Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation

T. Kisuki P.M.W. Knijnenburg

Leiden Institute of Advanced Computer Science, Leiden University,

Niels Bohrweg 1, 2333 CA Leiden, the Netherlands

Phone: +31 71 5277063, Fax: +31 71 5276985

{kisuki,peterk}@liacs.nl

M.F.P. O'Boyle

Institute for Computing Systems Architecture, Edinburgh University,

Mayfield Road, Edinburgh EH9 3JZ, UK,

Phone: +44 131 6505117, Fax: +44 131 6677209,

mob@dcs.ed.ac.uk

Keywords

Program Optimization, Adaptive Compilation, Program Transformation, Locality Optimization, Instruction Level Paralllelism

Abstract

Loop tiling and unrolling are two important program transformations to exploit locality and expose instruction level parallelism, respectively. However, these transformations are not independent and each can adversely affect the goal of the other. Furthermore, the best combination will vary dramatically from one processor to the next. In this paper, we therefore 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 based on genetic algorithms, random sampling and simulated annealing. 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. In this way, we can reach high levels of optimization within 50 iterations.

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 [16, 12, 18, 23, 30]. 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 utilise the internal parallelism within modern processors which are capable of issuing several instructions per cycle [13]. Loop unrolling is an important transformation for this purpose [27] 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. To illustrate the transformation we consider in this paper, consider the example given in Figure 1. In this figure, `TJ` and `TK` are the tile sizes for the `J` and `K` loop, respectively, and `U` is the unroll factor of the `I` loop.[1]

In this paper we study the combination of these two transformations and 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 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.

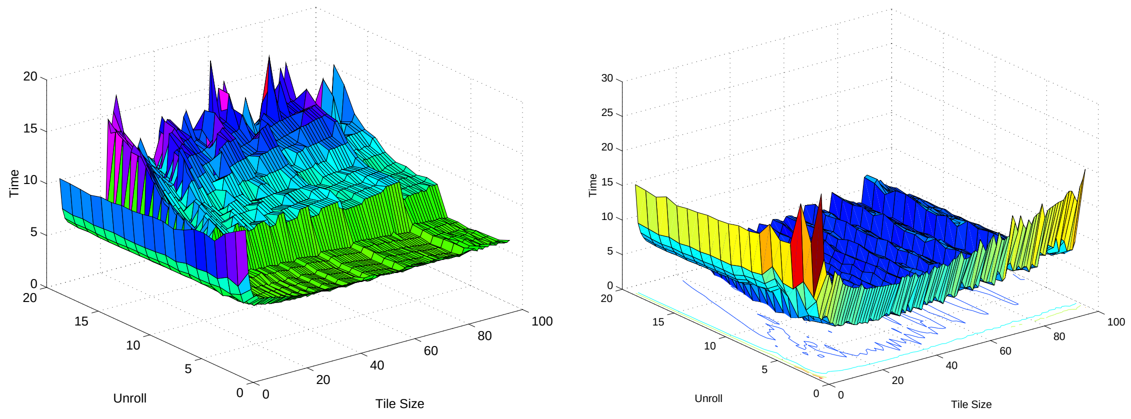
The close interaction between tiling and unrolling can be seen in Figure 2, which shows that a small deviation from 'good' tile sizes and unroll factors can cause a huge increase in execution time and even a slow down with respect to the original program. We need to answer the question for which tile sizes and unroll factors we obtain the minimum execution time. A static technique essentially tries to give an analytical expression for this minimum. In [7, 21] we have studied the characteristics of optimization spaces in detail for a variety of benchmarks and platforms and showed that different

[1]The variation on loop unrolling that we consider in this paper is unroll-and-jam [1, 9, 8] whereby an outer loop is unrolled and the inner loops are fused. Epilogue code is not shown here for simplicity.

Original	Transformed

`DO I = 1,N`
` DO J = 1,N`
`  DO K = 1,N`
`   A[I,J] = A[I,J] + B[I,K] * C[K,J]`
	`DO JJ = 1,N,TJ`
	` DO KK = 1,N,TK`
	`  DO I = 1,N,U`
	`   DO J = JJ,MIN(JJ+TJ-1,N)`
	`    DO K = KK,MIN(KK+TK-1,N)`
	`     A[I,J]     = A[I,J]     + B[I,K]     * C[K,J]`
	`     A[I+1,J]   = A[I+1,J]   + B[I+1,K]   * C[K,J]`
	`     ...`
	`     A[I+U-1,J] = A[I+U-1,J] + B[I+U-1,K] * C[K,J]`

Figure 1: Example Transformation



Pentium II UltraSparc

Figure 2: Execution Time MxM for Unrolling and Tiling

platforms give rise to widely varying optimization spaces (see Figure 2) 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. 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 the compilation time can be amortised 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.

Although, theoretically, iterative compilation can find the optimal version of a program for any architecture by simply considering all possibilities, in practice the search space is extremely large and therefore in this paper we examine techniques that reduce this cost and show that iterative compilation outperforms static techniques across a range of architectures.

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 and can find good solutions in less than 6.5 minutes on average. After analysing 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, future directions in section 9 and draw some concluding remarks in section 10.

2 Implementation of Iterative Compilation System

In this section we briefly discuss how the iterative compilation system is implemented.

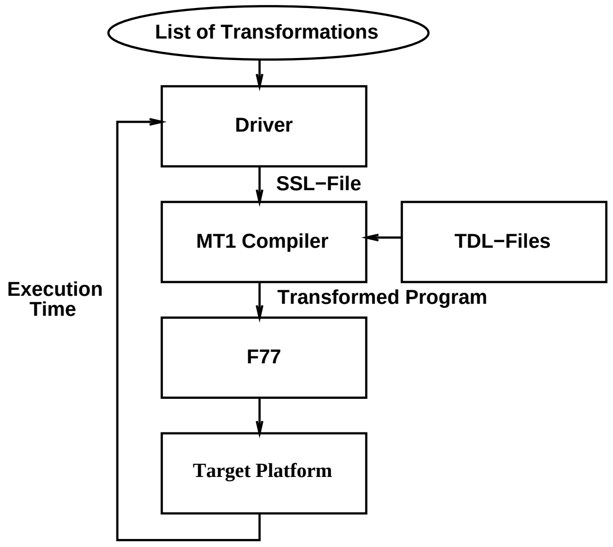


Figure 3: The Compilation Process

Figure 3 shows an overview of the compiler system. For more details, consult [22]. The compilation system is centred 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 [5] and instructs it which transformation to apply. MT1 has two mechanisms to control the application of transformations: a Transformation Definition Language (TDL) [4] and a Strategy Specification Language (SSL) [2]. 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.

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

1. Decide the next set of parameters for the transformations using 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.

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

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 several search algorithms, including a genetic algorithm, simulated annealing, pyramid search, window search and random search.

Genetic algorithm Genetic Algorithms are modelled on natural evolution processes and manipulate individuals in a population over several generations to improve their fitness. First, an initial population of S programs is randomly selected. Second, in the crossover phase, for a number of individuals a crossover point is determined in the bit representation of their 'chromosomes' that encode tile size and unroll factor. Different parts of the upperhalf and lowerhalf of these chromosomes are concatenated and thus new individuals are created. Third, in the mutation phase, bits are flipped in the chromosomes based on the mutation probability. Finally, the entire new population is evaluated and the execution time is used to establish the fitness of the individuals. If the fitness is too low, the individual is deleted from the population until a new population of 20 individuals is reached.

Simulated annealing SA is modelled on the physical process of heating up a solid and then cooling it down slowly until it crystallizes. Initially, a random point is selected and neigboring 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 2000 sets of parameters.

The GA, SA and Window algorithms contain parameters: the size of the initial population S, the cross-over rate c and the mutation rate m in GA. In SA we need to define the initial temperature T_0. In Window search we need to define the shrink factor p and number of samples per window s. We conducted a number of experiments with different values for these parameters on a limited set of benchmarks to establish which parameters perform best [20]. We found that for GA, a low cross-over rate performs best. We selected the following values for the parameters in GA: $S = 20$, $c = 0.4$ and $m = 0.01$. Likewise, we found that good values for the parameters in Window search are $s = 75$ and $p = 75$. Therefore, the experiments described below were conducted using these values. We used for the initial temperature in SA the value of $T_0 = 36700$.

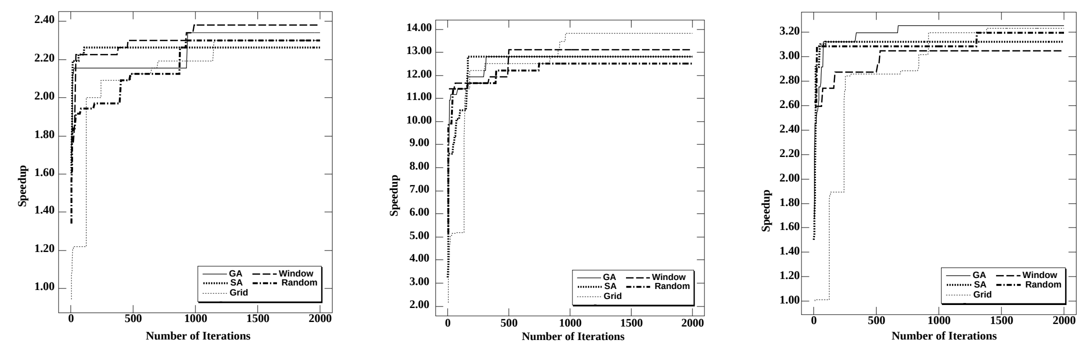
3.2 Benchmarks

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 behaviour. 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 behaviour. 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 innerproducts and the other loop is a matrix-matrix multiplication. These loops are hand optimised 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 $N = 301$.

3.3 Platforms

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



Pentium II HP-PA UltraSparc

Figure 4: Speedup MxM – ijk 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 limitation, we present only part of the speedup graphs that we have measured. For an exhaustive presentation of all these results the reader is referred to the full version of this paper [20]. We consider both rectangular and square tiles and show that square tiles achieve as much improvement as rectangular tiles in only a fraction of the compilation time. In section 5 we discuss how good these levels of optimization are by comparing them to existing static approaches.

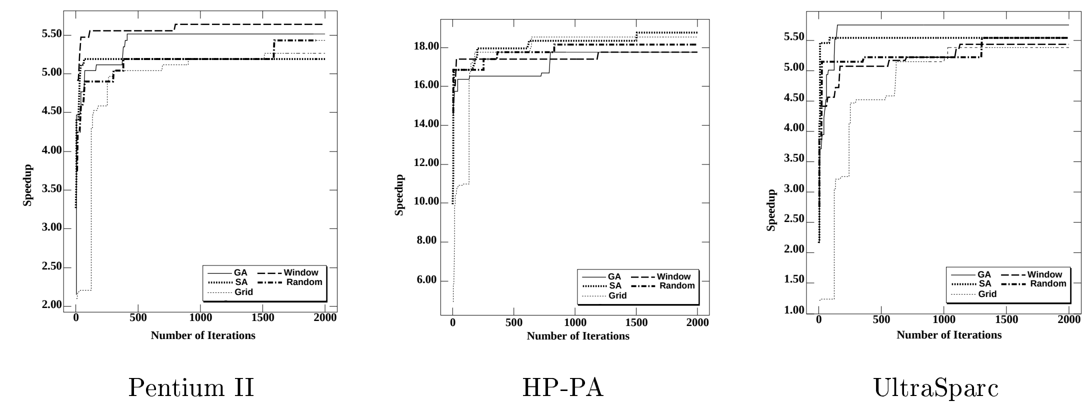


Figure 5: Speedup MxM – ikj version

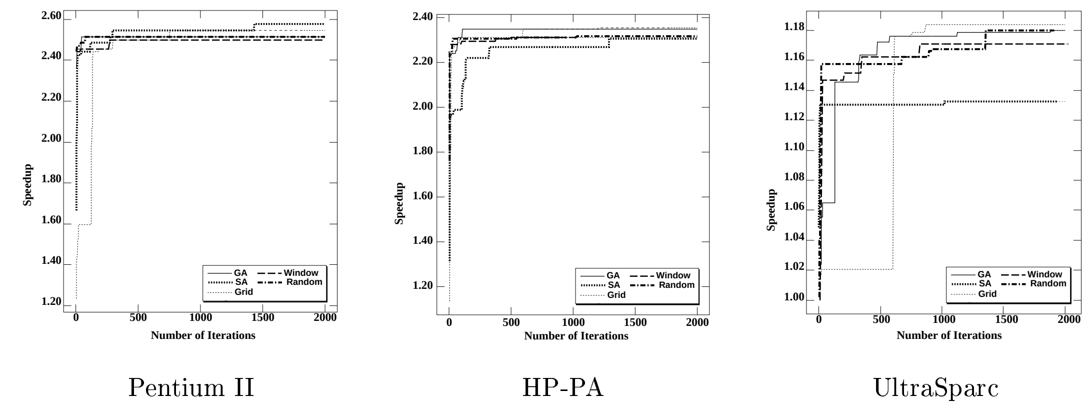


Figure 6: Speedup FDCT

4.1 Rectangular Tile Sizes

In this section we discuss the results when we search for 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. We did not consider MxV in this experiment since it only contains a double loop and rectangular tiling is not applicable.

4.1.1 Analysis of search algorithm performance

The results of iterative compilation are given in Figures 4 through 6. 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. For example, we obtain a speedup of over 18 for MxM (IKJ version) and a speedup of 30 for MxM (KIJ version) on the HP-PA. For these benchmarks, we obtain speedups of over 7, respectively 4, on the Pentium, and 5.5, respectively 3.3, on the UltraSparc. We have shown in [20] that these high levels of optimization are found across all our benchmarks and platforms for all data input sizes. This gives some confidence in the viability of our approach. In section 5 we give a more quantified assessment of this by comparing iterative compilation to two well-known static techniques.

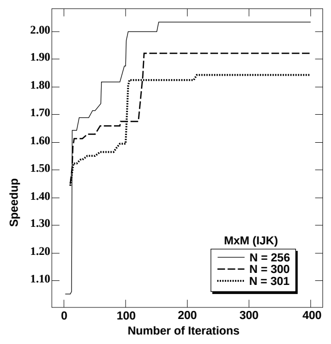
The second observation we can make is that these search algorithms do not differ much in their efficiency. The speedups found by the different search algorithms are within 5% on average of each other [20]. In the table below, we have given the number of iterations required to obtain maximal speedup. We also gave the number of iterations needed to reach 90% of this maximum.

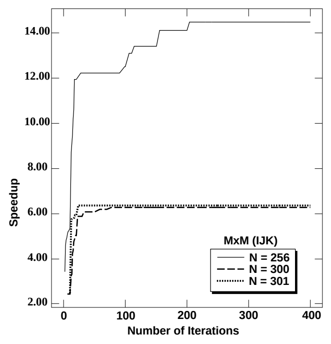
	max.	90% of max.
GA	808.4 its.	76.5 its.
SA	767.6 its.	144.6 its.
Pyramid	1043.9 its.	251.7 its.
Window	835.6 its.	65.6 its.
Random	747.0 its.	162.2 its.

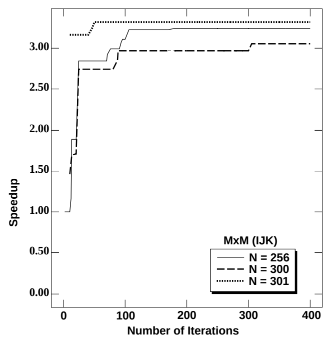
We observe that we on average need roughly between 750 and 1000 iterations in order to obtain the maximum speedup. We also observe, however, that iterative compilation reaches high levels of optimization much earlier and that the last few hundred iterations are used for a small increase in the final outcome. From the table we also observe that SA and Random reach their maximum fastest. It should be noted that SA shows quite random behaviour also, especially in its earlier phases where the probability to accept degradations is high. Also, Window search exhibits random behaviour because we draw random samples from the window that initially covers the entire search space. In our opinion this suggests that the underlying search space indeed is quite irregular and regular searching approaches will not yield acceptable results. Pyramid search is slowest because initially we define a grid over the entire search space that consists of 500 points. Many of these point may be located in regions that perform badly. Therefore, Pyramid search has a high initial overhead. This is also reflected in the high number of iterations needed to reach 90% of the maximal speedup. In general, we see that we need only a fraction of the number of iterations for maximal speedup to reach 90% of this maximum. Although the average number of iterations we need to reach this 90% lies between 0.03% and 0.13% of the entire search space, the absolute number is far too high to settle for it. However, this observation enables us to propose a trade-off between the number of evaluations we employ and the level of optimization that we can find. This topic is discussed in more detail in section 7 where we show a quantitative trade-off graph.

4.1.2 Conclusion

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. Several natural algorithms all perform almost equally well. We reach high levels of optimization visiting between 0.375% and 0.5% of the entire search space. We can reach 90% of maximum optimization by visiting a far smaller fraction of the search space: between 0.03% and 0.13%. However, searching for rectangular tile sizes requires a large 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.

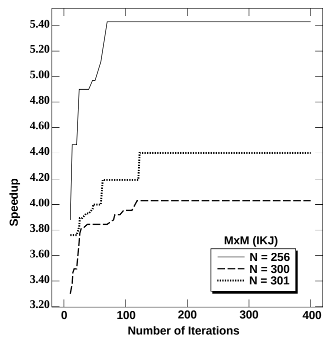


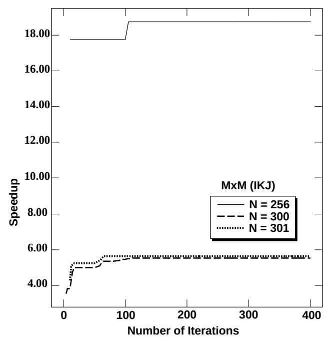


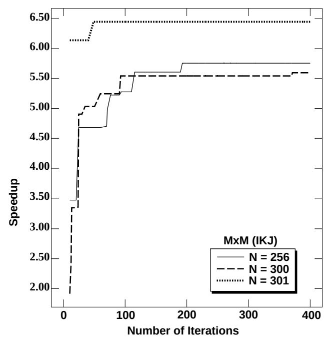


Pentium II HP-PA UltraSparc

Figure 7: Speedup MxM – ijk version





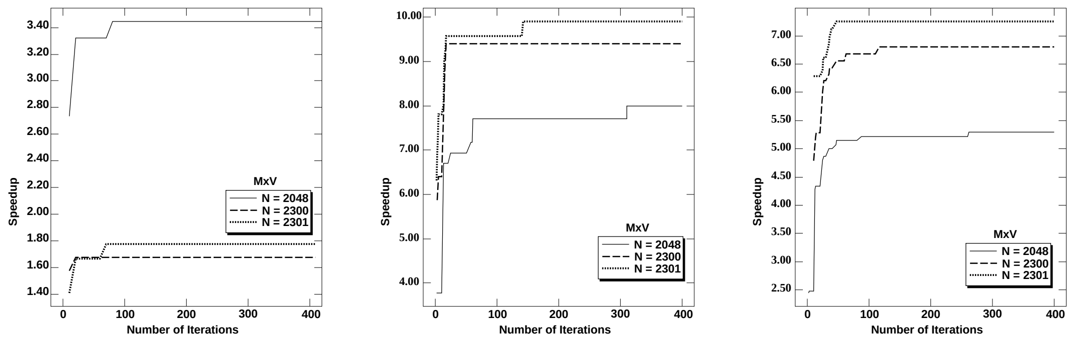


Pentium II HP-PA UltraSparc

Figure 8: Speedup MxM – ikj version

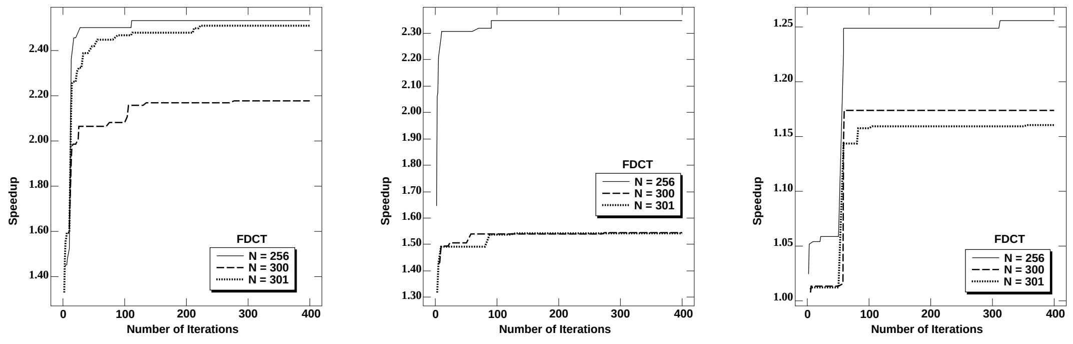
4.2 Square Tile Sizes

In this section we search for square tile sizes, reducing the size of the search space by a factor 100 to 2000 points. We implemented four search algorithms: GA, Pyramid, Random and SA. We compared the search algorithms and found that they reached their maximum improvement in about the same number of steps [20]. In this section we only present the results using Pyramid search. The results are given in Figures 7 through 10. We used 400 iterations to determine the maximal speedup, which is 20% of the number of iterations we used for rectangular tiles.



Pentium II HP-PA UltraSparc

Figure 9: Speedup MxV



Pentium II HP-PA UltraSparc

Figure 10: Speedup FDCT

4.2.1 Analysis of results

Comparing the speedups obtained using square tiles with the speedups obtained using rectangular tiles shows that they are about the same. In one case (MxM-KIJ on HP-PA) square tiling performs about 30% slower than rectangular tiling. In one other case (MxM-IKJ for $N = 300$ on Pentium) square tiling obtains a speedup of 4.03 whereas for rectangular tiling, speedups of 7.22 are obtained by 4 out of 5 search algorithms (the other algorithm also yielded a speedup of 4). However, in some other cases square tiling outperforms rectangular tiling: for the first loop in FDCT square tiling performs 45% better than rectangular tiling on the UltraSparc. In all other cases, square tiling is within 5% on average from rectangular tiling. In general, the difference between the different search algorithms for rectangular tiling is of the same order of magnitude than the difference between square and rectangular tiling [20]. This shows that 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. In the table below we have shown the average number of iterations needed to find the maximal speedup and 90% of this maximal speedup, for each platform. We see that we improve by a factor of 8 over searching for rectangular tile sizes, both for finding the maximal improvement and for finding 90% of this maximal improvement.

	max.	90% of max.
Pentium	146 its.	31.3 its.
HP-PA	79.9 its.	23.9 its.
Ultra	127 its.	38.2 its.
Average	116.2 its.	30.9 its.

4.2.2 Conclusion

We conclude that the speedup obtained using square tiles is almost as good as the one obtained using rectangular tiles. However, the time needed for iterative compilation is a factor of 8 smaller for square tiles than for rectangular tiles. This provides our first heuristic for managing the complexity of iterative compilation by only considering square tiles.

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. In [30] it has been shown that these tiling algorithms perform as good as or better than a host of other tiling techniques. The first algorithm, TSS by Coleman and McKinley [12], 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 minimise 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 [23], 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.

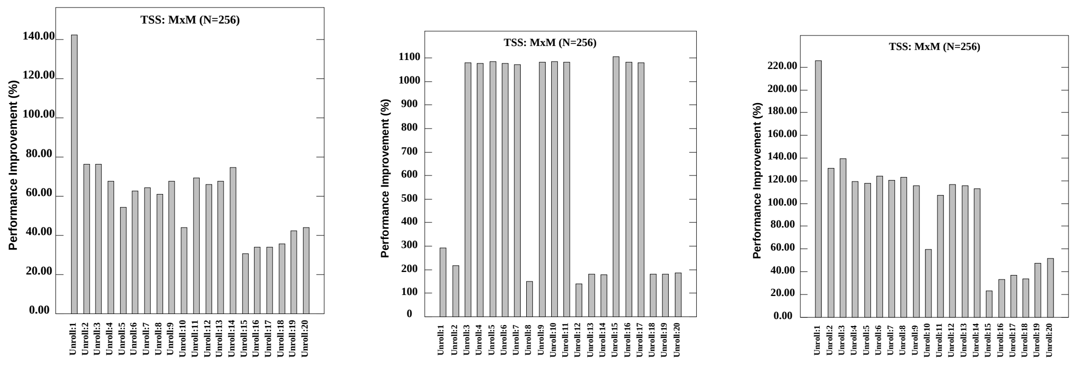
We compute the comparison between our approach and the static approaches as follows. Let S_{it} be the speedup obtained by iterative compilation and let S_{TSS} be the speedup obtained by TSS. Then the improvement of iterative compilation over TSS, I_{TSS}, is given by

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

We compute the improvement of iterative compilation over LRW likewise. Note that when iterative compilation produces a lower speedup than TSS, a negative improvement is obtained. The results are given in Figures 11 through 17 (see also [20]).

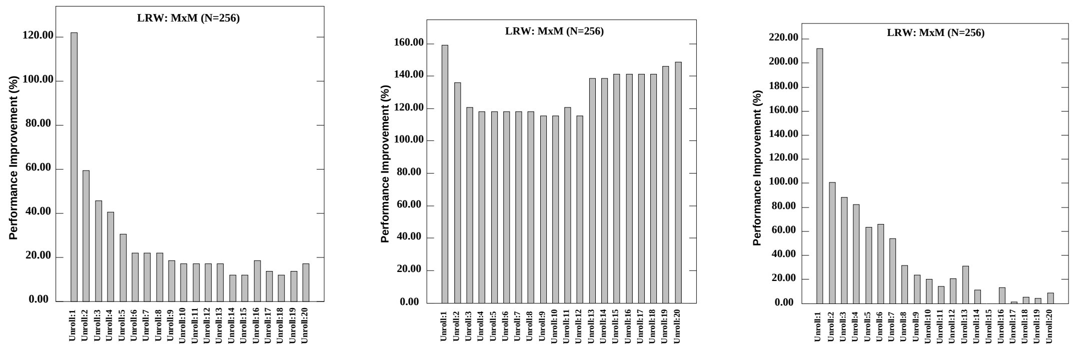
We immediately observe from these figures that iterative compilation outperforms the other techniques significantly, up to 1800% for MxM-IKJ on the HP-PA. Note that we show that this improvement holds for each unroll factor less than or equal to 20 that the compiler might choose. From the figures it can also be observed that for an unroll factor of 1, which corresponds to no unrolling, improvements over tiling only are large.

In the full version of the paper [20] it is shown that the observations given above hold for almost



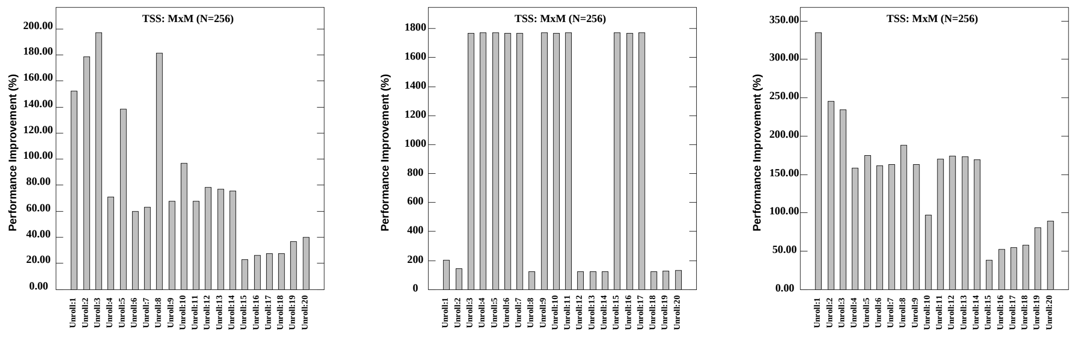
Pentium II HP-PA UltraSparc

Figure 11: Improvement over TSS for MxM – ijk version, $N = 256$



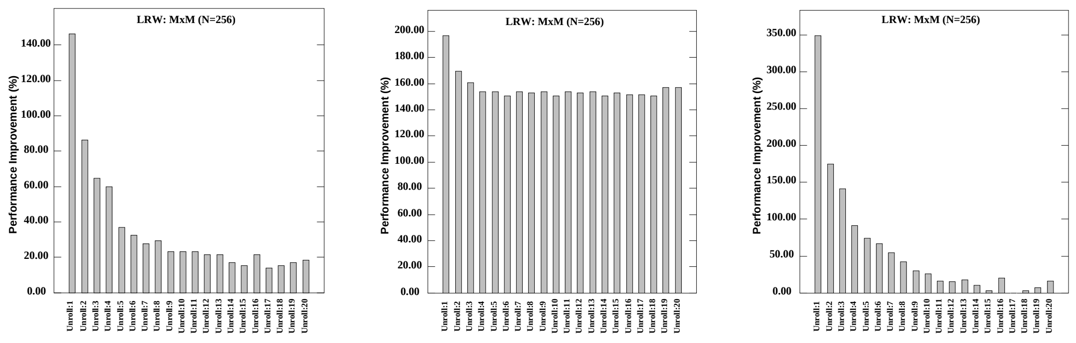
Pentium II HP-PA UltraSparc

Figure 12: Improvement over LRW for MxM – ijk version, $N = 256$



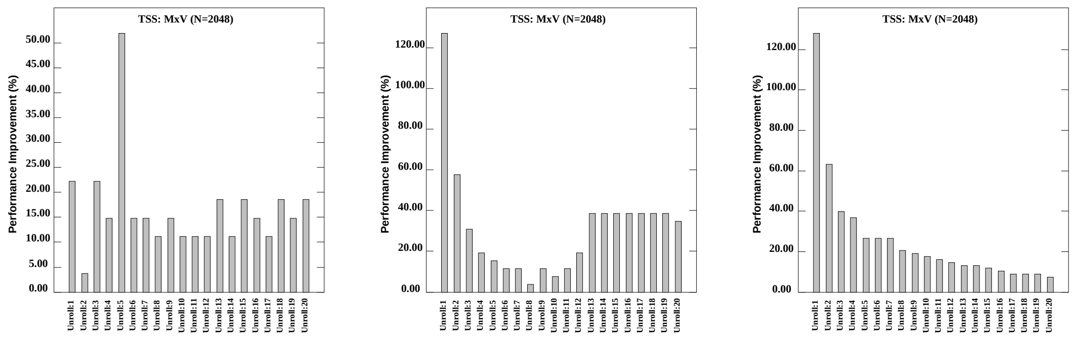
Pentium II HP-PA UltraSparc

Figure 13: Improvement over TSS for MxM – ikj version, $N = 256$



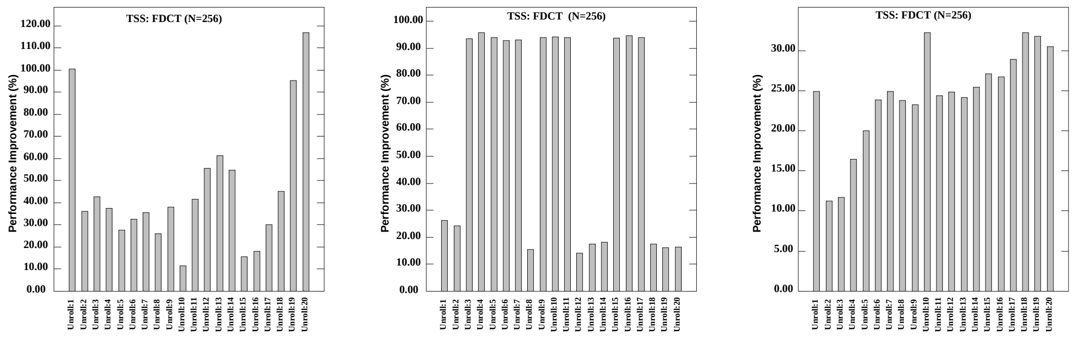
Pentium II HP-PA UltraSparc

Figure 14: Improvement over LRW for MxM – ikj version, $N = 256$



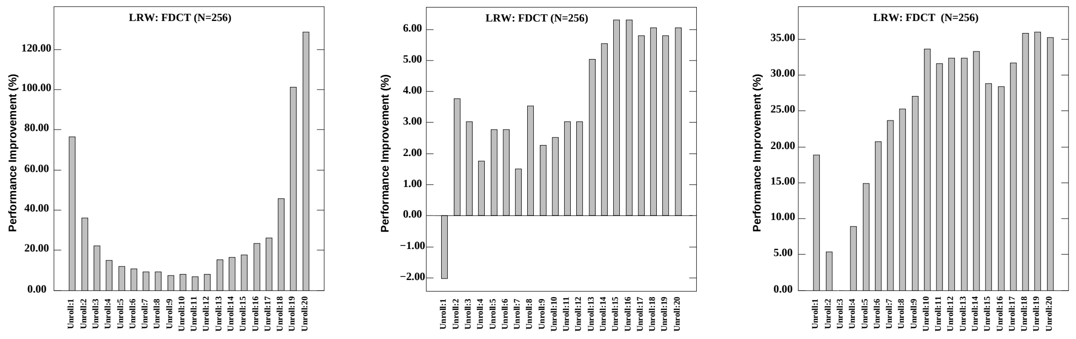
Pentium II HP-PA UltraSparc

Figure 15: Improvement over TSS for MxV, $N = 2048$



Pentium II HP-PA UltraSparc

Figure 16: Improvement over TSS for FDCT, $N = 256$



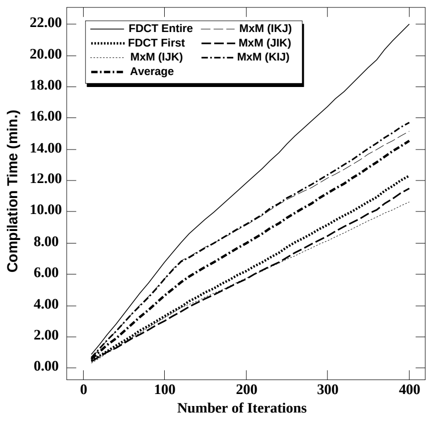
Pentium II HP-PA UltraSparc

Figure 17: Improvement over LRW for FDCT, $N = 256$

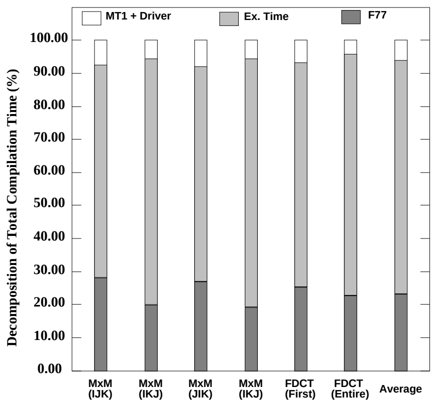
each benchmark, data size and platform. In particular, iterative compilation improves in all cases on Pentium. For UltraSparc, there is 1 case out of 600 where iterative compilation performs 2% less than TSS and it always performs better than LRW. For HP-PA, there are 20 out of 600 cases where iterative compilation performs 2% less than TSS and 2 cases where iterative compilation performs 3% less than LRW. On examining the code, this slight degradation is due to statistical noise in measuring the execution time. This means that speedups found by iterative compilation are the same in these cases as those found by static means. Hence, summing up, in 99.4% of our benchmarks iterative compilation outperforms a static tile size selection algorithm. In the other cases only a very slight degradation of about 2% can be observed that is largely due to noise. From this data we conclude that iterative compilation is a powerful optimization technique outperforming existing static techniques significantly.

6 Compilation Time

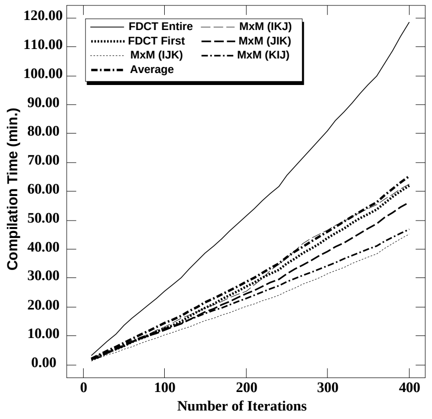
In this section we discuss the compilation time required for iterative compilation that we show in Figure 18 as a function of the number of iterations, together with the breakdown in the times required for searching and program transformation in MT1, native Fortran77 compilation and execution. We observe that the 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.



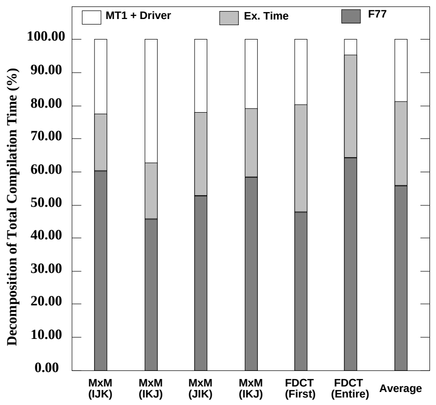
Pentium: Total time



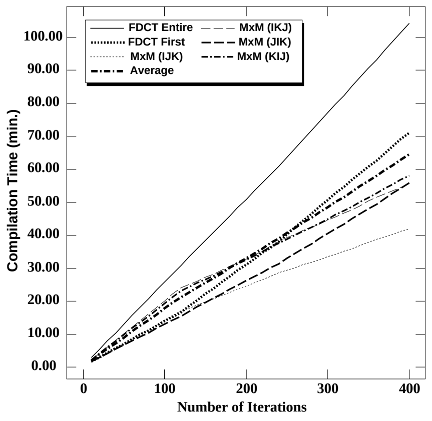
Pentium: Decomposition



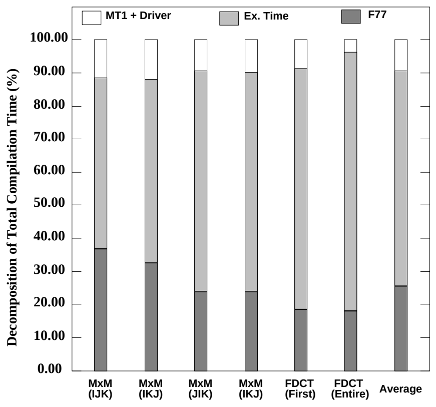
HP: Total time



HP: Decomposition



Ultra: Total time



Ultra: Decomposition

Figure 18: Compilation Time and Decomposition of Compilation Time

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 Number of Iterations and Level 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

We need to define a metric that we can use to compare improvements. In this subsection we discuss two possibilities. We use both metrics below to construct a trade-off graph between the number of iterations and the levels of optimization that result. First, we want to quantify the effect of a transformation. There are two natural approaches.

Speedup Suppose the execution time of the original program is t_0 and the execution time of the transformed program is t_1. The *speedup* of the transformation S is defined as

$$S = \frac{t_0}{t_1}$$

As we keep track of the best version found so far, the speedup is always greater than or equal to one, $S \geq 1$.

Execution time improvement This quantity is defined as the difference between the original execution time and the execution time of the transformed program, relative to the execution time of the original execution time. The improvement I of the transformation is therefore defined as

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

If the transformation slows down the original program, $I < 0$. Otherwise, $0 \leq I < 100\%$. If $I = 0\%$, then the transformation has no effect. If t_1 decreases to 0 seconds, I increases to 100%. In the present case of iterative compilation, I lies between 0% and 100%.

Using the two measures above, we can define a metric to quantify how close a transformation comes to another transformation. 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.

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

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

where S_i is the speedup obtained from transformation T_i. It is easy to see that M_S has the two properties discussed above.

Execution time improvement We want the metric M_I to measure how close the improvement I_1 comes to I_2. That is, we want the relation $I_1 = M_I(T_1, T_2) \cdot I_2$ to hold. Therefore, we define

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

where t_0 is the original execution time and t_i is the execution time resulting from transformation T_i. Again it is easy to see that M_I has the two properties discussed above. We can rewrite M_I in terms of speedup as follows.

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

Below, T_2 is the transformation with maximal improvement, hence $S_2 \geq S_1$ and $M_I \geq M_S$.

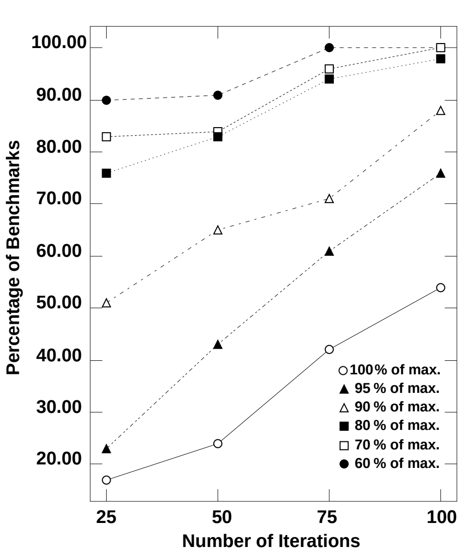
For example, suppose we have a program that runs in 20 seconds. Suppose transformation T_1 reduces the running time to 4 seconds, and transformation T_2 reduces the running time to 2 seconds. Then $S_1 = 5$ and $S_2 = 10$, and $I_1 = 80\%$ and $I_2 = 90\%$. To compute comparisons, $M_S(T_1, T_2) = \frac{4}{9} \cdot 100\% \approx 44\%$ 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. There is a large difference between speedups and therefore M_S is only 44%. On the other hand, there is not so much difference in running times compared to the original running time. T_1 already reduces the running time to a large extend. The metric M_I records this and says that T_1 reaches 88% of T_2.

7.2 Quantitative Trade-off Graphs

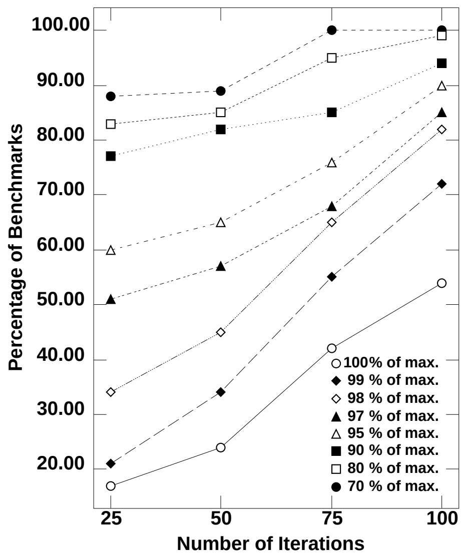
We now want to compare the transformation T_i found in iteration i to the final transformation T_m with maximal effect that has been found after 400 iterations. We use the results of both the Pyramid and the Random search algorithm since this last algorithm inspects the entire search space in a few iterations and hence delivers high levels of optimization rapidly. We use both metrics discussed above in this comparison. In this way, we are able to focus on both speedup and execution time improvement. We proceed as follows.

We base our comparison on the transformations using square tiles discussed in section 4.2. We measured the speedup found after 25 iterations in all 82 experiments that we have conducted. We counted the number of cases where we reached 100% of the speedup. This yields a percentage of the experiments that reach this maximal result. Likewise, we counted the number of cases where we reached at least 95%, 90%, 80% and 70% of the maximal speedup. We also counted the cases where we reached at least 100%, 99%, 98%, 97%, 95%, 90% and 80% of the maximal execution time improvement.

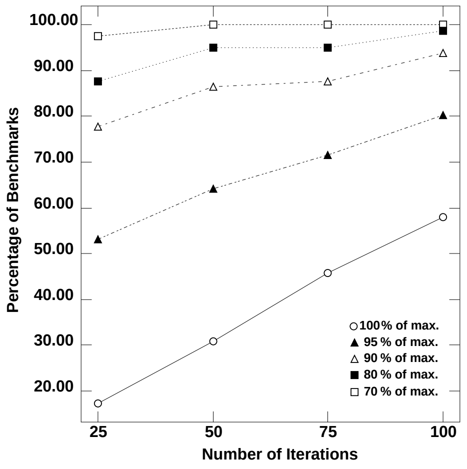
We followed the same procedure after 50, 75 and 100 iterations. The results are plotted in Figure 19 that provide quantitative trade-off graphs. From these figures we can deduce, for example, that using the Random algorithm after 100 iteration, 58% of the benchmarks were fully optimized and thus reached 100% of the speedup or execution time improvement. Likewise, we see that after 50



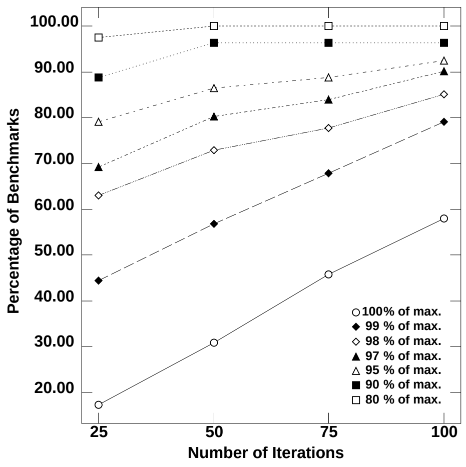
(a) Pyramid: Trade-off Speedup



(b) Pyramid: Trade-off Time Improvement



(a) Random: Trade-off Speedup



(b) Random: Trade-off Time Improvement

Figure 19: Trade-off Graphs for Pyramid search (top) and Random search (bottom)

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 if N iterations, we can deduce the effect of iterative compilation in terms of how 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 "equibars" for 70% and 80% of the speedup, and the "equibars" 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.

In Table 1, we have shown the compilation time in minutes required for 25, 50, 75 and 100 iterations. We observe that the times for 25 or 50 iterations can be afforded. In future work we hope to bring down these compilation times even further by including static models in the search.

Compilation time	25 its.	50 its.	75 its.	100 its.
Pentium	1.23 m.	2.37 m.	3.27 m.	4.63 m.
HP-PA	4.10 m.	7.68 m.	10.4 m.	14.34 m.
UltraSparc	4.66 m.	9.02 m.	12.51 m.	17.77 m.

Table 1: Compilation Time for Small Numbers of Iterations

8 Related Work

There are many paper dealing with tile size selection [12, 16, 23, 27, 30]. All these selection algorithms use static analysis and models to compute tile sizes, in contrast to the present approach that uses dynamic profiling information. Carr and Kennedy [9] and Carr [8] compute unroll-and-jam factors in order to minimise the difference in machine and loop balance. Carr computes how much benefit the unroll-and-jam of a loop has for a range of unroll factors based on static models and searches at compile time to decide which unroll factor has the most benefit. In contrast, the present approach uses actual execution times and moreover considers loop tiling at the same time.

Currently, searching techniques are employed in hardware generation, for example, in design space exploration [14]. In this approach, many implementations of a design are generated and static models are used to estimate for example die size and speed of the circuit. Optimal points in the design space are called Pareto points. For example, one such point signifies that some implementation gives the fastest circuit for a given die size, or alternatively the smallest die for a given speed.

Whaley and Dongarra [32], and Bilmes et al. [6] 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. In contrast to the present approach, however, these systems are only able to optimise BLAS routines and are not general purpose compilers.

Wolf, May Dan and Chen [33] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. Han, Rivera and Tseng [18] 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. Our approach based on actual execution times will deliver superior performance and can adapt to any architecture, requiring no prior modelling phase.

Chow and Wu [10] 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 [19] or hyperblocks [24] to enable efficient scheduling for ILP processors. These techniques are currently being employed in commercial compilers [11]. Profiles are also used to identify runtime constants that can be exploited at compile time [26]. The recently established workshop on Feedback Directed Optimization shows that currently many proposals are being put forward to exploit profile information in the compiler chain [15]. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

The present research was started within the Esprit project OCEANS [3]. Within this project, other approaches to iterative compilation are considered. Bodin explores in [31] the interplay between loop unrolling and software pipelining. This approach can be fully integrated with the present approach since Bodin targets a different phase in the compiler, namely, the code generation phase. In [28], Nisbett proposes a genetic algorithm approach to searching.

9 Future Directions

The obvious drawback of iterative compilation is its long compilation time that is required in order to generate many versions of the source program and execute them to obtain profiling information. In this section we discuss a few possible solutions.

The first approach is to use analytical models in the search. We intend to use the static models to guide the search by using them to decide whether one version of the program is better than another. Only if the models predict that some version might be better than the best one found so far, we will actually profile that version. In this way, profiling the transformed program will only be done in situations where static information is insufficient to give reliable predictions. These models should be accurate enough to cover a large portion of the search space or, alternatively, should be accurate for certain aspects of the search space. Currently, we are implementing a cache model to measure the impact of the transformations at compile time on memory access behaviour and a parameterised scheduler to measure the impact on ILP exploitation. We are also implementing Cache Miss Equations [17] and the procedure to statically evaluate the effect of Unroll-and-Jam proposed by Carr [8]. 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, the number of points that are inspected and the number of programs that are actually executed. In this compilation model, traditional compilers use models of low to medium complexity, visit one point and execute no programs. The present paper discusses the situation where there are no models and many points are visited and evaluated. The compilers by Wolf, Maydan and Cheng [33], and the one by Han, Rivera and Tseng [18] fall in between by using low complexity models, visit many points but do not execute any program.

In the present paper we discuss how to deal with parameters for a given transformation. However, many other transformations can be employed that do not have such a parameter, like loop interchange. In [25] we discuss an approach by considering decision trees for applying a host of transformations, including data transformations [29]. Loop unrolling and tiling is one node in this tree and the present approach to transformation space exploration can be used in this node to determine optimal parameters. However, we need to ensure that good tile sizes and unroll factors can be found very quickly for this approach to be feasible.

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. The compilation time required is then reduced to less than 6.5 minutes on average.

References

[1] F.E. Allen and J.Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

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

[9] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[10] 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.

[11] 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.

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

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

[14] G. de Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[15] B. Calder et al., editor. *Proc. Workshop on Feedback Directed Optimization*, 1999. Available through `http://www-cse.ucsd.edu/users/calder/fdo`.

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

[17] S. Gosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tunig memory behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703–746, 1999.

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

[19] 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.

[20] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Iterative compilation for tile sizes and unroll factors: Implementation, performance, search strategies. Technical Report TR2000-06, LIACS, Leiden University, 2000.

[21] 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.

[22] 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.

[23] 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.

[24] 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 *Proc. MICRO 25*, 1992.

[25] 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.

[26] 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.

[27] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

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

[29] M.F.P O'Boyle and P.M.W. Knijnenburg. Efficient parallelization using combined loop and data transformations. In *Proc. IEEE Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 283–291, 1999.

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

[31] 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.

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

[33] 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.