Cache Models for Iterative Compilation

Peter M. W. Knijnenburg[1], Toru Kisuki[1], and Kyle Gallivan[2]

[1] LIACS, Leiden University, the Netherlands
{peterk,kisuki}@liacs.nl
[2] Department of Computer Science, Florida State University, USA
gallivan@cs.fsu.edu

Abstract. In iterative compilation we search for the best program transformations by profiling many variants and selecting the one with the shortest execution time. Since this approach is extremely time consuming, we discuss in this paper how to incorporate static models. 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 will likely not deliver optimal code. In order to solve this problem, we have proposed *iterative compilation* where many variants of the source program are generated and the best one is selected by actually profiling these variants on the target hardware [7]. 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. We have shown [7] that this approach outperform existing static approaches significantly, albeit at the price of being extremely time consuming. In this paper, we propose to use static models that cover part of the behavior of the target platform, to estimate the effect of transformations and to exclude transformations that are likely to produce poor results. We 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.

We restrict attention to two well-known program transformations: loop tiling [5,8] and unroll-and-jam [4]. Both transformations are targeted towards cache exploitation. Unroll-and-jam, moreover, duplicates the loop body to expose more instructions to the hardware that can be executed in parallel. These two transformations, therefore, are highly interdependent and their compound result gives rise to a highly 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.

Recently, there have been approaches where the compiler searches the transformation space using static models [4,13]. These approaches, however, do not use profile information. Also, there have been several approaches to *feedback directed optimization*, in which run time information is exploited to alter a program [10]. We can distinguish between on-line and off-line approaches. On-line approaches optimize at runtime or during the lifetime of a program [1,6,11]. Off-line approaches include architectural tuning systems for BLAS [2,12] or DSP kernels [9]. For embedded systems an off-line approach is best suited since high compilation times can be amortized across the number of systems shipped.

This paper is organized as follows. In Section 2 we discuss the cache models used, the iterative search algorithms and the benchmarks and platforms. In Section 3 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 4 we discuss the results obtained in this paper and we draw some concluding remarks in section 5.

2 Experiment

Cache Models First, as an upperbound for the present approach, we use a full level 1 cache simulator to compute hit rates. 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. Second, as a realistic case, we use a simple model proposed by Coleman and McKinley [5] that 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. In order for models to be effective, we assume that they are far less costly to evaluate than profiling the program.

Execution driven search We consider the following two strategies.

– **EXEC-CS** A new point `next` is selected and evaluated if the cache hit rate $H($`next`$)$ is within a factor α of the current best cache hit rate $H($`current`$)$. By experimentation, we found that a slack factor $\alpha = 99.9\%$ is optimal. The search algorithm will stop after N combinations are executed.

```
current = initial transformation
REPEAT
   next = next transformation
   IF $H(\texttt{next}) \geq \alpha \times H(\texttt{current})$
   THEN execute(next)
         IF exec_time(next) < exec_time(current)
         THEN current = next
```

– **EXEC-CM** is based on the Coleman/McKinley model [5]. 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 $\gamma = 50\%$.

```
current = initial transformation
REPEAT
    next = next transformation
    IF $WS(\texttt{next}) \geq \beta \times CS$  &&
         $WS(\texttt{next}) \leq \gamma \times CS$
    THEN execute(next)
         IF exec_time(next) < exec_time(current)
         THEN current = next
```

Model driven search We consider the following three strategies.

– **MOD-CS1** 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.

```
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
```

– **MOD-CS2** 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.

```
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
```

– **MOD-CM** In this strategy, we select for each unroll factor the largest tile size such that the working set WS is within γ% of the cache size CS. $N/20$ combinations with largest tile size are selected for each unroll factor.

```
current = initial transformation
FOREACH Unroll Factor
    next = next largest tile size
        s.t. $WS(\texttt{next}) \leq \gamma \times CS$
    execute(next)
    IF exec_time(next) < exec_time(current)
    THEN current = next
```

Benchmarks and Platforms 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 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. 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. Loop tiling uses tile sizes of 1 to 100, and loop unrolling uses unroll factors of 1 to 20. We allow a maximum of 400 program executions.

3 Results

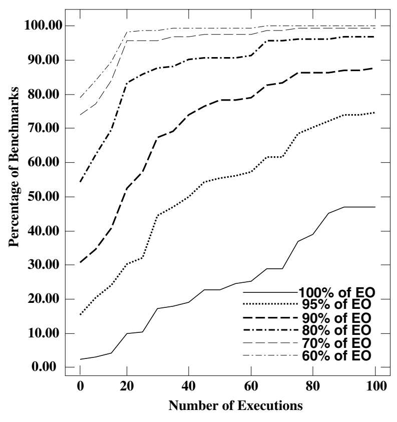
In this section we discuss the results we obtained for iterative compilation incorporating cache models. For practical purposes, it is important to restrict the number of program executions to be small. To analyze the efficiency of iterative compilation for this case, we use a *trade-off graph* [7]. This graph contains a number of *equi-optimization curves* indicating the percentage of benchmarks that reach a certain percentage of the maximal speedup as a function of the number of program executions. The trade-off graph for the algorithm that only uses profiles and no models and that is studied in [7], is depicted in Figure 1(a). We call this algorithm the Execution-Only Algorithm (EO) and it is used as the base case with which to compare the other algorithms. 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 maximal speedup. Likewise, after 50 executions, 77% of the benchmarks reached at least 90% of the maximal speedup. After 20 executions, almost every benchmark reached at least 60% of this speedup.

Execution driven search Inspecting Figures 1(a) and 1(b), we see that EXEC-CS only needs about half as many executions as the Execution-Only Algorithm and still obtains the same trade-off. For example, we can see that the levels of optimization obtained after 25 program executions is about the same as the levels of optimization obtained by EO after 50 executions, and likewise after 50 executions it is the same as EO obtains after 100 executions. Comparing the trade-off graph for EXEC-CM in Figure 1(c) to the trade-off graph for EXEC-CS in Figure 1(b), we see that for 10 to 30 program executions both strategies perform equally well and improve the Execution-Only Algorithm substantially. For more than 30 executions, EXEC-CS is superior to EXEC-CM.

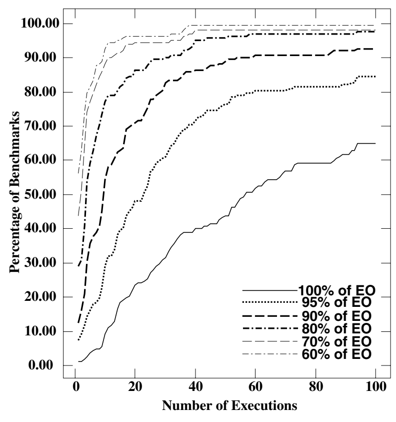
Model driven search From Figure 1(d) it follows that for up to 10 executions, the strategy MOD-CS1 is as effective as EXEC-CS. For more executions, EXEC-CS is superior. This shows that a cache model that assumes that all memory references go through the cache is not an adequate model for real platforms. Moreover, the left-most point in the trade-off graph for EXEC-CS1 corresponds to a strategy where we only use static model information, as is customarily done in traditional compilers. It follows that such a strategy is not able to reach levels of optimization that iterative compilation can.

In the strategies MOD-CS2 and MOD-CM we execute programs in batches of 20 (one for each unroll factor). From Figure 1(e) it follows that MOD-CS2 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.

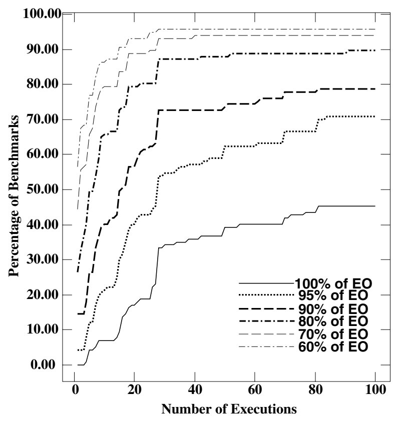
Finally, Figure 1(f) shows that MOD-CM is superior to EXEC-CS for 20 executions and reaches about the same levels of optimization for 40 executions. Comparing the trade-off graphs from Figures 1(b) and 1(f), we see that after 20 executions using the MOD-CM strategy, we reach the same level of optimization as EXEC-CS does after 30 to 40 executions. Comparing Figure 1(f) to Figure 1(a), we see that these levels of optimization are reached in the Execution-Only approach after about 70 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 of the exact hit rate, the working set size constraint is highly effective.



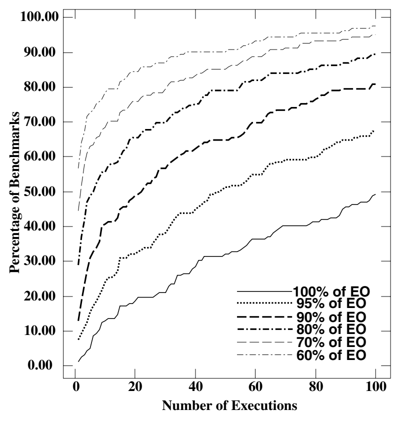
(a) Execution-Only



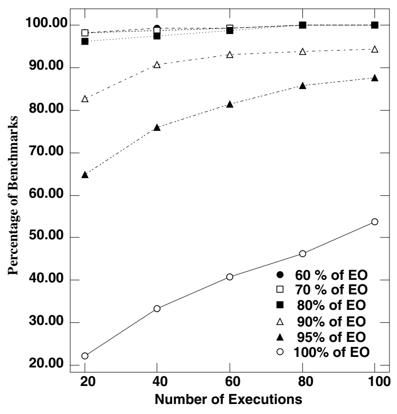
(b) EXEC-CS



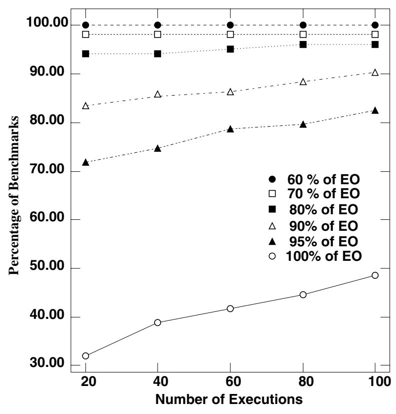
(c) EXEC-CM



(d) MOD-CS1



(e) MOD-CS2



(f) MOD-CM

Fig. 1. Trade-off graphs

4 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, profiling a number of programs with highest hit rates according to the simulator (strategy MOD-CS1) is actually the worst strategy. This shows that a static L1 cache simulator ignores many issues that are crucial for performance. It also follows that a strategy that only uses static knowledge obtained from a highly accurate cache model will be outperformed by iterative strategies that also use profiling information.

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, and 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

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.

5 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 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. 255
2. J. Bilmes, K. Asanović, C. W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proc. ICS*, pages 340–347, 1997. 255
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. 255
4. S. Carr. Combining optimization for cache and instruction level parallelism. In *Proc. PACT*, pages 238–247, 1996. 255
5. S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. PLDI*, pages 279–290, 1995. 255, 256
6. P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. PLDI*, pages 71–84, 1997. 255
7. 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. 254, 257, 258
8. 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. 255
9. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proc. Conf. on Machine Learning*, 2000. 255
10. M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proc. Dynamo*, 2000. 255
11. M. J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proc. ICPP*, 2000. 255
12. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998. 255
13. 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. 255