The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling

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

* LIACS, Leiden University, the Netherlands
{kisuki,peterk}@liacs.nl
† Department of Computer Science, Florida State University, USA
gallivan@cs.fsu.edu
‡ Institute for Computing Systems Architecture, Edinburgh University, UK
mob@dcs.ed.ac.uk

Abstract

In previous work we have introduced iterative compilation where we search for the best program transformations by profiling many variants and selecting the one with the shortest execution time. We have shown that this approach produces very efficient code, outperforming static techniques significantly. However, this approach is extremely time consuming. In order to reduce compilation times, we discuss in this paper how to incorporate static models. We focus on tiling and unrolling as transformations and cache models, as a test case for the feasibility of the approach. We show that a highly accurate model as a filter to profiling can reduce the number of executions by 50%. We also show that using a simple model to rank transformations and profiling only those with highest ranking can reduce the number of executions even further, in case we have a limited number of profiles at our disposal. We conclude that a production compiler might perform best using the last approach.

1 Introduction

An important task of a compiler is to transform a source program into an efficient variant for a specific target platform. Traditionally, compilers use static, simplified machine models and hardwired strategies to determine the order in which to apply certain transformations and their parameter values, such as tile sizes or unroll factors. However, actual machines and their back end compilers have so complex an organization that this approach has much difficulty in delivering optimal code. Recently, several approaches have been proposed to use profiling and run time information for determining which transformations are best for a given application and platform [1, 9, 8, 18]. In particular, the present authors have proposed an optimization framework called *iterative compilation* in which many variants of the source program are generated and the best one is selected by actually profiling these variants on the target hardware [11]. This framework is essentially target neutral since it consists of a driver module that navigates through the optimization space and a source to source restructurer that allows the specification of the transformations it employs. The native compiler is used as back end compiler and it is treated together with the platform as a black box.

In [11] we have shown that this approach can obtain high levels of optimization, outperforming existing approaches significantly. In some cases, improvements of several hundreds of percents are reached. However, we need many profiles for this purpose and compilation time, that includes searching, program transformation, back end compilation and profiling, is extremely time consuming. In order to reduce compilation time, we propose to incorporate target specific models in the driver module, covering part of the behavior of the underlying platform, to estimate the effect of transformations. We can distinguish two approaches.

Execution driven The driver searches through the optimization space and the model is used to filter out bad candidate transformations during the search. Only if the model predicts that a transformation may be better than the best one found so far, profiling of the transformed program will take place.

Model driven The models are used to rank a collection of transformations before any profiling takes place. Only the transformations with highest ranking are profiled. The transformation that gives rise to the shortest execution time is chosen.

In this paper, we present a feasibility study into the effects of including models in the driver to reduce compilation time. We restrict attention to two well-known and much studied program transformations: loop tiling [7, 12, 10] and loop unrolling [14]. Loop tiling is targeted towards cache exploitation by dividing loops in small tiles such that the working set of one tile can be kept in the cache. Loop unrolling duplicates the loop body, thereby exposing more instructions to the hardware that can be executed in parallel. Moreover, loop unrolling (and in particular unroll-and-jam [4] that is considered in the present paper) affects the memory reference pattern of the loop and thereby affects the locality of the loop too. These two transformations, therefore, are highly interdependent and the compound result of applying them in terms of execution time gives rise to a very irregular optimization space [3]. Since the dominant effect of the transformations is their effect on cache behavior, static cache models are the prime models of interest. We assume that the models are far less costly to evaluate than profiling the program. We use a simple working set model as proposed in [7] that is easy to evaluate. We have also implemented a full level 1 cache simulator that has ideal static knowledge of cache behavior. We use this model as an upperbound case, and compare the efficiency of the simple model to it. Since a simulator can actually be slower than profiling the program, we do not propose to use a simulator in a realistic compiler.

This paper is organized as follows. In Section 2 we briefly discuss the implementation of the iterative compilation system and 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 and give a detailed analysis of the levels of optimization that can be reached with a limited number of program executions. In section 5 we discuss the results obtained in this paper. In section 6 we discuss related work and we draw some concluding remarks in section 7.

2 Iterative Compilation

In this section we briefly discuss how the iterative compilation system is implemented and review its performance. For more details, consult [11]. The compilation system is centered around a driver module that keeps track of the different transformations evaluated so far and decides which transformations to apply next. The driver invokes the source to source compiler MT1 that has two mechanisms to control the application of transformations: a Transformation Definition Language (TDL) and a Strategy Specification Language (SSL) [2]. Each transformation needs to be specified and named in the TDL file. The driver constructs an SSL file to specify the order in which to apply certain transformations and outputs it to MT1. After a predetermined number of profiles, the driver stops searching and outputs the transformed program with the shortest execution time.

In [11] we have shown that iterative compilation is capable of yielding high levels of optimization, outperforming several static techniques significantly. The code produced by iterative compilation comes within 2% of the best possible speedup that can be obtained on our benchmarks using tiling and unrolling. We need 116 executions on average to reach maximal speedup, but invariably the largest improvement is obtained early in the search: 90% of the maximal speedup is obtained within 30 executions. Compilation time, that includes searching, program transformation, back end compilation and profiling, is almost linear in the number of executions, with program execution the dominant factor. On average, we need less than 7 minutes for 50 executions. Since we assume that model evaluation is far less costly than profiling, these compilation times still apply for the present case. This approach is best suited when high levels of optimization are required and compilation times can be amortized over many production runs and particularly for embedded systems.

In [15] we present an implementation of an iterative compilation system for realistic 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 additionally inserting counters around each loop, one run of the program can give timing information for many loops. In this way we can optimize large applications within a reasonable number of program executions.

In the long term we envisage a compilation system where the user can trade-off levels of optimization and compilation time by tuning the complexity and accuracy of static models and the number of programs that are executed. Hence, in this paper emphasis is placed on the trade-off between number of executions and levels of optimization.

3 Experiment

In this section, we discuss two cache models, seven search algorithms and our benchmarks and platforms.

3.1 Cache Models

First, as an upperbound for the present approach, we use a full level 1 cache simulator to compute hit rates. All memory accesses are assumed to be issued to the cache although some memory accesses can actually be held in registers, depending on the register allocator of the back end compiler. This model is too expensive to be used in practice but we include it since it is more accurate than any other static model. We interpret results obtained for the simulator as upperbound results with which we can compare other models. We are currently implementing a sophisticated cache model [16] that is reported to be almost as accurate as a real simulator.

Second, as a realistic case, 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 selects the tile size giving rise to the largest working set that still fits in the cache.

3.2 Search Algorithms

3.2.1 Execution-Only and Model-Only Algorithms

The Execution-Only Algorithm (EO) generates a number of candidate transformations and profiles the transformed source program. It selects the program with the shortest execution time. This algorithm has been discussed in more detail in [11] where we have considered several algorithms to navigate through the optimization space. We have shown that these algorithms perform equally well. Below, we use the Execution-Only Algorithm based on random search as the base case with which to compare the other strategies. The speedup obtained from EO is called the *Execution-Only speedup*, or EO speedup. Since, for a given application, it is impossible to determine its absolute minimum execution time, we consider the EO speedup the best speedup that can be obtained using our approach. Next, the Model-Only Algorithm (MO) generates a number of candidate transformations and selects the transformation that gives rise to the highest hit rate computed by the cache simulator. This algorithm represents the traditional approach of statically selecting optimizations.

3.2.2 Execution driven search

In Execution driven search, the driver generates a number of candidate points from the optimization space. A point is only executed if the cache model predicts a specified level of performance. Points with poor cache hit rates or improper working set sizes are filtered out, reducing the number of program executions. In contrast to Model driven search, the selection of points is done by the driver and the value of transformed program is verified during the search checking whether reduction in execution time is made. We consider the following two strategies.

- **EXEC-CS**

```
current = initial transformation
REPEAT
   next = next transformation
   IF H(next) ≥ α × H(current)
   THEN execute(next)
        IF exec_time(next) < exec_time(current)
        THEN current = next
```

In this strategy, the cache simulator is used to guide the search. A new point `next` is selected and its execution time is evaluated if the cache hit rate $H(\texttt{next})$ is within a factor α of the current best cache hit rate $H(\texttt{current})$. We have included a slack factor α to allow imprecision of the model. By experimentation, we found that $\alpha = 99.9\%$ is optimal. The search algorithm will stop after N combinations are executed or if the cache hit rates of all combinations are examined. The combination with the smallest execution time is selected.

- **EXEC-CM**

```
current = initial transformation
REPEAT
    next = next transformation
    IF WS(next) ≥ β × CS &&
        WS(next) ≤ γ × CS
    THEN execute(next)
        IF exec_time(next) < exec_time(current)
        THEN current = next
```

The second strategy is based on the Coleman/McKinley model. CS denotes the cache size and WS is the working set of one tile. This strategy selects on an lower and upper bound for the working set, so that only programs with a large working set are profiled. By experimentation, we found that optimal values are $\beta = 40\%$ and $\gamma = 50\%$. γ can be seen as the *effective cache size*. After executing N combinations or examining all combinations in the parameter space, the search algorithm stops and reports the fastest version.

3.2.3 Model driven search

The first step of the model driven search is to estimate the cache hit rates or working set sizes of a large collection of combinations of tile sizes and unrolling factors in the parameter space. These combinations are then ordered based on a particular ranking strategy and the ones with the highest ranking are compared by actually executing those combinations. In this way, feedback information is used to verify and support the cache models in order to achieve high levels of optimization. Equivalently, the process can be viewed as using the cache models to filter candidates from the entire parameter space relatively cheaply and thereby reducing the number of expensive execution time evaluations required. We consider the following three strategies.

- **MOD-CS1**

```
calculate cache hit rates
rank transformations on hit rate
current = best transformation
REPEAT
    next = next best transformation
    execute(next)
    IF exec_time(next) < exec_time(current)
    THEN current = next
```

First, we calculate the cache hit rate of a large collection of tile sizes and unrolling factors using the cache simulator. The N combinations with the highest hit rates are executed and compared. The one with the fastest execution time is selected.

- **MOD-CS2**

```
calculate cache hit rates
FOREACH Unroll Factor
    rank transformations using hit rate
current = initial transformation
FOREACH Unroll Factor
    next = next best for this unroll factor
    execute(next)
    IF exec_time(next) < exec_time(current)
    THEN current = next
```

In this strategy we consider the impact of loop unrolling more explicitly. We consider each unroll factor from 1 to 20 and compute the cache hit rate for a large set of tile sizes and that unroll factor. Then, for each unroll factor, the $N/20$ combinations with highest hit rate are selected and executed. The fastest combination is selected.

- **MOD-CM**

```
current = initial transformation
FOREACH Unroll Factor
    next = next largest tile size
        s.t. WS(next) ≤ γ × CS
    execute(next)
    IF exec_time(next) < exec_time(current)
    THEN current = next
```

In this strategy, we select for each unroll factor the largest tile size such that the working set WS

is within $\gamma\%$ of the cache size CS. In total, N combinations are executed. Like in the previous strategy, $N/20$ combinations with largest tile size are selected for each unroll factor. The fastest combination is selected.

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 a 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. For the Execution-Only Algorithm and the other strategies we allow a maximum of 400 program executions.

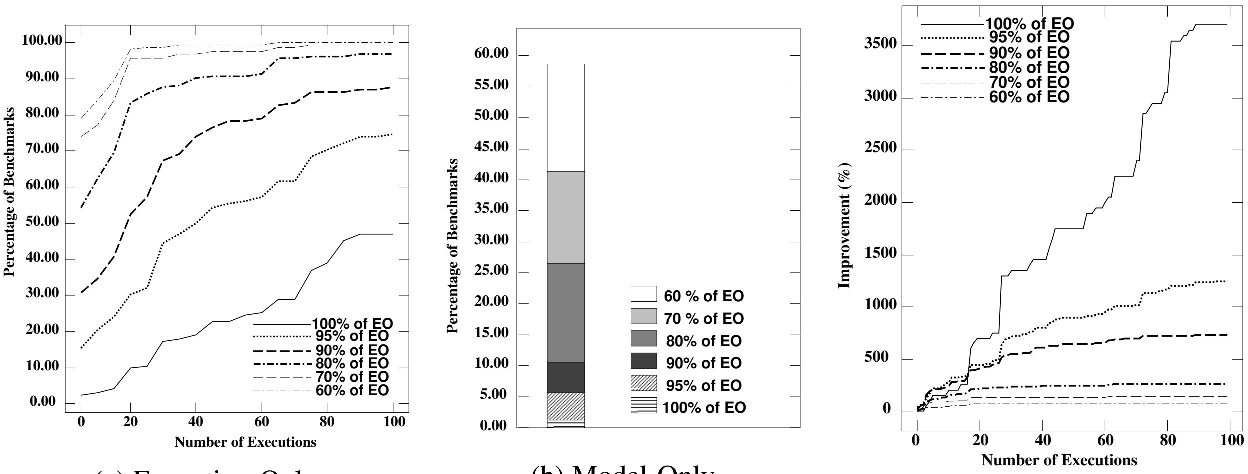
4 Results

In this section we discuss the results we obtained for iterative compilation incorporating cache models.

4.1 Execution-Only and Model-Only

For practical purposes, it is important to restrict the number of program executions to be small. To analyze the efficiency of iterative compilation for this case, we use a *trade-off graph* [11]. This graph contains a number of *equi-optimization curves* indicating the percentage of benchmarks that reach a certain percentage of the EO speedup as a function of the number of program executions. The trade-off graph for the Execution-Only Algorithm is depicted in Figure 1(a). From this graph we can deduce, for example, that after 100 executions, 48% of the benchmarks were fully optimized and thus reached 100% of the Execution-Only speedup. Likewise, after 50 executions, 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 and hence we claim statistical relevancy.

Next, we show the breakdown of the levels of optimization obtained from the Model-Only Algorithm in Figure 1(b). Since MO uses no profiles, this breakdown consists of a single bar. The speedup obtained is on average 82% of the EO speedup, with a standard deviation of 15%. 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. In Figure 1(c) we show the improvement of EO over MO as a function of the number of program executions. 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. For more executions, extremely high improvements are found: after 40 executions about 15 times as many benchmarks reach full EO speedup. Below we show that the other strategies improve on EO and hence their improvement on MO is at least as good as it is for EO. We conclude that a search technique using static models alone is not capable of obtaining the same levels of optimization as iterative compilation that uses profiling information can.

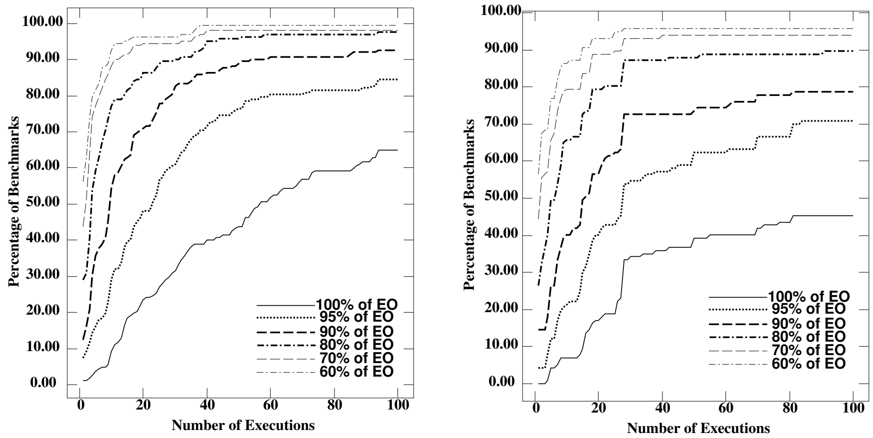


(a) Execution-Only

(b) Model-Only

(c) Improvement EO over MO

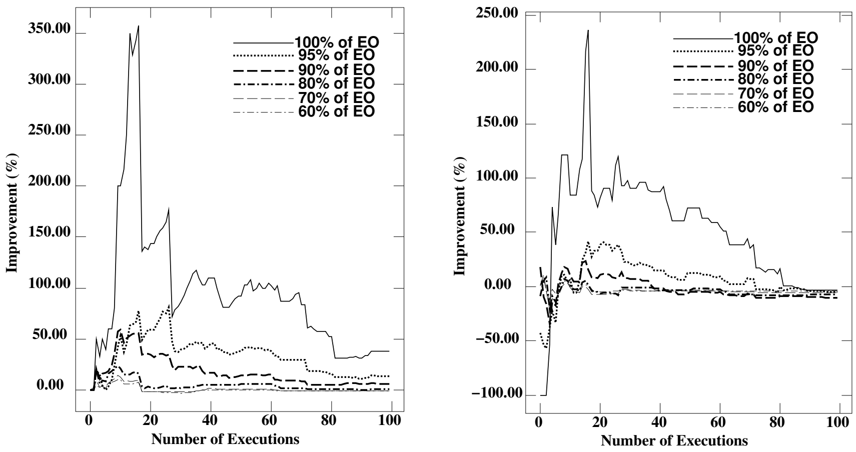
Figure 1. Trade-off Graphs



(a) EXEC-CS

(b) EXEC-CM

Figure 2. Trade-off graphs Execution driven search



(a) EXEC-CS

(b) EXEC-CM

Figure 3. Improvement Execution driven search over EO

4.2 Execution driven search

In Figure 2(a) we show the trade-off graph for EXEC-CS. From Figure 3(a), we observe that the improvements over the Execution-Only Algorithm are substantial: up to 3.5 times as many benchmarks reach full optimization within 20 executions. 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. This shows that an accurate cache model is effective in eliminating transformations that do not perform well early in the search.

Next, we turn attention to the improvement in the number of executions needed so that EXEC-CS has approximately the same performance as the Execution-Only Algorithm. Inspecting Figures 2(a) and 1(a), we see that EXEC-CS only needs about half as many executions as the Execution-Only Algorithm. For example, after 50 executions 23% of the benchmarks reach 100% of the EO speedup, 55% of the benchmarks reach 95% of the EO speedup and 78% of the benchmarks reach 90% of the EO speedup when we use the Execution-Only Algorithm. After 22 executions, 24% of the benchmarks reach 100% of the EO speedup, 54% of the benchmarks reach 95% of the EO speedup and 76% of the benchmarks reach 90% of the EO speedup using EXEC-CS. Hence the efficiency of EXEC-CS after 22 executions is about the same as the efficiency of EO after 50 executions. 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 levels of optimization.

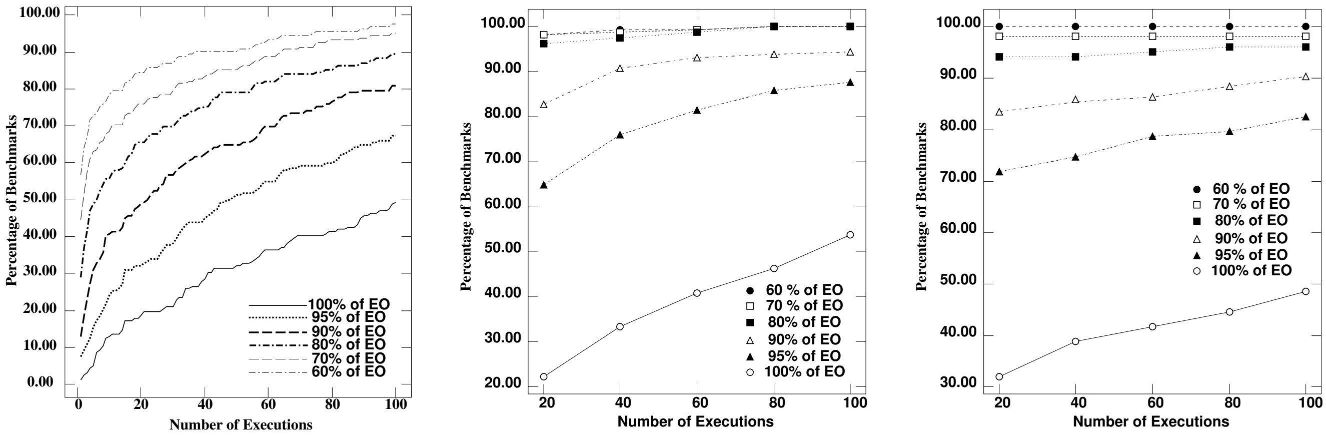
Comparing the trade-off graph for EXEC-CM in Figure 2(b) to the trade-off graph for EXEC-CS in Figure 2(a), we see that for 10 to 30 program executions both strategies perform equally well and improve EO substantially. For more than 30 executions, EXEC-CS is superior to the simple working set model. This is also apparent from the improvement graph in Figure 3(b) that shows that for less than 10 executions, EXEC-CM is inferior to EO. We also observe that less benchmarks reach the 60%- to 80%-curves than for EO. However, more benchmarks reach higher levels of optimization and therefore the total result is still positive.

4.3 Model driven search

In Figure 4 we have shown the graphs for the Model driven search strategies. From Figure 4(a) it follows that for up to 10 executions, the strategy MOD-CS1 is as effective as EXEC-CS. For more executions, EXEC-CS is superior. Figure 5(a) shows that not as many benchmarks reach 60% to 90% of the EO speedup as for the Execution-Only Algorithm for any number of executions. However, we see that more benchmarks reach 95% or 100% of the EO speedup. This shows that a cache model that assumes that all memory references go through the cache is not an adequate model for real platforms. However, at the source level there exists no knowledge of the back end code generation. Hence this assumption seems inevitable and it is made by other compilers that use a model to distinguish between alternative tile sizes and unroll factors.

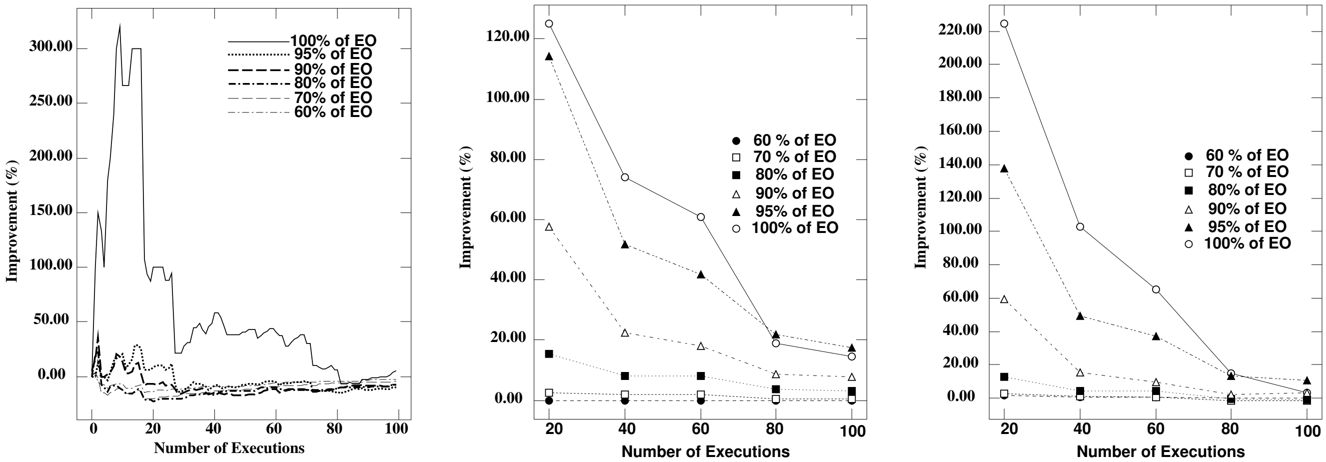
In the strategies MOD-CS2 and MOD-CM we execute programs in batches of 20 (one for each unroll factor). From Figure 4(b) it follows that MOD-CS, that selects the best tile size for each unroll factor, performs equally well for 20 executions as EXEC-CS. It is inferior to EXEC-CS for more executions. At the same time, it is superior to MOD-CS1 for every number of execution that we would allow. Figure 5(b) shows that MOD-CS2 is superior to EO.

Finally, Figure 4(c) shows the graphs for the last strategy MOD-CM. We see that this strategy is superior to EXEC-CS for 20 executions and reaches about the same levels of optimization for 40 executions. Comparing the trade-off graphs from Figures 2(a) and 4(c), we see that after 20 executions using the MOD-CM strategy, we reach the same level of optimization as EXEC-CS does after 30 to 40 executions. Comparing Figure 4(c) to Figure 1(a), we see that these levels of optimization are reached in the Execution-Only approach after about 70 executions. From Figure 5(b) it follows that MOD-CM is better than EO. This shows that the MOD-CM approach can reduce the number of required executions by 70% in case we only allow a limited number of 20 executions. MOD-CM is also superior to EXEC-CS1 and EXEC-CS2 for up to 80 executions. Only if we would allow 100 executions, these latter strategies prove to be better than MOD-CM. This shows that, although it is only a crude approximation



(a) MOD-CS1 (b) MOD-CS2 (c) MOD-CM

Figure 4. Trade-off graphs Model driven search



(a) MOD-CS1 (b) MOD-CS2 (c) MOD-CM

Figure 5. Improvement Model driven search over EO

of the exact hit rate, the working set size constraint is highly effective.

5 Discussion

In this paper we have discussed the inclusion of static cache models in iterative compilation where the best optimization is found by using model information and actual execution times.

First, we have shown that searching for the best transformation using only static models yields levels of optimization that are far less than those obtained by iterative compilation. Iterative compilation outperforms this static approach significantly already after a few program executions.

Second, using an Execution driven approach, accurate cache models improve the Execution-Only Algorithm substantially by reducing the number of executions by 50%. More accurate knowledge of the back end compiler and the target platform is required to enlarge this reduction. A tight feedback between source level restructurer and code generator is required. Obviously, this connection makes the implementation of an iterative compilation strategy more complex. Next, we have also shown that less accurate models, like the working set size constraint, can be almost as effective as a full cache simulator if only a limited number of up to 30 program executions is allowed. This number seems reasonable in a production compiler where large numbers of profiles would be too time consuming.

Third, comparing the Execution driven and Model driven approaches, we have shown that for up to 20 profiles the Model driven approach can be superior. In fact, 20 profiles using the MOD-CM strategy gives the same levels of optimization as the EXEC-CS strategy does after 30 to 40 profiles and as the Execution-Only Algorithm does after about 70 executions, giving an improvement of MOD-CM over EO of 70%. If we would allow more profiles, Execution driven selection is superior. Summing up, we can produce the following ordering of the search algorithms in terms of their efficiency after 20 and 40 executions.

20 execs. MOD-CM $\succ$ EXEC-CS $\sim$ MOD-CS2 $\succ$ EXEC-CM $\succ$ MOD-CS1

40 execs. EXEC-CS $\succ$ MOD-CM $\succ$ EXEC-CM $\succ$ MOD-CS2 $\succ$ MOD-CS1

Note that MOD-CS1, that profiles programs with highest hit rates, is inferior to any other strategy. This again stresses that an accurate cache model that only takes into account source level memory references does not model real memory access exactly. Next, we see that Model driven search is particularly effective for a small budget of executions and, in particular, MOD-CM is a good strategy performing almost as well as EXEC-CS. Since we expect that in a production compiler simple models like this working set size constraint will be preferable to highly complex models and that such a compiler would have a small budget for profiling, this result indicates that in this situation a Model driven search procedure can be preferable. Hence we believe that a good approach to implementing iterative compilation in a production compiler will consists of ranking transformations using simple models and profiling up to 20 candidates.

6 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 [12] 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 using a compile time search based on a static model. In contrast to these approaches, the present approach uses actual execution times and moreover considers both loop tiling and unrolling at the same time. Wolf, Maydan and Chen [20] describe a compiler that searches for the best optimization by considering the entire optimization space. Han, Rivera and Tseng [10] describe a compiler that searches for tile and pad sizes. In contrast to the present approach, their compilers use static cost models to evaluate the different optimizations. Chow and Wu [5] apply 'fractional factorial design' to decide on experiments to run for selecting a collection of compiler on/off switches. However, they do not consider the choice of a large range of parameter values. Bodin et al. explore in [17] the interplay between loop unrolling and software pipelining. This approach targets the code generation phase. Haley and Dongarra [19] describe a system for generating highly optimized BLAS routines that probes the underlying hardware to find optimal transformation parameters. In contrast to the present approach, this system is not a general purpose compiler.

The recently established workshop on Feedback Directed Optimization shows that many proposals are being put forward to exploit profile information [9]. Some of these techniques are currently employed in commercial compilers [6]. This paper can be seen as taking profiling one step further by using many profiles for deciding between many alternatives.

Diniz and Rinard propose Dynamic Feedback where several versions of a code section are generated [8]. At runtime, the best version is selected during a sampling phase and is used during the production phase of the program. This is repeated periodically to adapt dynamically to different execution environments. Voss and Eigenmann overlap code generation with program execution and periodically generate new variants of code sections using different compilers that run in a separate thread [18]. These approaches also search for the optimal code variant using profiling. Their most important assumption, however, is that the execution environment can change during the run of a program. For embedded systems this is not the case and the selection of optimizations can be done at compile time, as proposed in the present paper. Moreover, these approaches can cause code bloat and/or require special OS support which is also unwanted for embedded systems. Several approaches generate code at runtime, by exploiting runtime constants [1] or by partial evaluation [13]. For embedded systems these approaches are less suited since they entail code generation overhead at runtime and much of their effect can be obtained at compile time.

7 Conclusion

In this paper we have discussed the inclusion of cache models in iterative compilation where we search for the best optimization. We have considered Execution-Only, Model-Only, Execution driven and Model driven search strategies, based on two types of model. First, we have shown that a highly accurate model alone and using no profiling is not capable of producing levels of optimization as high as iterative compilation can. Second, we have shown that Execution driven search using accurate models is capable of reducing the number of required program executions by 50% and still obtain the same levels of optimization as Execution-Only iterative compilation does for any given budget of profiling. Third, we have shown that Model driven search using a simple model can improve Execution-Only iterative compilation by 70% in case there is a small budget of profiles. We conclude that, for a production compiler that would likely prefer simple models and few profiles, Model driven iterative compilation can be highly effective.

References

[1] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In *Proc. PLDI*, pages 149–159, 1996.

[2] A.J.C. Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Transformation mechanisms in MT1. Technical Report 2000-21, LIACS, Leiden University, 2000.

[3] 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.

[4] S. Carr. Combining optimization for cache and instruction level parallelism. In *Proc. PACT*, 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.

[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.

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

[8] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. PLDI*, pages 71–84, 1997.

[9] B. Calder et al., editor. *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999. `http://www-cse.ucsd.edu/users/calder/fdo`.

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

[11] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. PACT*, pages 237–246, 2000.

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

[13] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proc. PLDI*, pages 281–292, 1999.

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

[15] M.F.P. O'Boyle, P.M.W. Knijnenburg, and G.G. Fursin. Feedback assisted iterative compilation. 2000.

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

[17] 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.

[18] M.J. Voss and R. Eigenmann. ADAPT: Automated decoupled adaptive program transformation. In *Proc. ICPP*, 2000.

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

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