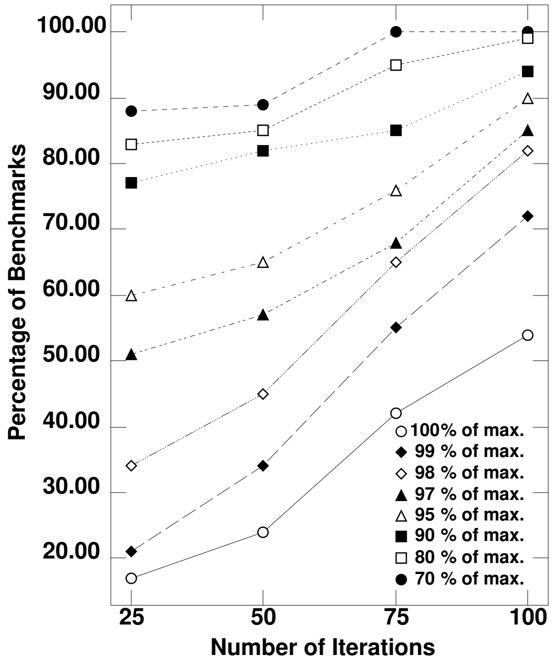


Ultra: Decomposition

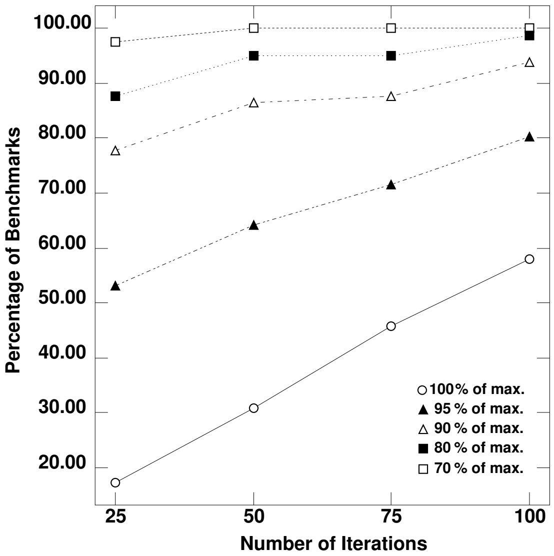
Figure 7. Compilation Time and Decomposition of Compilation Time



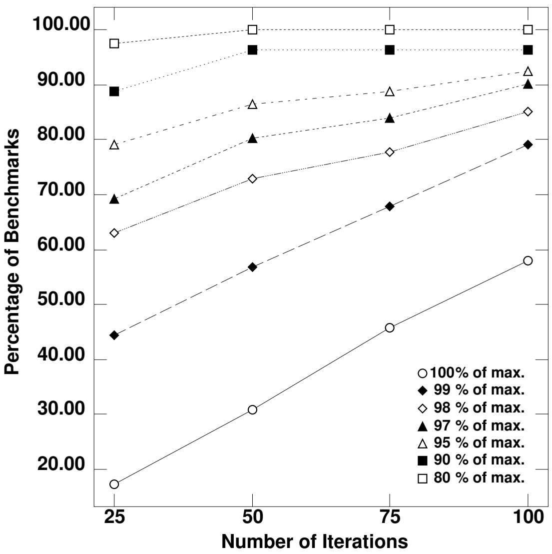
(a) Pyramid: Trade-off Speedup



(b) Pyramid: Trade-off Time Improvement



(a) Random: Trade-off Speedup



(b) Random: Trade-off Time Improvement

Figure 8. Trade-off Graphs for Pyramid search (top) and Random search (bottom)

likely it is that a certain level of improvement is reached.

There are a number of observations we can make. First, we see that after 25 iterations most benchmarks reach high levels of optimization: more than 50% reach 95% of the maximal speedup and 45% reach 99% of the maximal execution time improvement for Random search. Likewise, after 25 iterations more than 50% of the benchmarks reach 90% of the maximal speedup and 97% of the maximal execution time improvement for Pyramid search. For 50 iterations, all benchmarks reach at least 70% of their maximal speedup and 80% of their maximal execution time improvement for Random search, and 90% of the benchmarks reach 60% of their maximal speedup and 70% of their maximal execution time improvement for Pyramid search. Conversely, after 100 iterations most benchmarks reach high levels of optimization. For 50 or more iterations, the equi-optimization curves for 70% and 80% of the speedup, and the equi-optimization curves for 80% and 90% of the execution time improvement are close together. This means that there are few programs with an improvement between 70% and 80% in speedup (or between 80% and 90% in execution time improvement). The vast majority reaches a higher level of optimization.

Comparing the trade-off graphs for Pyramid search and Random search we observe that Random search reaches the highest levels of optimization quickest. One reason for this is that Random search takes samples from the entire optimization space, whereas Pyramid search evaluates a 50 point grid first. Therefore, after 25 iterations, Pyramid search only has inspected half of the search space. This shows that if we are only willing to execute a few iterations, Random search can outperform other approaches. The trade-off graph suggests that after taking 50 *random* samples, we have a high *probability* to have found a good optimization. From Figure 7 it follows that 50 iterations can be performed in less than 6.5 minutes on average and 100 iterations in less than 12.5 minutes. These compilation times can be afforded when we need to highly optimize time critical embedded applications.

8. Related Work

Over the past years, many authors have considered limited search techniques for optimization purposes. In particular, for tiling and unrolling, Coleman and McKinley [7] and Lam, Rothberg and Wolf [14] employ a restricted search for tile sizes based on a simple cache model. In [4] an improved tile size selection algorithm is presented that also uses a static searching technique. Carr [3] computes several unroll factors and chooses the best in order to minimize the difference in machine and loop balance. In contrast to these approaches, the present approach uses actual execution times and moreover considers both loop tiling and unrolling at the same time.

Whaley and Dongarra [19], and Bilmes et al. [2] describe systems for generating highly optimized BLAS routines that probe the underlying hardware to find optimal transformation parameters. They show to be capable of outperforming vendor supplied, hand optimized library BLAS routines. The present approach, however, describes a full compiler.

Wolf, Maydan and Chen [20] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. Han, Rivera and Tseng [9] also describe a compiler that searches for tile and pad sizes using static models. In contrast to the present approach, however, their compilers use static cost models to evaluate the different optimizations. From this paper it follows that our approach based on actual execution times delivers superior performance and can adapt to any architecture, requiring no prior modeling phase. Chow and Wu [5] apply 'fractional factorial design' to decide on a number of experiments to run for selecting a collection of compiler switches. They, however, focus on on/off switches and do not consider the choice of parameter values that might come from a large range of values. Bodin and co-workers explore in [18] the interplay between loop unrolling and software pipelining that can be fully integrated with the present approach since they target the code generation phase.

Over the past years, many proposals have been put forward to use profile information that are currently being employed in commercial compilers [6]. Profiles are also used to identify runtime constants that can be exploited at compile time [15]. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

9. Discussion

In this paper we have discussed iterative compilation and shown that this approach can achieve high speedups, outperforming static techniques. Although high levels of optimization can be achieved in a limited number of iterations, the obvious drawback of iterative compilation is its long compilation time, required to generate many versions of the source program and to profile these versions. In order to reduce this compilation time, we propose to add static models in the driver. These models can be used to filter out certain program executions. Only if the model predicts that some version might be better than the best one found so far, we will actually profile that version. Alternatively, the models can be used to rank many versions of the program and profiling can be used for the versions with the highest ranking.

In the long term we envisage a compilation system where the user can trade-off levels of optimization and compilation time by tuning the complexity and accuracy of static models and the number of programs that are actually executed. In this compilation model, the compiler described in [20] and the tile size selection algorithms described in [4, 9, 14, 7] use low to medium complexity models and statically evaluate many points but do not execute any program. The present paper discusses the situation where there are no models and many points are evaluated. In [11] we recently discussed the inclusion of level 1 cache models and we have shown that accurate cache models can reduce the number of program executions by 50%, compared to the approach in this paper. Currently, we are implementing more static models, including a static instruction scheduler to be used for wide issue machines.

In this paper we have considered computational kernels only. In [16] we have discussed how to apply iterative compilation to entire applications where we need many more transformations. We have shown that a tree-like approach can be efficient where tiling and unrolling is a node in this tree. By careful use of profiling information, many loop nests can be transformed in tandem. In this way we expect that we can optimize large applications within 1000 program executions. Hence, the optimization process can be done within several days. This is much faster than hand optimization of these applications that may take months and, moreover, iterative compilation is an automatic procedure requiring no human interaction. Therefore, iterative compilation can be competitive for time critical embedded applications.

10. Conclusion

In this paper we have discussed how to simultaneously select tile sizes and unroll factors using a novel approach to program optimization, namely, iterative compilation. This approach generates many transformed versions of the source program and searches for the best by compiling and executing these programs. We have implemented this approach and we have shown that it is able to find high levels of optimization rapidly, outperforming existing approaches significantly. We have shown how to trade-off compilation time and levels of optimization by limiting the number of iterations. In this way, within 50 iterations, high levels of optimization on average can be found.

References

[1] A.J.C. Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Transformation mechanisms in MT1. Technical Report 2000-21, LIACS, Leiden University, 2000. Available through `www.liacs.nl/~peterk`.

[2] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS'97*, pages 340–347, 1997.

[3] S. Carr. Combining optimization for cache and instruction level parallelism. In *Proc. PACT'96*, pages 238–247, 1996.

[4] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *Proc. ICS'99*, pages 492–499, 1999.

[5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizatons. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999. Organized in conjunction with MICRO 32.

[6] R. Cohn and P.G. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999. Organized in conjunction with MICRO 32.

[7] S. Coleman and K.S. McKinley. Tile size selection using cache organization and data layout. In *Proc. PLDI'95*, pages 279–290, 1995.

[8] H. Corporaal. *Microprocessor Architectures: From VLIW to TTA*. John Wiley, 1997.

[9] H. Han, G. Rivera, and C.-W. Tseng. Software support for improving locality in scientific codes. In *Proc. CPC2000*, pages 213–228, 2000.

[10] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. Technical Report 2000-07, LIACS, Leiden University, 2000. Available through `www.liacs.nl/~peterk`.

[11] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Incorporating cache models in iterative compilation for combined tiling and unrolling. Technical Report 2000-10, LIACS, Leiden University, 2000. Available through `www.liacs.nl/~peterk`.

[12] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, F. Bodin, and H.A.G. Wijshoff. A feasibility study in iterative compilation. In *Proc. ISHPC'99*, volume 1615 of *Lecture Notes in Computer Science*, pages 121–132, 1999.

[13] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H.A.G. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC2000*, pages 35–44, 2000.

[14] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS'91*, pages 63–74, 1991.

[15] M. Mock, M. Berryman, C. Chambers, and S.J. Eggers. Calpa: A tool for automating dynamic compilation. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999. Organized in conjunction with MICRO 32.

[16] M.F.P. O'Boyle, N.P. Motogelwa, and P.M.W. Knijnenburg. Feedback assisted iterative compilation. Technical Report 012, Division of Informatics, Edinburgh University, 2000.

[17] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proc. 8th Int'l Conf. on Compiler Construction*, 1999.

[18] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline – unrolling tradeoffs. In *Proc. SCOPES99*, 1999.

[19] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance 98*, 1998.

[20] M.E. Wolf, D.E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. *Int'l. J. of Parallel Programming*, 26(4):479–503, 1998.