Incorporating Cache Models in Iterative Compilation for Combined Tiling and Unrolling

T. Kisuki* P.M.W. Knijnenburg* M.F.P. O'Boyle†

* LIACS, Leiden University, the Netherlands

{kisuki,peterk}@liacs.nl

† Institute for Computing Systems Architecture,
Edinburgh University, UK, mob@dcs.ed.ac.uk

Abstract

In this paper we further investigate the notion of iterative compilation, in which the problem of determining the optimal program transformation is approached by generating many versions of the source program and by searching for the best by actually executing these versions on the target hardware to measure their execution time. In previous work we have shown that this approach can obtain high levels of optimization, outperforming existing static techniques significantly. In this paper we address how to incorporate static models in the search procedure in order to reduce the number of program executions. We focus on cache models since exploitation of the memory hierarchy is very important in obtaining execution speed. First, we show that by using

these models alone and no profiling, far lower levels of optimization are obtained than by using profiling information. Second, we show that including accurate cache models can reduce the number of program executions by 50% and still obtain the same levels of optimization. We also show that less accurate models are capable of improving iterative compilation as much as a simulator, in case we have a limited number of profiles. Otherwise, these models may actually degrade the performance of iterative compilation.

1 Introduction

By the year 2010 it is predicted that there will be approximately 1 billion transistors available on a chip[1]. This represents an opportunity and challenge to computer architects to design and build processors capable of exploiting such a resource. Increasingly, processors rely on compiler technology to exploit the potential resources by carefully mapping applications to hardware [8]. However, the rate of architectural change is such that in the near future it will not be possible to produce high performance optimizing compilers in the time available and we therefore need to consider adaptive compilers, i.e., those that are able to cope with a changing hardware platform throughout their lifetime.

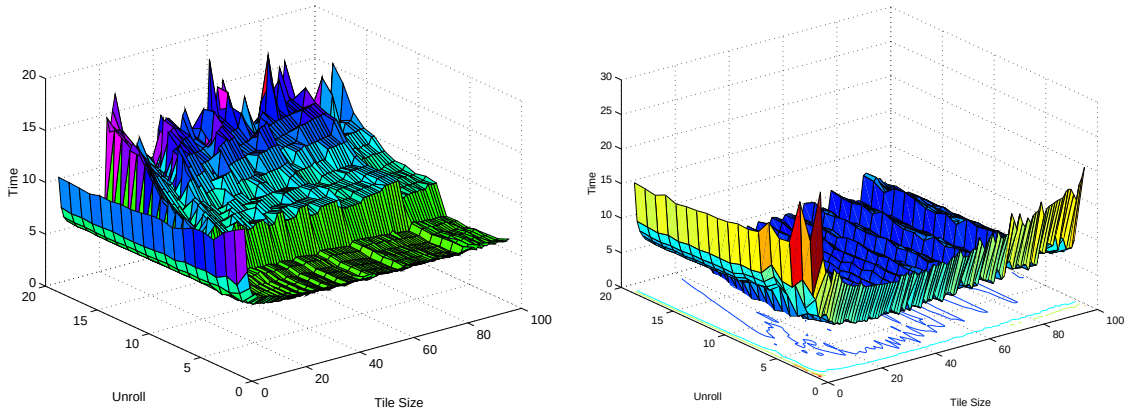
One of the main reasons that it will become increasingly difficult to develop compilers of a sufficient quality at a certain rate, is that the inherent problem of optimization is highly complex and varies considerably from one architecture to the next. Consider the frequently occurring problem of trying to exploit the memory hierarchy and internal parallelism of a processor by applying tiling and unrolling transformations, respectively, to the most visited

[1] The Semiconductor Industry Association's 1997 projection states that by 2010 a processor chip will contain around 800 million transistors and operate at over 2 GHz.

sections of code, namely loop nest. Finding the correct tile size and unroll factor is critical as can be seen from figure 1, that shows the execution time of matrix multiplication as a function of tile size and unroll factor for two distinct processors. A small deviation from 'good' tile sizes and unroll factors may cause a large increase in execution time and even a slow down with respect to the original program. Furthermore, the best tile size and unroll factor radically changes from one processor to the next as can also be seen in Figure 1. Thus, even if a compiler were able to determine the best optimization for a particular processor in this highly non-linear optimization space, it is extremely unlikely to be able to perform the same task on an entirely different processor. Yet this is precisely what we hope to achieve, namely, an optimizing compiler that has a longer lifetime than its hardware platform and is capable of adaptation.

Clearly no compiler technology solely based on static analysis and a hardwired cost model of the target processor is capable of changing to a new architecture. What is required is a method where the compiler can receive dynamic feedback regarding its performance and modify its behavior. The use of dynamic information for compiler decision making has been investigated for a number of years. Indeed, most popular production compilers are capable of making use of profile information [21]. All of these schemes however, choose between options determined statically beforehand, again relying on hardwired cost-models.

In this paper we investigate how compiler technology may adapt to architectural change by taking an extreme point of view, where the compiler has no knowledge of the underlying architecture and attempts to search for the best optimization using iterative compilation. Here, the compiler investigates the optimization space off-line, generating different version of the source program based on a generic search strategy and actual execution time feed-back.



Pentium II UltraSparc

Figure 1: Execution Time MxM for Unrolling and Tiling

Iterative compilation, based on the selection of high level transformations, has been shown to work across a range of architectures [13] and although preliminary work has shown this approach to be highly effective [14], the number of executions needed to find a good program may be prohibitively expensive. We therefore also consider how additional information may be used to guide the search strategy and its effect on both compiler efficiency and code performance.

In this paper we consider how a completely generic approach to adaptive compilation may be augmented with additional machine specific cost models to improve the running time of the compiler. Such an approach allows the iterative compiler to always produce good results regardless of the platform, but available static information can be used to improve efficiency. In order to focus the comparison, we will consider only a small transformation space, namely tiling and unrolling, across several benchmarks and platforms.

This paper is organized as follows. In Section 2 we discuss the implementation of the iterative compilation system and briefly review its performance. In Section 3 we discuss the cache

models used, the iterative search algorithms and the benchmarks and platforms. In Section 4 we discuss the performance of iterative compilation with cache models, compared to iterative compilation without cache models. In Section 5 we give a detailed analysis of the levels of optimization that can be reached when we limited the number of program executions. We show that cache models are capable of reducing this number of program executions by 50%. In section 6 we discuss the results obtained in this paper and some future directions in our research. In section 7 we discuss related work and we draw some concluding remarks in section 8.

2 Iterative Compilation

In this section we briefly discuss how the iterative compilation system is implemented and we briefly review its performance.

2.1 Implementation

Figure 2 shows an overview of the compiler system. For more details, consult [15]. The compilation system is centered around a global driver that reads a list of transformations that it needs to examine together with the range of their parameters. The driver keeps track of the different transformations evaluated so far and decides which transformations have to be applied next using a search algorithm to steer through the optimization space. We have implemented several search algorithms, including a Genetic Algorithm, Simulated Annealing, Pyramid Search, Window Search and Random Search [13]. In this paper we have included cache models in the driver that are used to decide whether or not to execute the

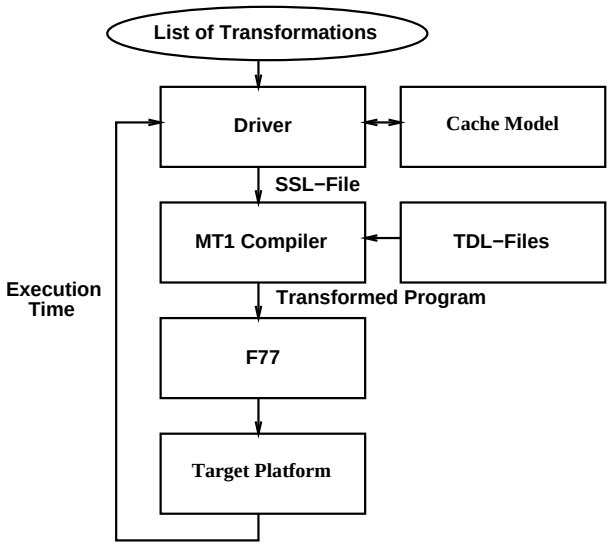
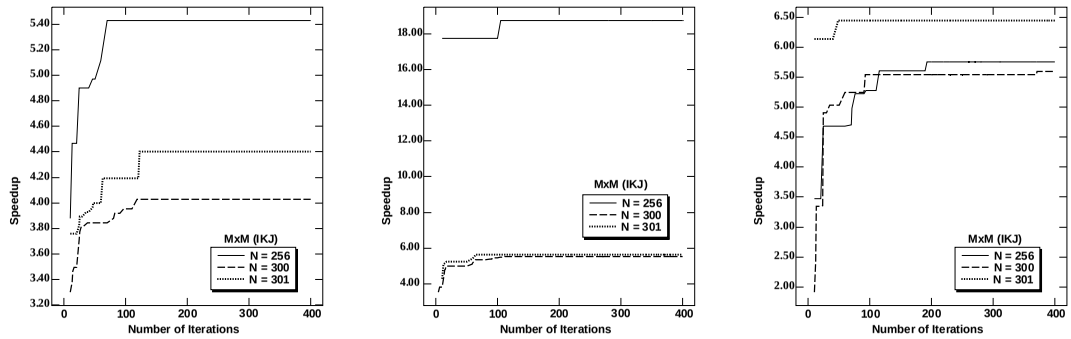


Figure 2: The Compilation Process

program, as explained in Section 3 below. The global driver invokes the source to source compiler MT1 [2] and instructs it which transformation to apply. MT1 has two mechanisms to control the application of transformations: a Transformation Definition Language (TDL) and a Strategy Specification Language (SSL) [1]. For each transformation included in the list of transformations, a transformation needs to be specified in the TDL-file. The global driver constructs an SSL file that specifies the order in which to apply certain transformations and outputs it to MT1. After a predetermined number of iterations, the global driver stops searching and outputs the transformed program with the shortest execution time.

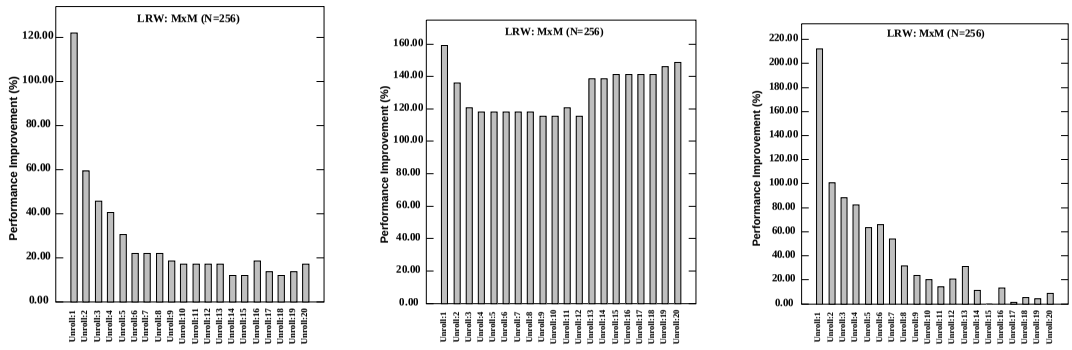
2.2 Performance of Iterative Compilation

In this section we briefly review how much speedup can be obtained by iterative compilation without using cache models, as discussed in [13]. In Figure 3 we show an example of the



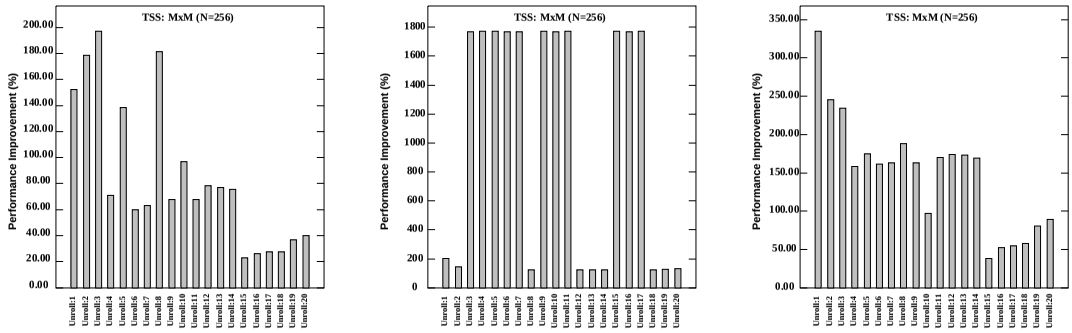
Pentium II HP-PA UltraSparc

(a) Speedup



Pentium II HP-PA UltraSparc

(b) Improvement over LRW



Pentium II HP-PA UltraSparc

(c) Improvement over TSS

Figure 3: Example Performance Iterative Compilation: MxM (IKJ version)

use of iterative compilation where we optimize Matrix-Matrix Multiplication (IKJ version) for different data input sizes. Figure 3(a) shows the speedup as a function of the number of iterations. We see that significant speedups are obtained. We quantify the quality of the transformation found by comparing this speedup to the speedups obtained from two well-known static tile size selection algorithms: TSS proposed by Coleman and McKinley [7] and LRW proposed by Lam, Rothberg and Wolf [16]. First, we unrolled the loop with unroll factors of 1 to 20 and subsequently computed the tile size for this unrolled loop. Improvements of iterative compilation over LRW and TSS are given in Figures 3(b) and 3(c), respectively. In [13] we show that iterative compilation outperforms both TSS and LRW in more than 99% of the cases we considered. The compilation time is linear in the number of program executions and in [13] we show that it takes 6 minutes on average for 50 executions. We conclude that iterative compilation is a powerful approach to program optimization, outperforming existing static techniques significantly.

3 Experiment

In this section we discuss the effect of adding cache models to the global driver. We discuss the search algorithms, two different cache models and the benchmarks and platforms used.

3.1 Search Algorithms

Given a cache model, we can use it in two ways to select a transformation, as shown in Figure 4. The first algorithm, shown in Figure 4(a), only uses the model. The second algorithm, shown in Figure 4(b), uses the model to exclude certain transformations from

current = *initial transformation*
REPEAT
  next = *next transformation*
  IF *model*(next) *better than model*(current)
  THEN current = next
	current = *initial transformation*
	REPEAT
	  next = *next transformation*
	  IF *model*(next) *better than* α% *model*(current)
	  THEN *execute*(next)
	  IF *exec_time*(next) < *exec_time*(current)
	  THEN current = next

(a) Model-Only	(b) Model+Execution

Figure 4: Search Algorithms

being executed. However, we still use execution times to select a new transformation. We have included a "slack factor" α since we expect that a model will only capture the behavior of the program partially. If a model is more accurate, we expect that we can use higher values for α and thereby exclude more transformations from actual execution. Below we show that the performance of iterative compilation is highly sensitive to α.

In [13] we have described several algorithms that can be used to determine the next transformation, including a Genetic Algorithm, Simulated Annealing, Pyramid Search and Random Search. We showed that these algorithms give rise to the same speedups within a margin of a few percentage. Also, their running times are in the same order of magnitude. We can explain this by observing that GA and SA are targeted towards huge search spaces requiring many samples. In the present context we have to deal with a small search space (albeit where sampling one point is an expensive operation) and we focus on the effect of taking fewer than 100 sample points. In this case, both GA and SA are still in their first phases and they exhibit quite random behavior. Therefore, we will use a random algorithm to determine the next transformation in the remainder of this paper.

Finally, we will compare the results obtained from iterative compilation including a cache model with results obtained from iterative compilation without a cache model as described

in [13]. The search algorithm described in [13] that only uses execution time is called the Execution-Only Algorithm below. This algorithm repeatedly picks a set of parameters, executes the corresponding transformed program and selects the set of parameters that gives rise to the shortest execution time.

3.2 Cache Models

We distinguish two extremes in the spectrum of possible models. First, as an *upperbound*, we use a full cache simulator to compute the exact hit rate. This model is highly accurate but in practice it is unlikely that it would be used, due to its high cost, but it provides a good upperbound for the available static analysis. Other sophisticated cache models also try to capture hit rates [22]. Second, as a *lowerbound*, we use a simple model proposed by Coleman and McKinley [7] that is inexpensive but less accurate than a simulator. It uses an approximation of the working set WS and the cross-interference rate CIR. Other cache models that might be used will fall in between these models in terms of accuracy and cost. Therefore, the results obtained in this paper can be used to give insight into the efficiency of such a model, by our analysis of the two extremes of the spectrum of possible models.

Using the full cache simulator, we say that a version of a program P_1 is better than another version P_2 iff the hit rate of P_1 computed by the simulator is larger than the hit rate of P_2. The model proposed by Coleman and McKinley [7] can be described as follows. For two versions P_1 and P_2 of a program, the model says that P_1 is better than P_2 iff

$$WS(P_1) > WS(P_2) \;\; \&\& \;\; WS(P_1) < CS \;\; \&\& \;\; CIR(P_1) < CIR(P_2)$$

where WS is the working set of one tile, CS is the cache size and CIR is the cross interference

rate of one tile.

We include a slack factor α in the models as follows. For the cache simulator, we use the following selection criterion in the search procedure:

$$\textbf{CS}: \quad \textit{hit rate}(P_1) > \alpha\% \ \textit{hit rate}(P_2)$$

For the Coleman/McKinley model, we have the following selection criterion:

$$\textbf{CM}: \quad WS(P_1) > \alpha\% \ WS(P_2) \ \ \&\& \ \ WS(P_1) < CS \ \ \&\& \ \ \alpha\% \ CIR(P_1) < CIR(P_2)$$

Note that by taking $\alpha = 0$, the simulator reduces to the Execution-Only Algorithm. The simple model then only checks whether the working set is smaller than the cache size.

3.3 Benchmarks and Platforms

In order to test the efficiency of the use of cache models in iterative compilation, we use many small kernel benchmarks exhibiting widely different memory access behavior, on several data input sizes and several platforms. In total we collected 162 measurements that we use to quantify the efficiency of our approach to produce statistically relevant results. The benchmarks considered are the most important and compute intensive kernels from multimedia applications. We use all 6 possible loop permutations of matrix-matrix multiplication on 3 data input sizes of 256, 300 and 301. We denote these by MxM-IJK, MxM-IKJ etc. We use the 2 loop orders in matrix-vector multiplication on data input sizes 2048, 2300 and 2301. We use 6 loop orders in Forward Discrete Cosine Transform (FDCT), one of the most important routines from the low level bit stream video encoder H263. This routine consists of an initialization loop, two 3D computation loops and one finalization loop. We also use the

6 variations of the second main computation loop from FDCT that consists of multiplication of a transposed matrix. We use data input sizes of 256, 300 and 301. Finally, we use a Finite Impulse Response filter (FIR), one of the most important DSP operations, with data sizes of 8192, 8300 and 8301.

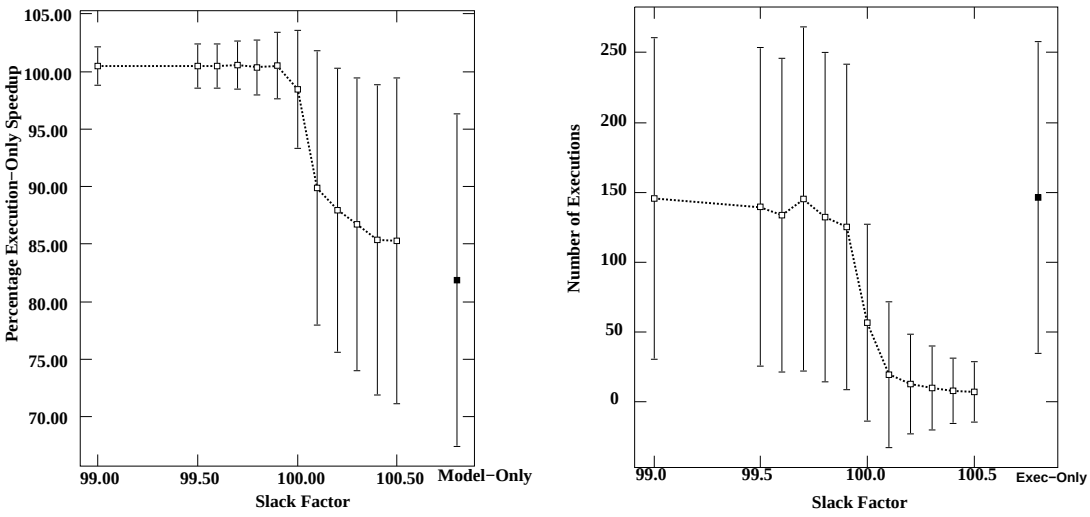
We executed on the following platforms: Pentium II, Pentium III, HP-PA 712, UltraSparc I. We used the native Fortran compiler or g77, with full optimization on. In this paper we consider loop tiling, with tile sizes of 1 to 100, and loop unrolling, with unroll factors of 1 to 20. For the Model-Only Algorithm, we take 500 random points, and for the Execution-Only and Model+Execution Algorithm, we allow a maximum of 400 program executions.

4 Performance of Cache Models

In this section we discuss the results we obtained for iterative compilation incorporating cache models. We use the speedup obtained from the Execution-Only Algorithm [13], called the Execution-Only speedup below, as the base line to which we compare results produced by the cache models. We also compare the number of executions to the number of executions required by the Execution-Only Algorithm.

4.1 Cache Simulator

In this section we discuss the performance of iterative compilation incorporating a full cache simulator. First, in Figure 5(a), we plotted the speedup obtained from the Model-Only Algorithm as the point labeled Model-Only. On average, this speedup is only 82% of the



(a) Performance

(b) Number of Executions

Figure 5: Performance Cache Simulator

Execution-Only speedup, with a standard deviation of 15. This means that hit rate is not an exact model for execution time and, in fact, it is less accurate than we expected. We can explain this by observing that the hit rate is an average over the entire execution of a program and does not discriminate between a burst of misses and the same number of misses that is more uniformly distributed. However, for a real processor, a burst of misses will stall the machine for a significant number of cycles whereas occasional misses may be hided.

Second, in Figure 5(a) we show the speedup from the Model+Execution Algorithm as a fraction of the Execution-Only speedup for different values of α. In Figure 5(b), we have plotted the number of real program executions against α. The point labeled Exec-Only in this figure corresponds to the number of iterations of the Execution-Only Algorithm. We see that for values of α up to 99.9 we obtain full speedup with a small standard deviation. For $\alpha \geq 100$, the average speedup drops quite fast. However, we need as many program

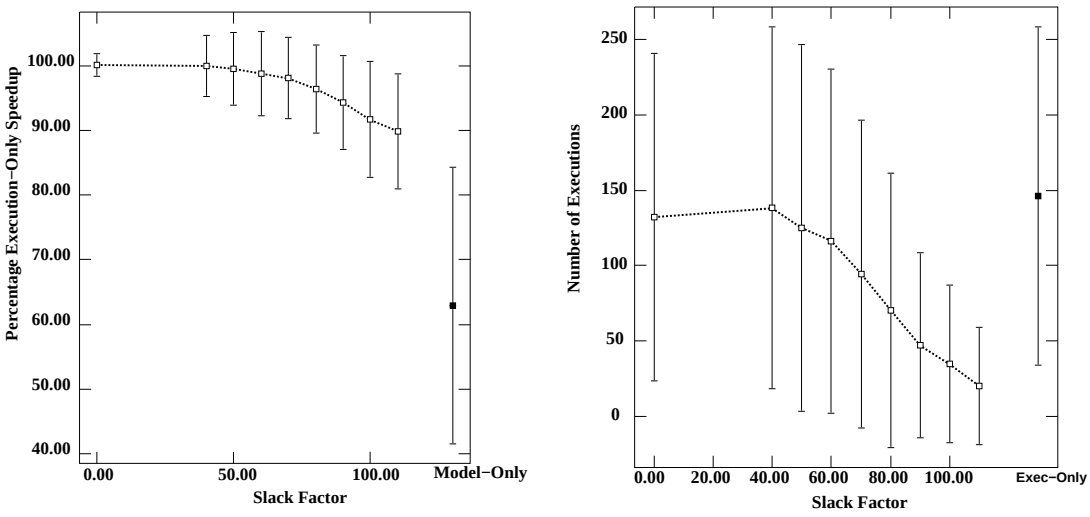
executions as for the Execution-Only Algorithm for values of α up to 99.5. We can explain this by observing that the cache hit rate for the benchmarks is, on average, high. For low values of α, most transformations that are tried in the search will have a hit rate within $\alpha\%$ of the best hit rate so far and hence they are executed. This means that the search proceeds quite similar to the Execution-Only Algorithm and the speedups found are the same.

The number of iterations required drops for $99.5 \leq \alpha \leq 100$ whereas speedups are around 100%. For $\alpha = 100$, the average number of executions is reduced to 64. For $\alpha > 100$, there is a sharp and sudden drop in the speedup that we reach and in the number of program executions. This means that we quickly find a transformation with a high hit rate that subsequent transformations are not capable of improving by more than 100%. However, the actual execution time for this transformation is far from optimal.

We conclude that iterative compilation incorporating a full cache simulator is capable of reaching the same speedups as the Execution-Only Algorithm does. For values of the slack factor α of 99.9 or 100, we reach this performance using less program execution than for the Execution-Only Algorithm. In Section 5 we will investigate this improvement in more detail.

4.2 Simple Cache Models

In this section we discuss the performance of the simple cache model. First, the speedup obtained from the Model-Only Algorithm is plotted as the point labeled Model-Only in Figure 6(a). We obtain 62% on average of the Execution-Only speedup with a large standard deviation. This means that a search using this simple model only (as has been proposed by Coleman and McKinley [7]) produces suboptimal results. In Figure 6(a), we show the speedup



(a) Performance

(b) Number of Executions

Figure 6: Performance Simple Cache Model

for the simple cache model as a fraction of the Execution-Only speedup for different values of α. For $\alpha \leq 80$ we reach at least 95% of this speedup. However, the standard deviation shows that there are many benchmarks that reach a much lower speedup. At the same time, we observe from Figure 6(b) that the number of program executions drop to well below the number of executions for the Execution-Only Algorithm (labeled by Exec-Only). From our detailed analysis in the next section, we deduce that an optimal value of α equals 40. In this case, the simple model is capable of excluding many executions early in the search and reaching levels of optimization that are close to those of the simulator. In general, the speedup obtained is less than the Execution-Only speedup, which shows that the simple model excludes many program versions that actually have good execution times.

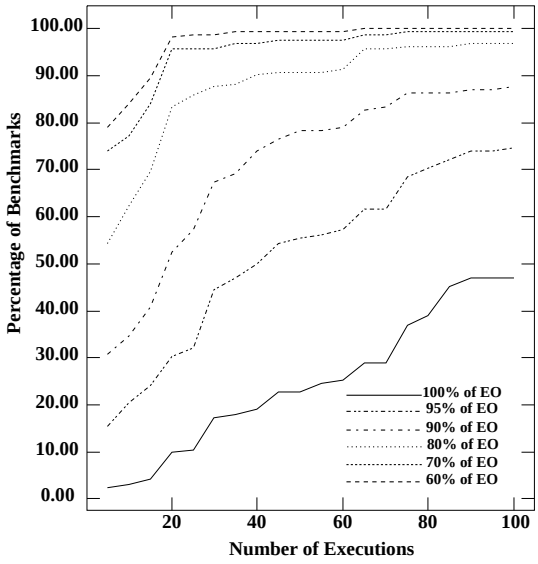


Figure 7: Trade-off Graph for the Execution-Only Algorithm

5 Trade-off between Number of Executions and Levels of Optimization

In this section we give a detailed description of the average performance of iterative compilation using cache models when we limit the number of program executions. First, in Figure 7, we depicted a trade-off graph between the number of executions and the levels of optimization for the Execution-Only Algorithm. We have discussed these trade-off graphs in more detail in [13]. The graph contains a number of *equi-optimization curves* that indicate the percentage of benchmarks that reach a certain level of optimization as a function of the number of program executions. This level is expressed in terms of how close the speedup is to the Execution-Only speedup (between 0 and 100%). From this graph we can deduce, for example, that after 100 iteration, 48% of the benchmarks were fully optimized and thus reached 100%

of the Execution-Only speedup. Likewise, after 50 iterations, 77% of the benchmarks reached at least 90% of the Execution-Only speedup. After 20 executions, almost every benchmark reached at least 60% of the Execution-Only speedup. Note that this graph is based on 162 experiments, in each of which we determined the speedup for 0 to 100 executions. Hence, the speedup of one benchmark after a certain number of executions contributes 0.6% on the y-scale. Therefore, we claim that this graph is statistically accurate.

Next we construct trade-off graphs for the static models and both search algorithms. We show in these graphs how close we come to the maximal speedup reached by the Execution-Only Algorithm. We quantify the improvement by comparing the trade-off graph for the Execution-Only Algorithm with the other trade-off graphs.

5.1 Model-Only Algorithm

In this section we discuss the trade-off that we obtain from the Model-Only Algorithm for both cache models, depicted in Figure 8. Consistent with our earlier findings that the Model-Only Algorithm reaches only 80% or less of the Execution-Only speedup, we see that the trade-off is low. Both models fair equally poor. Only a few benchmarks reach 100% or 95% of the Execution-Only speedup and only half the number of benchmarks reach 60% of this speedup. Comparing this trade-off graph with Figure 7, we see that the Execution-Only Algorithm only requires a few program executions to reach the same levels of optimization as the Model-Only Algorithm does. We conclude that a search technique using static models alone, as has been proposed by Wolf, Maydan and Chen [26], is not capable of obtaining the same levels of optimization as iterative compilation that uses profiling information can.

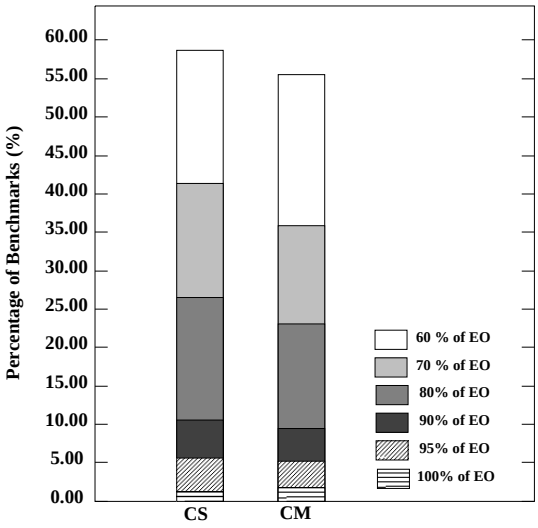
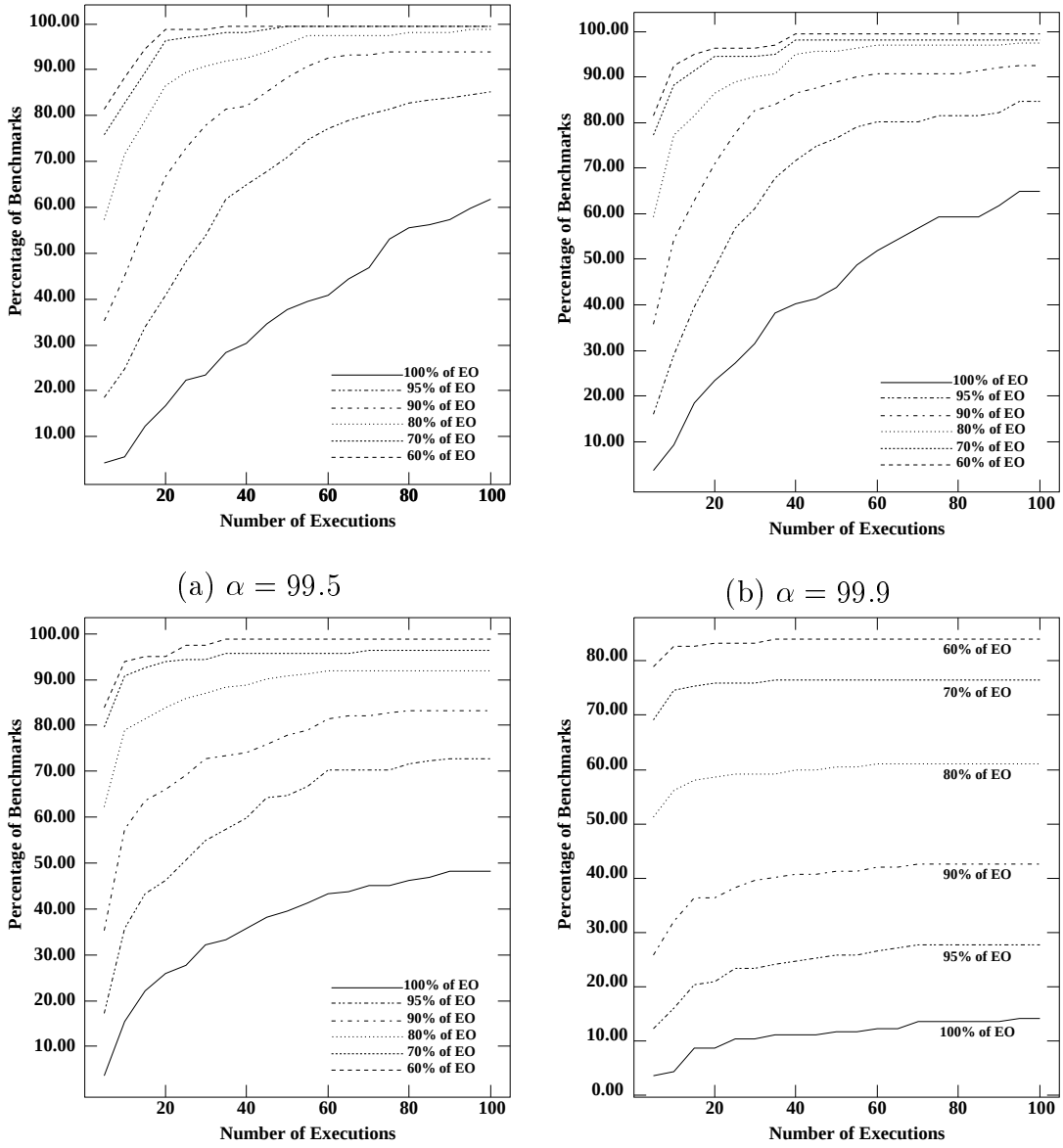


Figure 8: Tradeoff Model-Only for Both Cache Models

5.2 Cache Simulator

In Figure 9 we show trade-off graphs for the cache simulator for different values of α. In Figure 10 we show the improvement over the Execution-Only Algorithm for 4 equi-optimization curves. From Figure 10(a), we observe that the improvements over the Execution-Only Algorithm are substantial: up to 4 times as many benchmarks reach full optimization within 25 executions for $\alpha = 100$. We observe that, for 30 or more executions, the trade-off is better for $\alpha = 99.9$ than for $\alpha = 100$. The improvement drops slowly as the number of executions increase so that, eventually, cache models provide no improvement over the Execution-Only Algorithm for more than one hundred executions. For $\alpha > 100$, there is a substantial degradation with respect to the Execution-Only Algorithm for almost any number of executions. In general, for $\alpha = 99.9$ or 100, there is a high improvement for up to 20 or 30 executions but for more executions this improvement drops. This shows that a cache simulator is highly



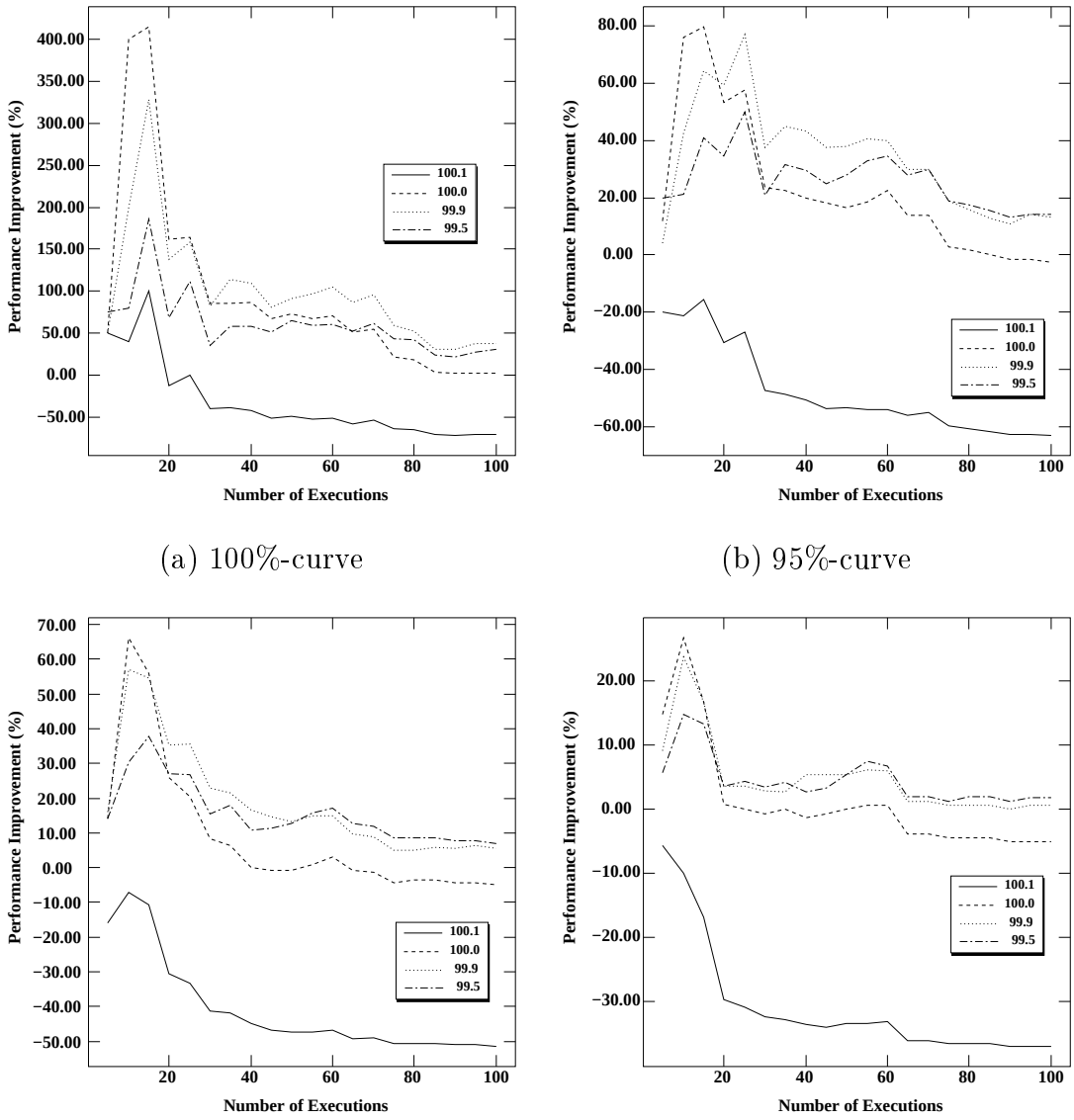
(a) $\alpha = 99.5$

(b) $\alpha = 99.9$

(c) $\alpha = 100.0$

(d) $\alpha = 100.1$

Figure 9: Trade-off Graphs for Cache Simulator for Different Values of α



(a) 100%-curve

(b) 95%-curve

(c) 90%-curve

(d) 80%-curve

Figure 10: Improvement over the Execution-Only Algorithm for Simulator for Different Values of α

	EO 50 exs.	**CS** 22 exs.	**EO** 100 exs.	**CS** 52 exs.
100%-curve	23%	24%	48%	47%
95%-curve	55%	54%	75%	77%
90%-curve	78%	76%	88%	89%
80%-curve	90%	88%	96%	95%

Table 1: Comparison Number of Executions for Execution-Only and Simulator ($\alpha = 99.9$)

effective in eliminating transformations that do not perform well early in the search. However, for time critical embedded systems where failure to meet time constraints might cause failure of the entire system, we might need to spend many more executions and even the inclusion of a cache simulator is not sufficient to bring down the number of required executions to an amount that may be evaluated within a few minutes.

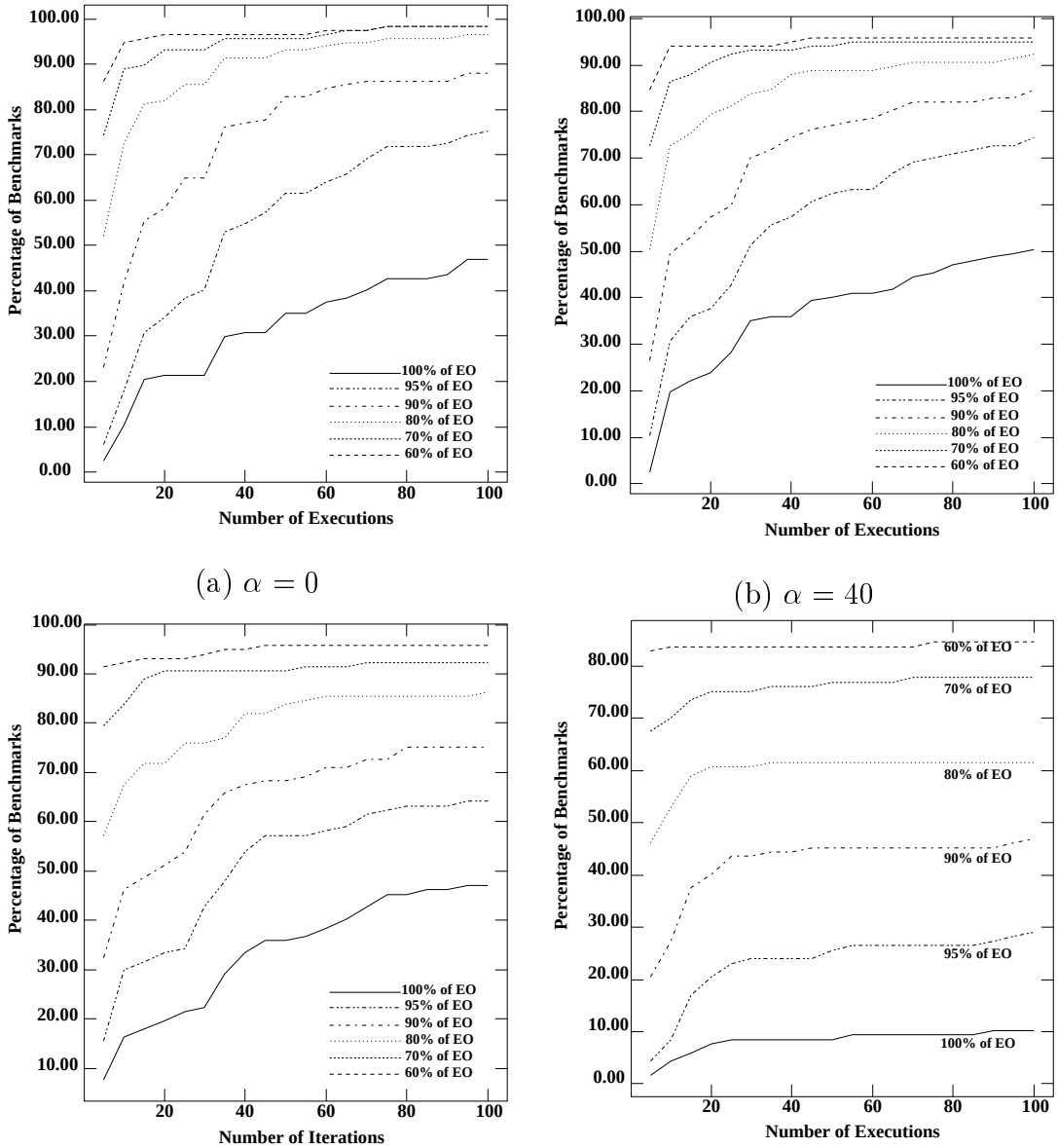
Next we turn attention to the improvement in the number of executions needed so that the simulator has approximately the same performance as the Execution-Only Algorithm. In Table 1 we show how many executions are required for the simulator to reach the same level of optimization as the Execution-Only Algorithm obtains after 50 and 100 executions. We see that the simulator only needs about half as many executions as the Execution-Only Algorithm. We conclude that incorporating a highly accurate cache model can reduce the number of executions by 50% compared to the Execution-Only Algorithm and still reach the same level of optimization.

5.3 Simple Cache Models

In Figure 11 the trade-off graphs for the simple cache model are given. We reach levels of optimization that are lower than for the Execution-Only Algorithm. From Figure 12, we observe that for a small number of executions (up to 20) the simple model outperforms the Execution-Only Algorithm significantly for a slack factor of $\alpha = 40$. Note that for $\alpha = 0$, where we only have the constraint $WS < CS$, we obtain surprisingly good results. However, for $\alpha \geq 80$, the simple model will actually degrade the Execution-Only Algorithm since improvements less than zero are obtained. This holds, in particular, the original Coleman/McKinley model ($\alpha = 100$) that is comparable to the simulator in case $\alpha = 100.1$. Comparing Figures 10 and 12, we see that for small numbers of program execution, this simple model with $\alpha = 40$ actually reaches about 80% of the improvement of the simulator. For higher numbers of execution, however, the simulator outperforms the simple model significantly and a simple model will only slightly improve the Execution-Only Algorithm.

The trade-off graphs in Figures 11 show that simple models are not adequate to reach full optimization. In particular, this means that these models are of limited value in case we need to highly optimize applications, as for instance is the case in embedded systems.

We cannot construct a table like Table 1 for the simple cache model, since inspection of the trade-off graphs show that there does not exist a clearly defined number of executions after which the simple model reaches the same level of optimization as the Execution-Only Algorithm does after 50 or 100 executions.



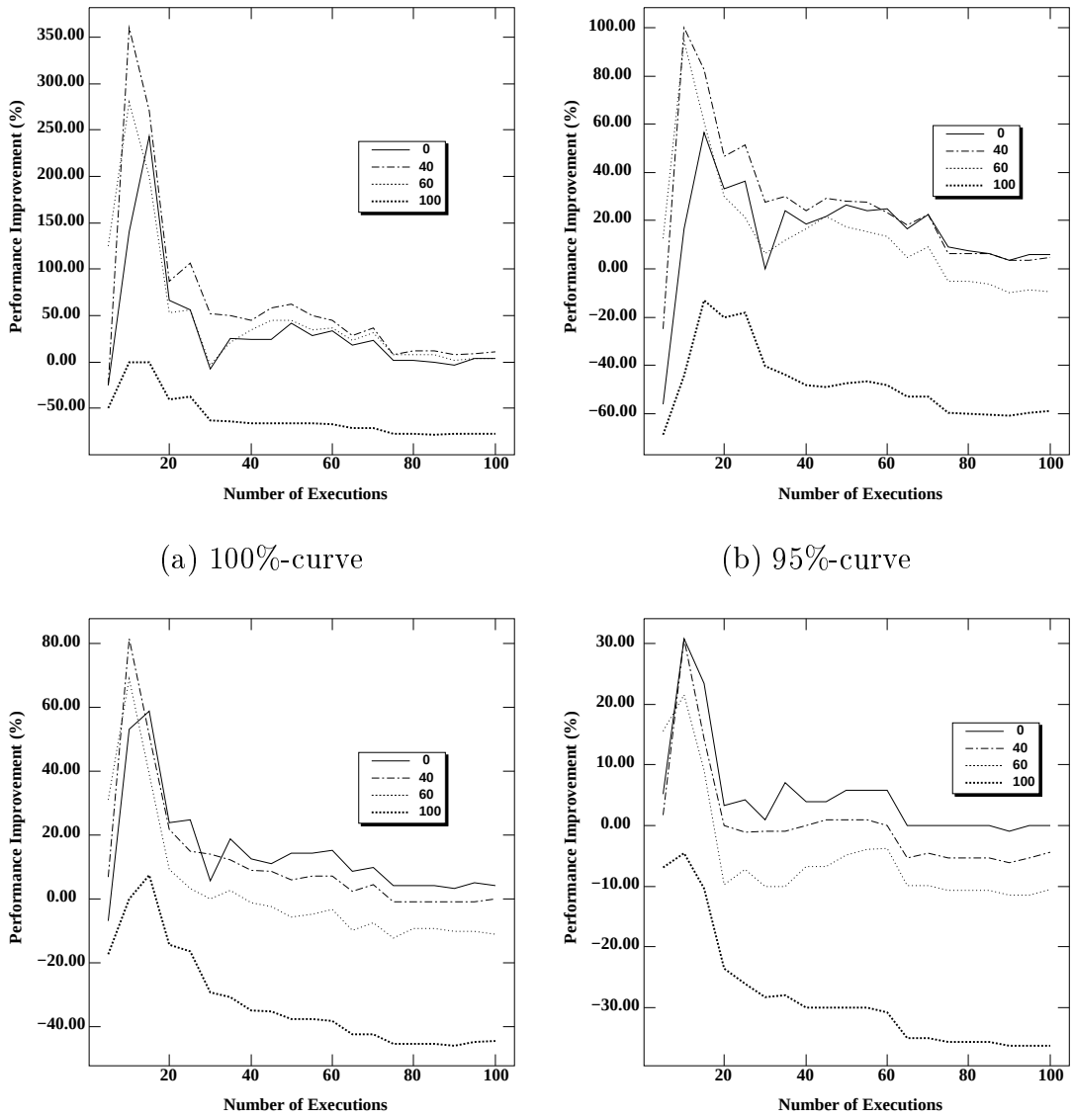
(a) $\alpha = 0$

(b) $\alpha = 40$

(c) $\alpha = 60$

(d) $\alpha = 100$

Figure 11: Trade-off Graphs for Cache Model for Different Values of α



(a) 100%-curve

(b) 95%-curve

(c) 90%-curve

(d) 80%-curve

Figure 12: Improvement over the Execution-Only Algorithm for Cache Model for Different Values of α

6 Discussion

In this paper we have discussed the inclusion of static cache models in iterative compilation where we search for the best optimization by using model information and actual execution times. In the entire spectrum of possible models, we have focussed on the two extremes: a highly expensive but accurate model (*viz.* a cache simulator), and an inexpensive but less accurate analytical model that tries to capture cache effects through simple expressions that approximate cache behavior, proposed by Coleman and McKinley [7].

First, we have shown that searching for the best transformation using only static models, as proposed by Wolf, Maydan and Chen [26], yields equal levels of optimization that are far less than those obtained by iterative compilation. For this model-only approach to be effective, we need more detailed models that also take into account, for instance, the distribution of the misses in the memory reference trace. However, based on our present experience with a full cache simulator, we claim that one needs real program profiling to find the best transformation. This approach has recently received more attention by several authors [9].

Second, accurate cache models improve iterative compilation to a large extend, reducing the number of executions by 50%. However, for a production compiler we need cheap analytical models. Possible candidates are Cache Miss Equations [10] that can be implemented efficiently using a stochastic approach [24]. Temam and coworkers have proposed other cache modeling techniques [22]. Furthermore, we need to model more features of the underlying hardware, like ILP exploitation, in order to be able to predict execution times accurately.

Third, less accurate models, like the simple model proposed by Coleman and McKinley [7] are capable of improving iterative compilation as much as accurate models do, but only for a

very limited number of program executions.

Fourth, we have included a slack factor α in the models. The performance of the models is highly sensitive to this slack factor. We have shown that for the cache simulator the best option is $\alpha = 100$ when we have a limited budget of up to 30 program executions and $\alpha = 99.9$ for higher budgets. However, for the simple cache model, a value of $\alpha = 40$ is best. In this case, a performance improvement of up to 80% of the improvement obtained by the simulator can be realized. However, if we would choose a larger value for α in the simple model, we may actually degrade from the Execution-Only Algorithm. In particular, this degradation is considerable for $\alpha = 100$, which corresponds to the original Coleman/McKinley model. Furthermore, a value of $\alpha = 0$ for the simple model which corresponds to a constraint that the working set is smaller than the cache size, produces remarkably good results.

Finally, in this paper we have considered computational kernels only. In [20] we have discussed an approach to applying iterative compilation to entire, large applications. We need to include many more transformations for this purpose. We have shown that a tree-like approach, where tiling and unrolling are just one node, can be efficient. By careful use of profiling information, many loop nests can be tiled and unrolled in tandem. In this way we expect that we can optimize large applications within 1000 program executions. Even if compilation and execution times for these applications are large, the optimization process can be done within several days or a few weeks at most. Although this is time consuming, hand optimization of these applications will takes months rather than weeks and, moreover, iterative compilation is an automatic procedure requiring no human interaction. Therefore iterative compilation will be highly competitive for time critical applications.

7 Related Work

Over the past years, many authors have considered limited search techniques for optimization purposes. In particular, for tiling and unrolling, Coleman and McKinley [7] and Lam, Rothberg and Wolf [16] employ a restricted search for tile sizes based on a simple cache model. Carr [4] computes unroll factors in order to minimize 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 to these approaches, the present approach uses actual execution times and moreover considers both loop tiling and unrolling at the same time.

Haley and Dongarra [25], and Bilmes et al. [3] 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 optimize BLAS routines and are not general purpose compilers.

Wolf, Maydan and Chen [26] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. Han, Rivera and Tseng [11] also describe a compiler that searches for tile and pad sizes using static models. In contrast to the present approach, however, their compilers use static cost models to evaluate the different optimizations. From this paper it follows that our approach based on actual execution times delivers superior performance and can adapt to any architecture, requiring no prior modeling phase. Chow and Wu [5] apply 'fractional factorial design' to decide on a number of experiments to run for selecting a collection of compiler switches. They, however, focus

on on/off switches and do not consider the choice of parameter values that might come from a large range of values. Bodin and co-workers explore in [23] the interplay between loop unrolling and software pipelining. This approach can be fully integrated with the present approach since they target a different phase in the compiler, namely, the code generation phase. In [19], Nisbett proposes a genetic algorithm approach to searching.

Over the past years, many proposals have been put forward to use profile information, for example, in the creation of superblocks [12] or hyperblocks [17] to enable efficient scheduling for ILP processors. These techniques are currently being employed in commercial compilers [6]. Profiles are also used to identify runtime constants that can be exploited at compile time [18]. 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 [9]. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

8 Conclusion

In this paper we have discussed the inclusion of cache models in iterative compilation where we search for an optimal optimization. In previous work we have shown that iterative compilation can yield high levels of optimization, outperforming static techniques significantly. We have considered two types of models: one highly accurate but highly expensive (cache simulator) as an upperbound, and one less expensive but also less accurate as a lowerbound. First, we have shown that these models alone while using no program execution are not capable of producing levels of optimization as high as iterative compilation can. Second, we have shown

that iterative compilation incorporating accurate models is capable of reducing the number of required program executions by 50% and still obtain the same levels of optimization. Less accurate models can improve iterative compilation in case there is a small budget of profiles. However, we have also shown that these less accurate cache models may actually degrade the performance of iterative compilation in case more profiles can be afforded. We conclude that, in order to obtain maximal speedup, accurate models together with a limited number of profiles are required.

References

[1] M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P. O'Boyle, E. Rohou, R. Sakellariou, A. Seznec, E.A. Stöhr, M. Treffers, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 98*, volume 1470 of *Lecture Notes in Computer Science*, pages 1123–1130, 1998.

[2] 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.

[3] 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.

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

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

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

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

[8] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr. An overview of the Intel IA-64 compiler. *Intel Tecnology Journal*, Q4, 1999.

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

[10] 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.

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

[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, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. Technical Report 2000-07, LIACS, Leiden University, 2000. Submitted to PACT2000.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] M.D. Smith. Overcoming challenges to feedback-directed optimization. In *Proc. Dynamo'00*, 2000.

[22] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proc. SIGMETRICS'94*, pages 261–271, 1994.

[23] 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.

[24] X. Vera, J. Llosa, A. González, and C. Ciuraneta. A fast implementation of Cache Miss Equations. In *Proc. CPC2000*, pages 319–325, 2000.

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

[26] 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.