Iterative Compilation in Program Optimization

T. Kisuki[†] P.M.W. Knijnenburg[†] M.F.P. O'Boyle[‡] H.A.G. Wijshoff[†]

[†]Dept. of Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, the Netherlands
{kisuki,peterk,harryw}@cs.leidenuniv.nl

[‡]Institute for Computing Systems Architecture
the University, Edinburgh EH9 3JZ, U.K.
mob@dcs.ed.ac.uk

Abstract

In this paper we describe a novel approach to program optimization, namely, iterative compilation. This technique enables compilers to deliver efficient code by searching for the best sequence of optimizations using profile-directed feedback. We have implemented this iterative approach in an existing compiler system which is designed for embedded systems and compared its performance to two of the best known static techniques. Initial experimental results show that this approach delivers an average improvement of 35% over existing techniques and delivers efficient code with reasonable compilation times.

1. Introduction

Modern compilers make extensive use of optimization to improve program performance. The use of a particular optimization largely depends on static program analysis based on simplified machine models. Optimizations include high-level loop transformations, such as loop unrolling and tiling. These techniques are extensively studied for over 30 years and have produced, in many cases, good results. However, the machine models used are inherently inaccurate, and transformations are not independent in their effect on performance making the compilers task of deciding the best sequence of transformations difficult. Typically, compilers use heuristics that are based on averaging observed behavior for a small set of benchmarks. Furthermore, while the processor and memory hierarchy is typically modelled by static analysis, this does not account for the behavior of the entire system. For instance, the register allocation policy and strategy for introducing spill code in the back-end of the compiler may have a significant impact on performance. Thus static analysis can improve program performance but is limited by compile-time decidability.

Recently, we have investigated aggressive optimization techniques for embedded systems where a different approach to optimization, namely *iterative compilation*, has been adopted [3]. In this approach, successive transformations are applied to a program and their worth determined by actually executing the resulting code. Such an approach does not suffer from undecidability issues and, given sufficient time, will find the optimal program. The obvious drawback is that compilation time dramatically increases. In the case of embedded applications, however, only one program is to be executed and the cost of compilation can be amortised over the number of systems shipped and the lifetime of the application. In such applications, performance is critical. Hence the long compilation times for iterative compilation can be afforded.

Approaches to optimization based on searching have recently received attention from other authors. Whaley and Dongarra [20], and Bilmes et al. [6] describe systems for generating highly optimized BLAS routines. These systems probe the underlying hardware to find optimal values for blocking factors, unroll factors etc. Experimentation has shown [6, 20] that these systems are capable of producing more efficient code than the

vendor supplied, hand optimized library BLAS routines. In contrast to the present approach, however, these systems are only able to optimize BLAS routines and are not general purpose compilers.

Wolf, Maydan and Chen [21] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. In contrast to the present approach, however, their compiler uses a static cost model to evaluate the different optimizations. We believe that the present approach based on actual execution times will deliver superior performance.

Chow and Wu [8] 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.

Over the past years, many proposals have been put forward to use profile information, for example, in the creation of superblocks [12] or hyperblocks [15] to enable efficient scheduling for ILP processors. These techniques are currently being employed in commercial compilers [9]. Profiles are also used to identify runtime constants that can be exploited at compile time [16]. The recently established workshop on Feedback Directed Compilation shows that currently many proposal are being put forward to exploit profile information in the compiler chain. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

Finally, within the OCEANS project, other approaches to iterative compilation are considered. In [19] the interplay between loop unrolling and software pipelining is explored. In [17] a genetic algorithm approach to searching is proposed.

In order to assess the feasibility of iterative compilation, we have conducted several experiments [7, 13]. We generated all versions of three important linear algebra routines using loop unrolling and tiling, and executed them on 7 different platforms. We used a grid-based search algorithm (the same as the search algorithm in the present study) to see how fast a search would find acceptable levels of optimization. We found that by visiting only 2.5% of the search space we could obtain an optimization that comes within 5% of the absolute minimum of this search space.

From these feasibility studies, we concluded that iterative compilation is a feasible and effective approach. In this paper we discuss the implementation of iterative compilation and report the initial results of this approach. We also discuss the complexity of the search procedure by examining the running times of the compiler.

This paper is organized as follows. In section 2 we discuss how iterative compilation is implemented. In section 3 we discuss our experimental setup. In section 4 results are discussed including performance improvement and actual compilation time. Finally, in section 5, we draw some concluding remarks.

2. Iterative Compilation

In this section we discuss how the iterative compilation system is implemented. Figure 1 shows an overview of the compiler system.

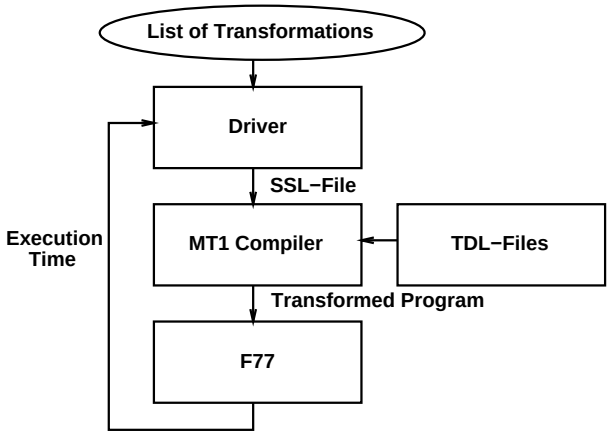

Figure 1. The Compilation Process

The compilation system is centred around a global driver. This driver keeps track of the different transformations evaluated so far and decides which transformations to apply next. The driver reads a list of transformations that it needs to examine together with the range of their parameters. For example, this list contains the entry `Unroll 1 - 20`. The driver uses an N dimensional array when N different optimizations need to be examined that represents the transformation space. Each point in this array corresponds to a specific set of parameters for the transformations. It is the role of the driver to search for the optimal parameters for each transformation.

The algorithm used by the driver searches the transformation space, where each axis of the space corresponds to a particular transformation and each point along an axis corresponds to a particular transformation parameter value. By placing an integer lattice or grid over this space and visiting lattice points, the algorithm has

a means of navigating the transformation space and determining the best parameters for all transformations. In [7, 13] we have shown that this algorithm can reach high levels of optimization in a relatively small number of steps. The grid search algorithm can be briefly described as follows.

1. Define a coarse grid on the search space. Evaluate all points on this grid by generating the transformed programs and executing them.
2. Find the point with minimum execution time and all points that are within an allowable distance from this minimum.
3. Order these points in a priority queue, ordered by execution time.
4. For each point in the queue,
   - if the execution time associated with this point is within an allowable distance from the minimum found so far, refine the grid around this point by forming a new grid with half the spacing in each dimension.
   - If new points are found that are close to the minimum found so far, enqueue them in the priority queue.

Using this algorithm the search space is traversed to find optimal parameters for the transformations.

The global driver invokes the source to source compiler MT1 [5]. MT1 has two mechanisms to control which transformation it applies to the input program: a Transformation Definition Language TDL [4] and a Strategy Specification Language SSL [2].

For each transformation included in the list of transformations used by the global driver, a transformation needs to be specified in the TDL-file. This file needs to be given beforehand by the user. A transformation given in the TDL has the following general format:

```
TRANSFORMATION
   name
TRANSFORM
   input pattern
INTO
   output pattern
CONDITION
   conditions
```

Although the TDL is pattern-matching based, it allows the inclusion of user defined functions that may examine and change the internal program representation. In this way, almost any transformation can be specified.

Next, in order to instruct MT1 to apply a specific sequence of transformations, the global driver constructs an SSL file that specifies the order in which to apply certain transformations.

Hence one step of the global driver consists of the following steps:

1. Decide the next set of parameters for the transformations using its internal search space and the search algorithm.
2. Construct an SSL file that corresponds to this new sequence.
3. Invoke MT1 that starts the transformation process by reading in the source program, the SSL file and the TDL file.
4. The transformed program is compiled for the target architecture and executed.
5. The execution time is measured and reported back to the global driver.
6. The global driver stores this execution time and starts the next step.

Finally, after a predetermined number of iterations, the global driver stops searching and outputs the transformed program with the shortest execution time.

3. Experimental Setup

In our experiments into the efficiency of iterative compilation, we use three transformations and five benchmarks. We have selected the following transformations [1]:

1. Loop Tiling, with tile sizes 1 to 100;
2. Loop Unrolling, with unroll factors 1 to 20;
3. Array Padding, with pad sizes 1 to 10.

The driver applies these transformations in this order and searches for the optimal tile size, unroll factor and pad size. We studied the behavior of the search algorithm in two cases: using unrolling and tiling, and using unrolling, tiling and padding.

We selected five benchmarks: three general purpose basic linear algebra routines and two routines from multimedia applications. Each benchmark is optimized using three input data sizes.

1. Matrix-Matrix Multiplication (MxM), data sizes 256, 300 and 301;
2. Matrix-Vector Multiplication (MxV), data sizes 2048, 2300 and 2301;
3. Successive Over Relaxation (SOR), data sizes 128, 150 and 151;
4. Forward Discrete Cosine Transform from mpeg2 (FDCT), data sizes 256, 300 and 301;
5. Motion compensation routine from H263 (RECO), data sizes 2048, 2300 and 2301.

As our target platform we choose the Pentium II at 233 MHz. We used the Fortran compiler g77 (version 0.5.19.1) with optimization flag `-O` on.

4. Results

In this section we discuss the results obtained by the iterative compilation approach. We also discuss the overall compilation time.

4.1. Single Data Size

We have executed the search algorithm on all benchmarks and for three fixed input data sizes. In Figures 2 and 3 we have shown the performance improvement of our technique over the original execution time. The x-axis shows the number of iterations (or compile/execute cycles) and the y-axis shows the speedup over the original program. We restrict the number of iterations to 400 in each case, since our previous research [7, 13] showed that it is likely that within this number of iterations high levels of optimization are obtained. Table 1 shows the optimal parameters for the transformations found in the search.

First, in Figure 2 loop tiling and loop unrolling are applied as transformations. The first observation is that, except in the case of SOR, the search algorithm finds good speedups, up to a speedup of 3.4 in the case of MxV. SOR is not improved much because this kernel the transformations that we currently examine do not apply to it (c.f. [10]). Note that the platform used in this study only has a small issue width. Hence the effect of loop unrolling to expose ILP is limited. Another observation is that the search algorithm finds good parameters quickly. Within 50 evaluations in all cases except for MxM and SOR the performance improvement is close to maximum. For MxM and SOR more than 100 evaluations are required to obtain a good performance improvement. This corresponds ot 5% of the entire search space. After 300 evaluations, there is no performance improvement observed in any case. This corresponds to 15% of the entire search space.

Second, Figure 3 shows the performance improvement when array padding is also considered. This enlarges the transformation space by a factor of 10. Comparing Figures 2 and 3, we can see that although the search space is larger comparable speedups are obtained within the same number of iterations. In case of MxV, a significantly larger speedup is found. In other cases (e.g., MxM) a slightly smaller improvement than in the previous case is found. 350 evaluations are required to obtain comparable or better results than in the first case. However, in this case, this number of evaluations corresponds to only 1.75% of the entire search space. Hence we do not observe a scaling up of the number of iterations required with the size of the search space.

Third, table 1 shows the best parameters for the transformations found within 400 iterations and the iteration in which these values have been found. From this table, we can observe that the best parameters highly depend on the input data size. The effect of interference among transformations can also observed from this table.

4.2. Multiple Data Sizes

In the previous section we have shown that our iterative approach to optimization yields significant speedups for the case of fixed input data sizes. Hence, if profiling shows that the kernel is heavily biased towards a certain input data size, this approach can be used. However, in many cases profiling will yield a distribution of input data sizes. In this case we cannot simply optimize for one single data input size. Instead we need to optimize the program so that the average execution time is minimized.

To deal with this situation, we execute the transformed program on several data input sizes. We collect the different execution times and compute the weighted average where weights correspond to the distribution of data sizes obtained from profiling. The search proceeds exactly like before, yielding an optimization that minimizes the average execution time.

We have conducted two experiments to assess this approach. We used unrolling, tiling and padding as transformations. We did not consider SOR as a benchmark since for this kernel only very small improvements can

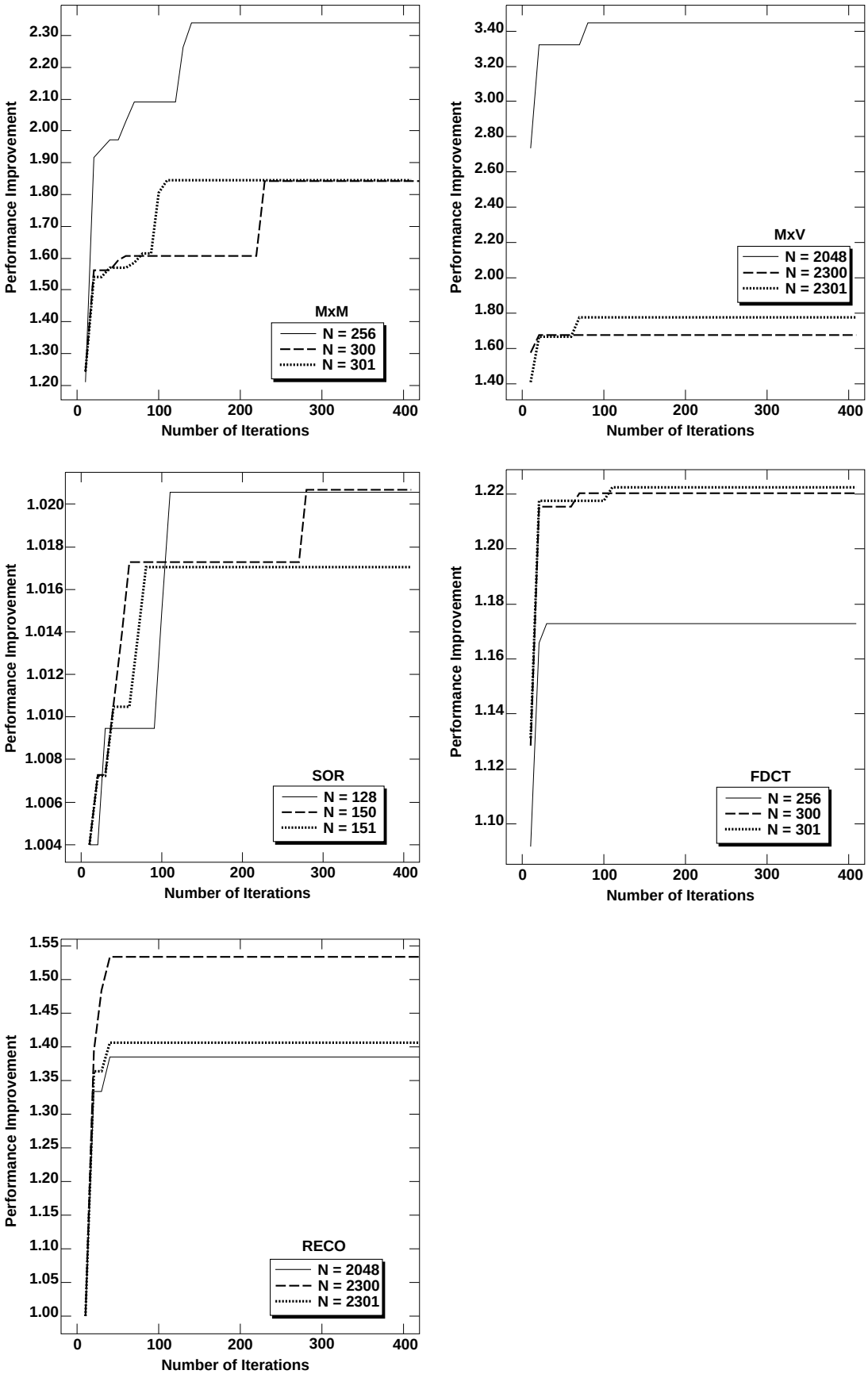

Figure 2. Performance Improvement using Unroll and Tile

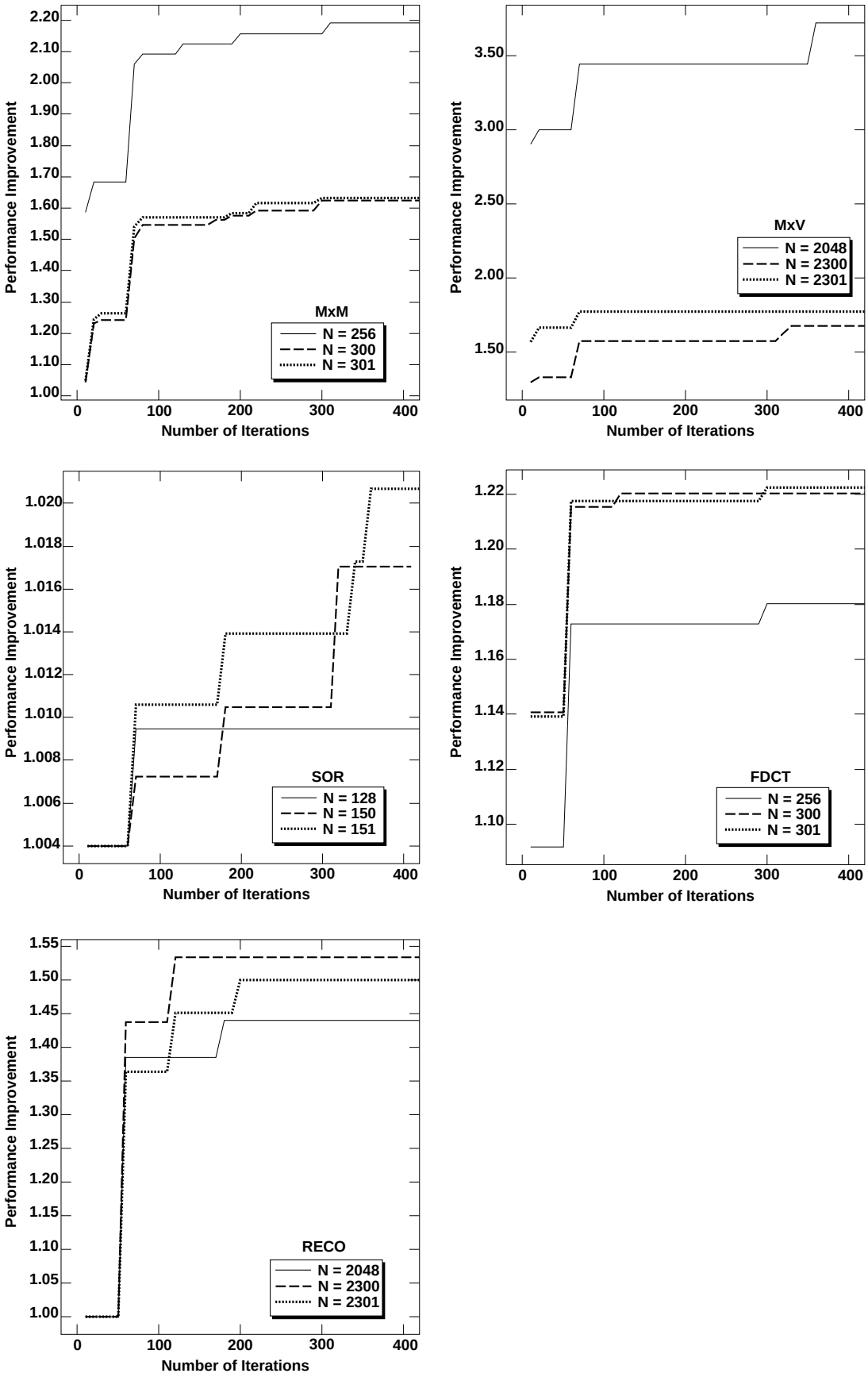

Figure 3. Performance Improvement using Unroll, Tile and Padding

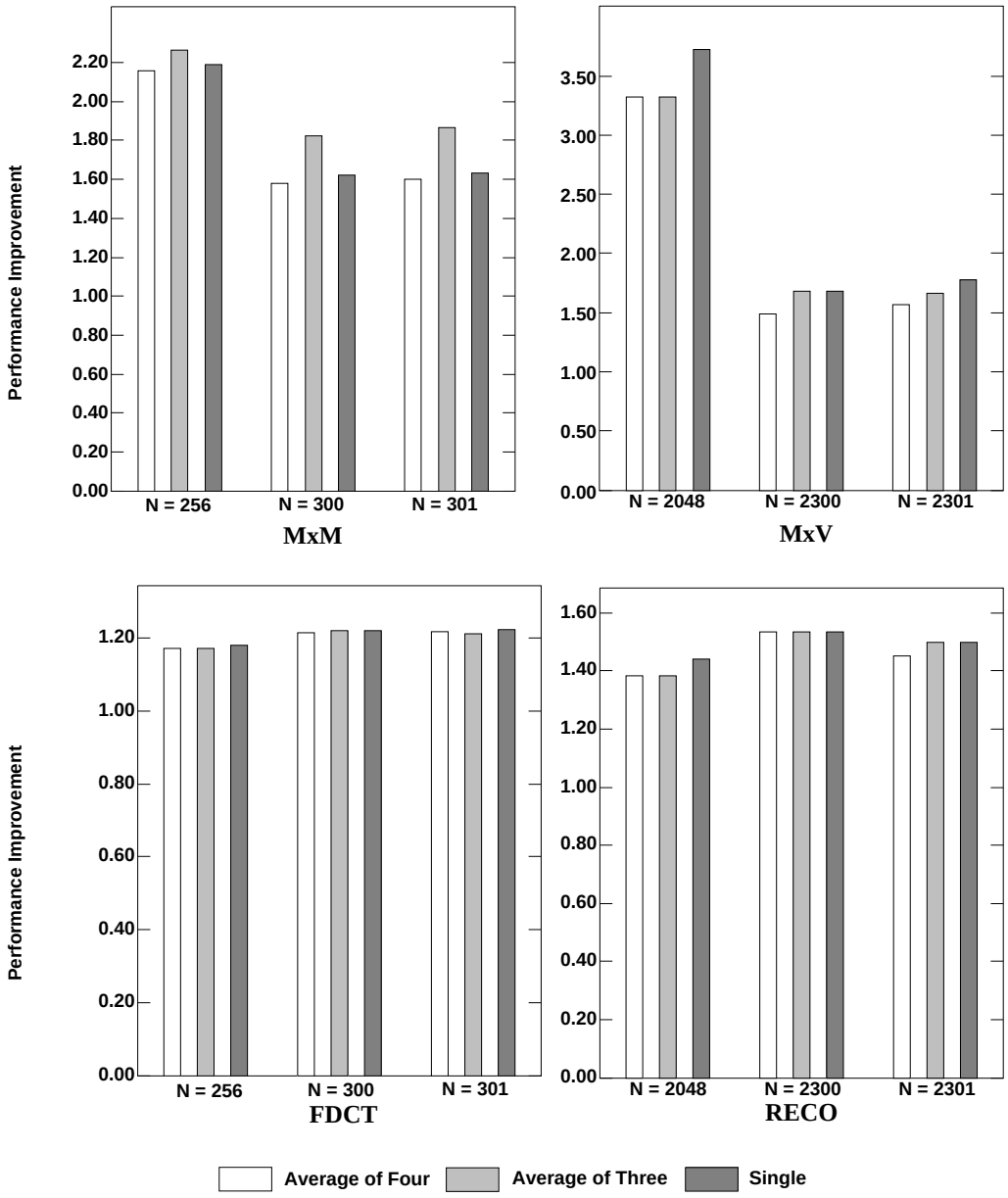

Figure 4. Comparison of Performance Improvement using Multiple Data Sizes

	Original	T & U				T & U & P				
	Space	2000				20000				
	Time	Tile	Unroll	Iteration	Time	Tile	Unroll	Pad	Iteration	Time
MxM										
256	1.38	4	15	299	0.59	51	13	5	299	0.63
300	1.64	32	13	228	0.89	99	13	3	292	1.01
301	1.68	32	13	102	0.91	99	13	3	290	1.03
MxV										
2048	0.93	16	9	73	0.27	16	7	5	354	0.25
2300	0.52	46	6	160	0.31	19	6	8	364	0.31
2301	0.55	32	7	117	0.31	21	6	7	68	0.31
SOR										
128	1.84	40	17	109	1.81	11	6	3	61	1.83
150	3.05	59	18	275	3.00	26	17	2	353	3.00
151	3.11	21	17	71	3.07	11	17	1	310	3.07
FDCT										
256	1.90	1	11	22	1.62	1	5	2	297	1.61
300	3.16	1	9	66	2.59	1	11	5	112	2.59
301	3.19	1	5	106	2.61	1	5	4	296	2.61
RECO										
2048	0.36	11	16	34	0.26	11	16	3	171	0.25
2300	0.46	76	16	365	0.30	71	15	4	315	0.30
2301	0.45	99	16	43	0.32	41	17	3	312	0.30

Table 1. Best Parameter Values
T = Tiling, U = Unrolling, P = Padding, Time = Execution Time in Seconds

be obtained. We expect that the speedups obtained when using many data sizes for optimization are less than the speedups obtained running the program on one data size. However, results reported below show that this is not always the case.

First, we searched for an optimization using input data sizes 256, 300 and 301 for MxM and FDCT, and 2048, 2300 and 2301 for MxV and RECO. We obtained a transformed program that was ran again on these data sizes. We measured the speedup and compared this with the speedup that we obtained in section 4.1. The results are given in Figure 4. The grey bar corresponds to the speedup found using the average and the black bar is the speedup found in section 4.1. We see that in the case of MxM even a better speedup is obtained than previously. In the other cases, a slight degradation can be observed. However, this degradation is small.

In order to check the stability of our technique over a wider number of data sizes, we searched for an optimization using four different data sizes: 250, 260, 290 and 310 for MxM and FDCT, and 2000, 2100, 2200 and 2400 for MxV and RECO. Once again,we ran the transformed program on the old data sizes and compared the speedup obtained with the speedup from section 4.1. The results are given in Figure 4. The white bar corresponds to the speedup found using the average. We see that in most cases the speedups found are slightly smaller than those found in section 4.1, but not much. We see that in this case the speedups found are also slightly less than in the first case above. Apparently, optimizing using the same data sizes as when executing the optimized program is better than using different data sizes for optimization.

We conclude that although different optimizations are found using the multiple data size approach, the resulting program still yields significant speedups. It appears that there many values for the parameters of the transformations that yield good speedups. The driver always finds a set of these good values. Hence we conclude that the technique of searching produces stable results in the sense that the optimization it finds is effective for a range of input data sizes.

4.3. Using Small Input Sizes

The time required for iterative compilation largely depends on the execution time of the transformed program, as we show in Section 4.5 below. The execution time, in turn, depends on the sizes of the arrays accessed in the program. In this section we consider the possibility to use small arrays to optimize programs that will use large arrays in reality thereby reducing

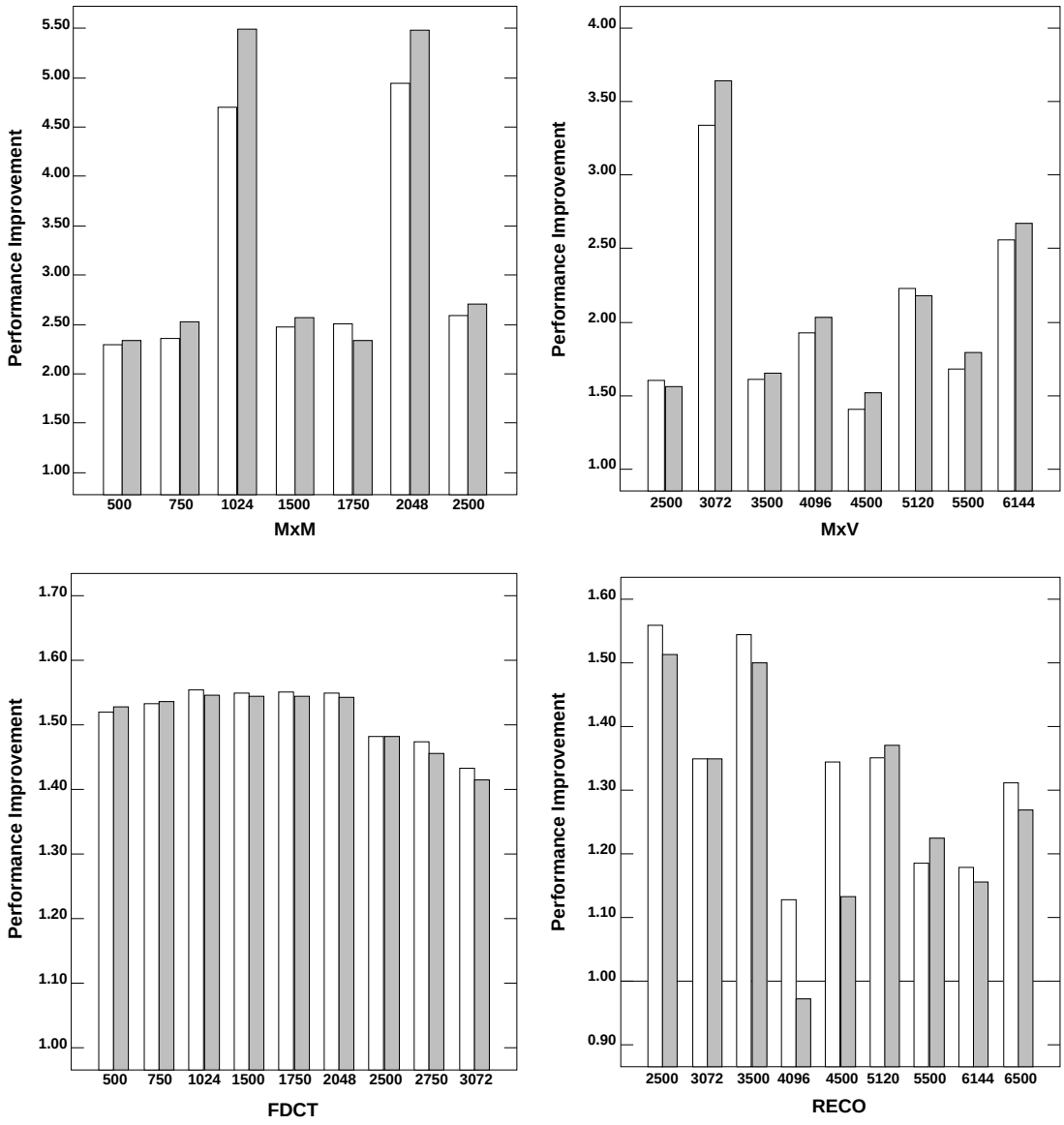

Figure 5. Performance Improvement using Small Data Sizes. White bar corresponds to tiling and unrolling. Grey bar corresponds to tiling, unrolling and padding.

the compilation time we would need when we would use these large arrays in the optimization process.

In order to gain more insight in this heuristic, we took the programs that have been optimized with input sizes $N = 256$ for MxM and FDCT, and input size $N = 2048$ for MxV and RECO. We ran these optimized programs on several larger data input sizes. We used the programs that have been optimized using unrolling and tiling as well as the programs that have been optimized using unrolling, tiling and padding. The results are shown in Figure 5. In this figure, the white bar corresponds to tiling and unrolling, and the grey bar corresponds to tiling, unrolling and padding.

From the figure we observe that all benchmarks, except RECO, can be optimized well by using small data sizes. For each data input size, the speedup is at least as large as the speedup obtained previously. In case of MxM, we see that data input sizes that are powers of two ($N = 1024$ and $N = 2048$) give rise to speedups that are twice the speedup obtained for $N = 256$. In the other cases, the speedup is about the same as the speedup for $N = 256$ and is larger than the speedup obtained for $N = 300$ and $N = 301$ that are non powers of two also. In case of MxV, again the data sizes that are powers of two perform very well, although not as good as was the case for MxM. The other data sizes reach a speedup that is about the same as the speedups obtained earlier for $N = 2300$ and $N = 2301$. In case of FDCT, we reach for each data input size a speedup that is larger than the speedups obtained for small data input sizes. In this case we do not observe a correlation between speedup and whether the data size is a power of two. Finally, RECO shows erratic behavior and even has a slow down for one case. In some cases we have a speedup that is the same as previously and in other cases it is smaller. In this benchmark we observe that adding padding as a transformation in general gives worse performance than without padding.

We conclude that the heuristic of using small data input sizes to reduce compilation time in general seems to give acceptable results, in some cases outperforming the speedup obtained with the small data size. MxM suggests that there might be a correlation between the size of the data used to optimize, the size of the data used in reality and the cache size. RECO shows that we need to study this correlation in more depth in order to use the heuristic to good effect.

4.4. Comparison with Static Techniques

In this section we address the question how efficient our iterative techniques is by comparing it to a number of 'standard' values that a compiler might choose for the parameters and to two well-known static tile size selection algorithms. The first algorithm has been proposed by Coleman and McKinley [10] and the second by Lam, Rothberg and Wolf [14]. Rivera and Tseng have shown that these algorithms are among the best known tile size selection algorithms [18].

4.4.1 Standard Parameter Values

In Figure 6 we have shown the improvement of iterative compilation over a number of 'standard' values for the unroll factor and the tile size. We considered unroll factors of 2, 4, 8 and 16, and tile sizes of 4, 8, 16 and 32. These values were chosen because they seem reasonable: small unroll factors and tile sizes that correspond to cache line sizes, from 4 to 64 lines per tile. We see that in each case (except unroll factor of 1 for MxV) iterative compilation outperforms these standard settings. In some cases, in particular for MxM, large improvements can be observed. This can be explained by the fact that MxM can be substantially improved using loop unrolling and tiling. If correct unroll factors and tile sizes are chosen, the improvement is much larger than when other factors are selected. In the other cases, loop unrolling and tiling cannot improve the loop to a large extend. This is the case, for instance, for FDCT where an output dependence sequentializes the loop body and loop unrolling cannot expose high levels of ILP to the hardware. Here the performance improvement is low (as can also be observed in section 4.1) and searching for optimnal parameters will not result in much better performance. Nevertheless, iterative compilation does find better unroll factors and tile sizes than just fixed settings. In the case of MxV, the transformations exploit the temporal reuse of the vector and it does not seem to matter much whether small or large portions of this vector are exploited. In all cases, speedups over 3 are obtained.

We conclude that iterative compilation is capable of outperforming simple fixed values for the parameters. However, the experiments also show that care must be taken when to invoke the iterative compilation process. We must be sure that the transformations under consideration can indeed improve the target code significantly and that the choice for the parameters will influence the efficiency of the resulting code to a large extend, as is the case for MxM. Under these conditions iterative compilation is capable of making a far better choice than standard settings can.

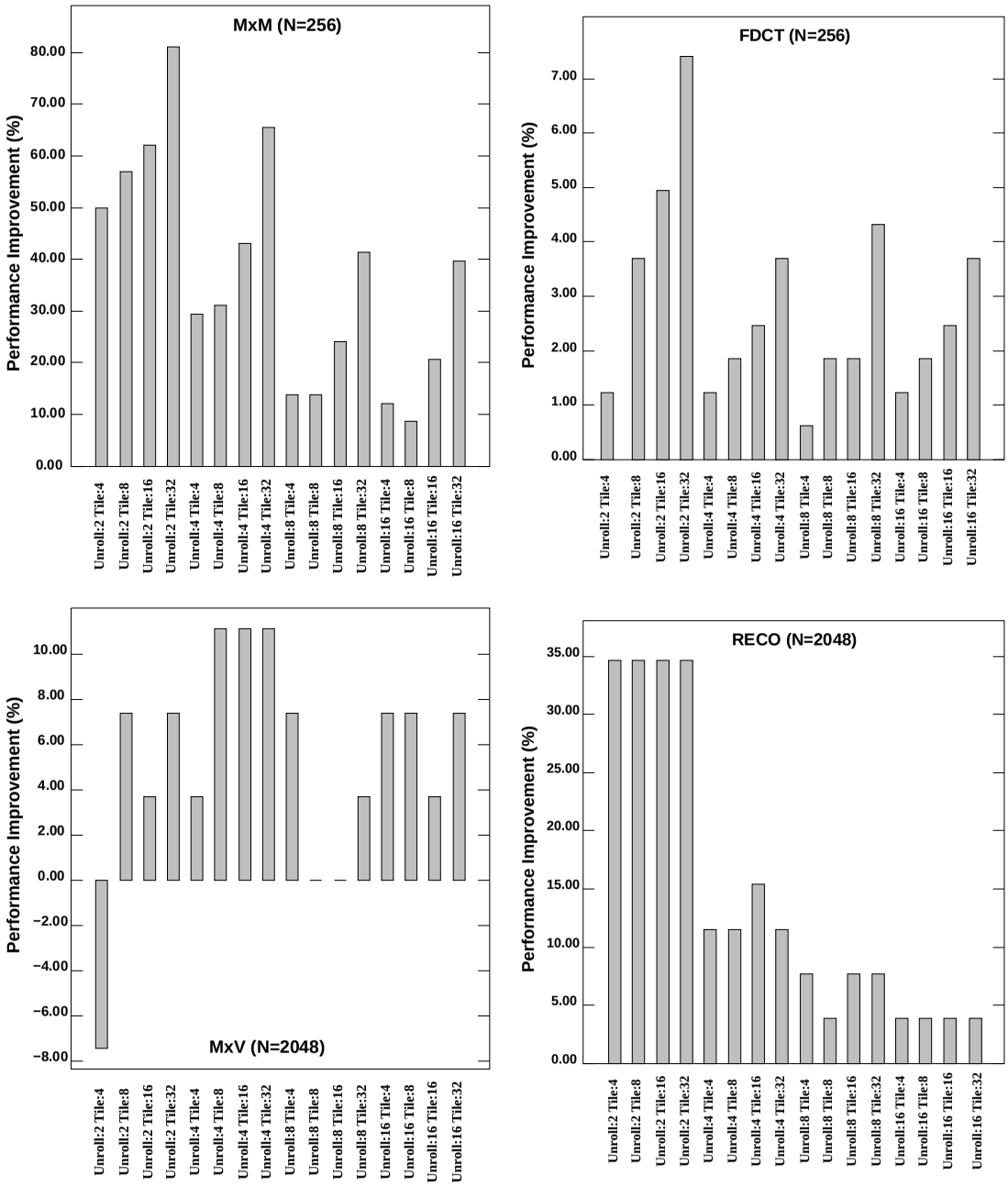

Figure 6. Improvement of Iterative Compilation over Standard Values

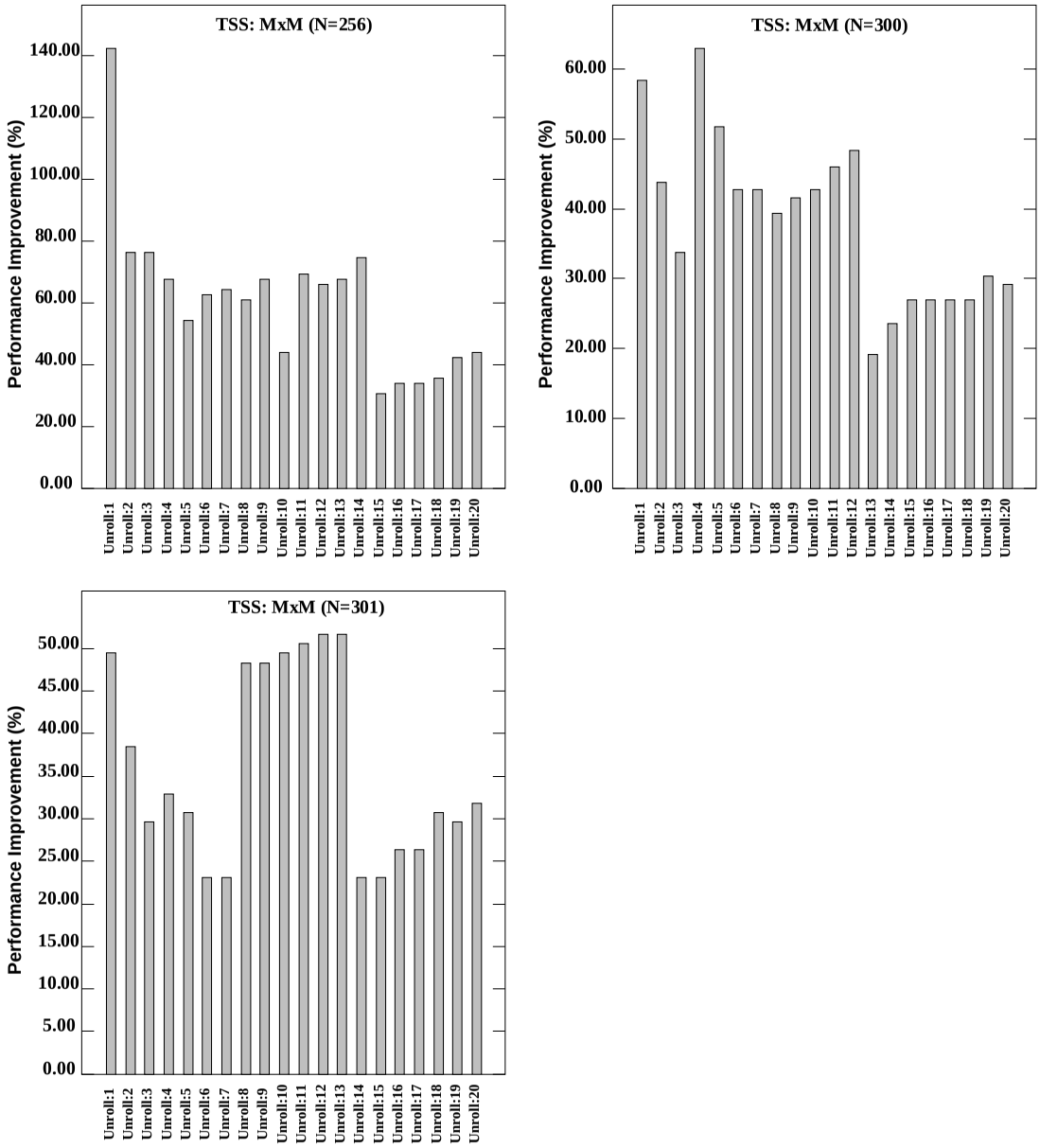

Figure 7. Improvement of Iterative Compilation over TSS

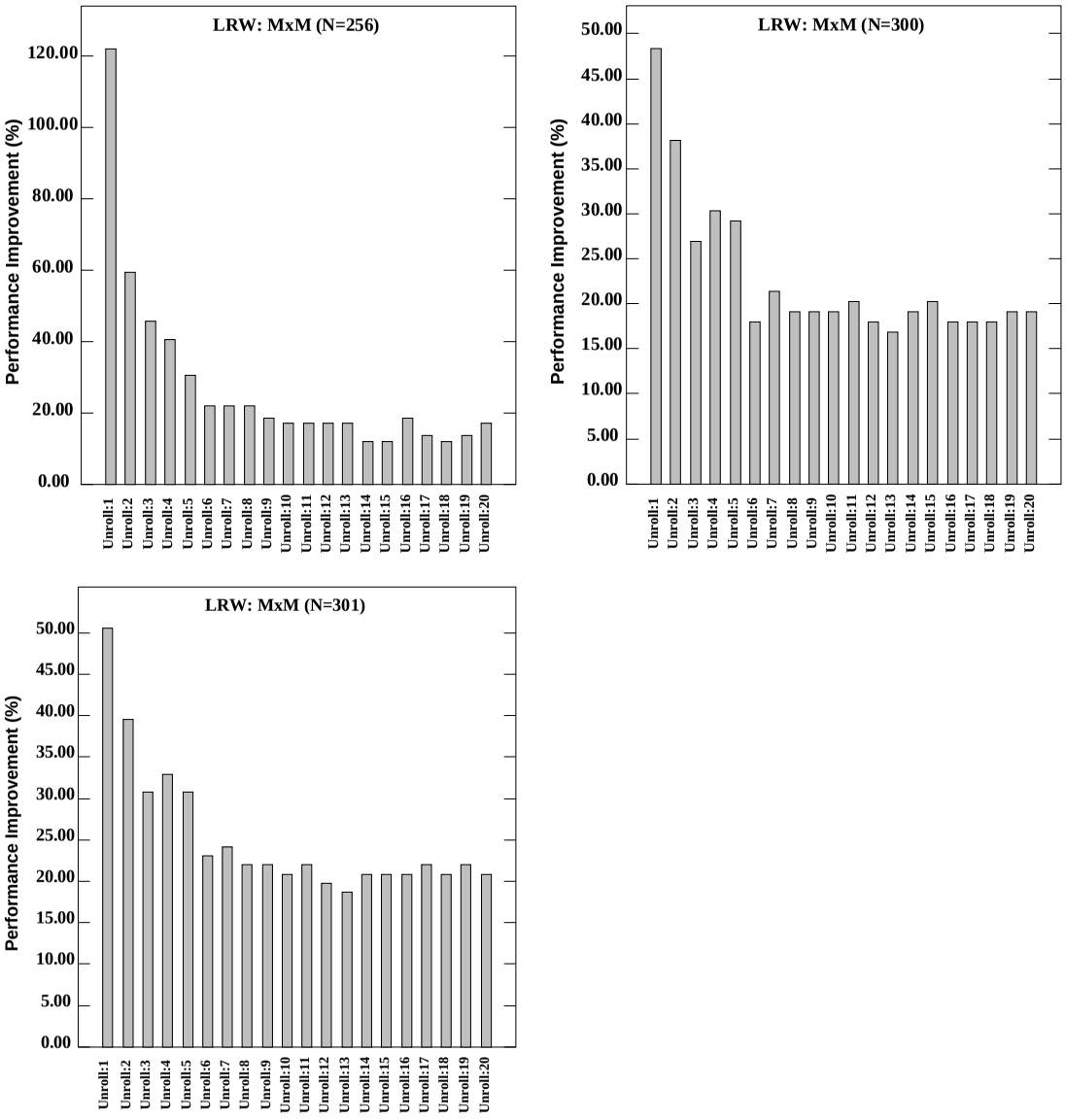

Figure 8. Improvement of Iterative Compilation over LRW

4.4.2 Static Tile Size Selction

In this section we turn attention to two important tile size selection algorithms. We show that iterative compilation outperforms both techniques. We restrict attention to matrix-matrix multiplication since it follows from the previous section that for this benchmark it matters significantly which tile size and unroll factor is chosen. Hence this benchmark is ideally suited to compare iterative compilation with other techniques.

The TSS algorithm by Coleman and McKinley [10] 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 LRW algorithm 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.

In Figures 7 and 8 we have shown the improvement of iterative compilation over TSS and LRW, respectively. In these figures we have considered matrix-matrix multiplication for three input sizes. We see that for each unroll factor the compiler might choose and corresponding tile size, iterative compilation outperforms the static technique significantly. Our technique provides, on average, a 44% improvement over TSS and a 25% improvement over LRW and the improvement in speedup is always at least 20%. In particular, we see that if the compiler would have chosen no unrolling (unroll factor of 1), the speedup obtained by iterative compilation is 140% better than the speedup obtained from TSS, and 120% better than the speedup obtained from LRW. If the compiler would have chosen an unroll factor of 4 (like the SGI compiler standardly does), iterative compilation improves TSS over 60% for input sizes 256 and 300, and over 30% for input size 301. In this case, iterative compilation improves LRW over 30% for each input size.

From these figures we conclude that iterative compilation is capable of outperforming static techniques significantly. This shows that the long compilation times for iterative compilation can pay off in terms of performance of the compiled code. Certainly in the case of embedded applications the improvements over static techniques are strong enough to warrant this approach.

4.5. Compilation Time

An important consideration for the feasibility of iterative compilation is the running time of the approach. Figure 9(a) shows the average compilation time for each benchmark and the average of these times.

We observe that compilation time is proportional to the number of iterations. The average compile time using 400 iterations ranges from 7.7 minutes (MxV) to 25.4 minutes (FDCT). On average we need 16 minutes for 400 iterations. Note that the kernels we used are representative for embedded kernels and in this case this amount of time can easily be afforded. In fact, the figure shows that for time-critical kernels many more iterations can be afforded: since compilation time can be seen as an integral component of the total development time of the embedded system, we can afford several hours to heavily optimize the compute intensive routines.

In Figures 9(b) and 9(c) we have given the breakdown of the execution times for 400 iterations. On average, about 50% of the total compilation is spent executing the transformed code. The time needed for native f77 compilation is not large, but significant. Note that in case the target platform needs static instruction scheduling or software pipelining, this time can be much larger. In some cases, the time for MT1 and the global driver is larger than the execution time of the transformed code (RECO).

5. Conclusions and Future Work

In this paper we have described a new approach to program optimization, namely iterative compilation. We have shown that this approach is able to find good optimizations by visiting a relatively small fraction of the entire optimization space. In the case where loop unrolling, tiling and array padding are considered, 350 evaluations are required to find a satisfactory optimization, which corresponds to 1.75% of the entire search space. On a Pentium II at 233 MHz, it took 16 minutes on average to execute all 400 iterations. This compilation time is very tolerable for embedded systems since we can afford several hours to highly optimize compute intensive routines.

The most important factor in the running time of iterative compilation is the execution time of the transformed program. This is what we expected. However, Figure 9 shows that within a few minutes the search can be completed, if the execution time of the benchmark

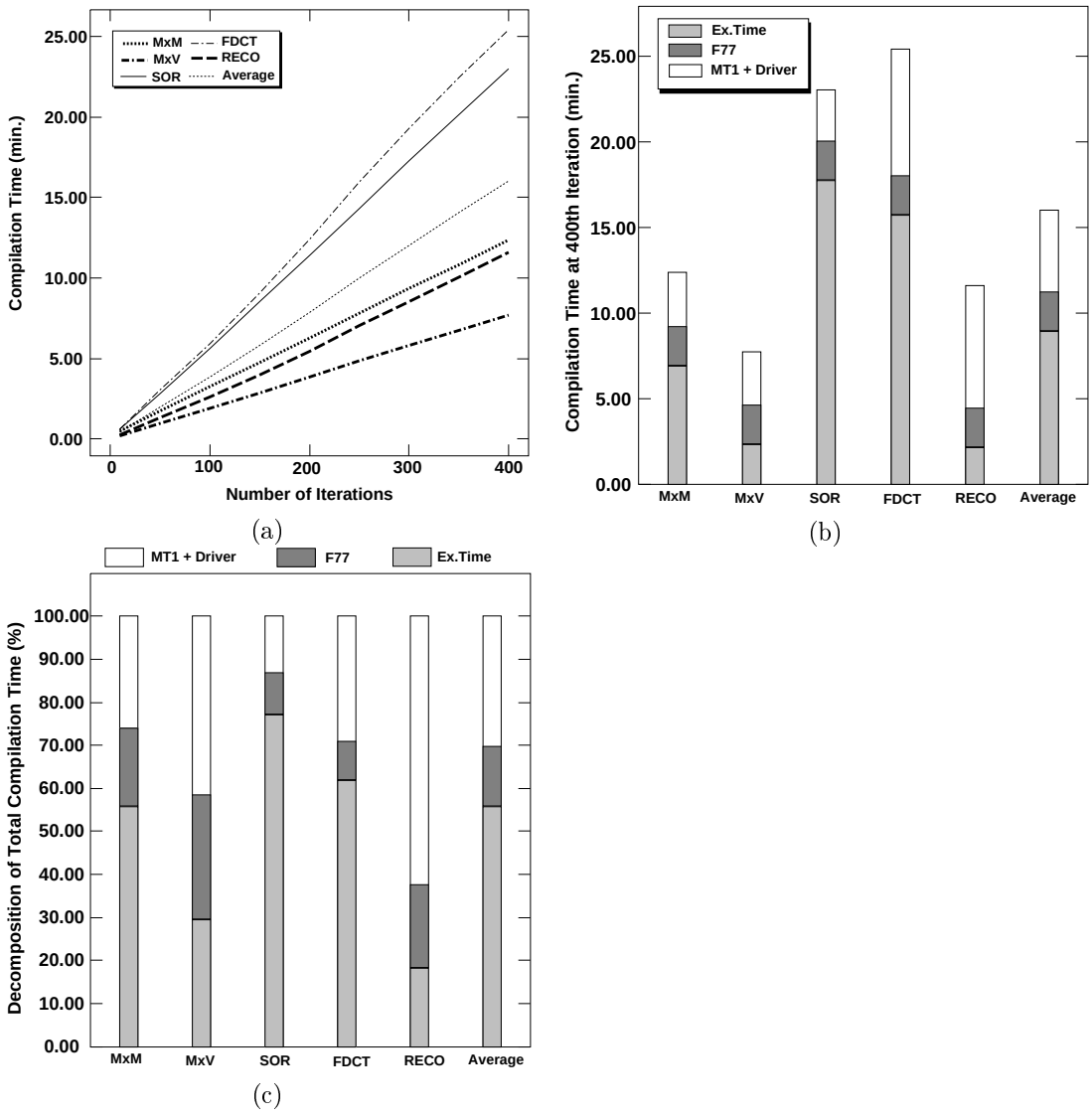

Figure 9. Compilation Time

itself is small. This is good news for embedded application where the compute intensive kernels themselves are small and fast, but are called very many times. In these cases very many points in the search space can be visited within a few hours, which is an acceptable time span to highly optimize embedded kernels.

In many other cases, however, the initial running time of the routine is much larger. Moreover, we need to consider more transformations and to apply this approach to entire applications. In such cases, search spaces are huge. Therefore, we need aggressive pruning and other improved search methods in order to bring down the number of iterations to a reasonable amount.

In this paper we have used a very simple search algorithm as a basis for iterative compilation. There is in fact a large body of literature on non-linear optimization [11], though it is based on a continuous underlying optimization function rather than the discrete space we consider. Techniques such as polynomial fitting could be applied to help improve the performance of the search algorithm. We will also study the applicability of other techniques from mathematical optimization theory, like simulated annealing, in the present setting.

There exists a large body of literature on static selection of transformations and static performance prediction. We intend to incorporate these models in the global driver to in order to reduce the search space size. The static models can provide initial seed points in the search space from which to begin a search, or equally importantly, eliminate from consideration those regions which we can statically predict will have poor performance.

One further approach to reducing compilation time is to optimize the code using small data sizes. The results reported in Section 4.3 suggest that optimizations found this way may well be suited for larger data sizes. However, these results also suggest that the optimizations found perform less than when optimizations are obtained using actual data sizes. In future work we will investigate the correlation between the data size used to optimize, the data size used in reality and the cache size. We will also investigate how to trade off compilation time and optimization levels using this approach.

References

[1] D.F. Bacan, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, Computer Science Division, UCB, 1993.

[2] R.A.M. Bakker, F. Breg, P.M.W. Knijnenburg, P. Touber, and H.A.G. Wijshoff. Strategy Specification Language. Oceans Deliverable D2.1.a, 1997. Available through `www.wi.leidenuniv.nl/~peterk`.

[3] M. Barreteau, F. Bodin, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, T. Kisuki, P.M.W. Knijnenburg, P. van der Mark, A. Nisbet, M.F.P. O'Boyle, E. Rohou, A. Seznec, E.A. Stöhr, M. Treffers, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In P. Amestoy *et al.*, editor, *Proc. Euro-Par 99*, volume 1685 of *Lecture Notes in Computer Science*, pages 1171–1175, 1999.

[4] A.J.C. Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, P. Touber, and H.A.G. Wijshoff. Transformation Definition Language. Oceans Deliverable D1.1, 1997. Available through `www.wi.leidenuniv.nl/~peterk`.

[5] A.J.C. Bik and H.A.G. Wijshoff. MT1: A prototype restructuring compiler. Technical Report no. 93-32, Department of Computer Science, Leiden University, 1993.

[6] 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.

[7] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Organised in conjuction with PACT'98.

[8] 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.

[9] 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.

[10] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. Programming Language Design and Implementation*, pages 279–290, 1995.

[11] E. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker Inc., New York, 1992.

[12] Wen-mei W. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1/2):229–248, May 1993.

[13] 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.

[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] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *iProc. MICRO 25*, 1992.

[16] 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.

[17] A. Nisbet. GAPS: Genetic algorithm optimised parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Workshop organised in conjunction with PACT'98.

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

[19] 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.

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

[21] 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.