Pre- and Post Selection of Compiler Optimizations by Program Execution

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 this paper, we investigate the combined use of static techniques and dynamic feedback information to achieve a high level of optimization of both compiler efficiency and code performance. In previous work, we have introduced a new compiler approach, *iterative compilation*, to select the best tile sizes and unrolling factors. In this approach, many versions of programs are generated and their worth is determined by the actual execution time. We found that this approach can obtain a much higher level of optimization than conventional static techniques. To further achieve both compiler efficiency and high code performance, we have incorporated cache models in the iterative compilation system. We utilize feedback information in two ways: Post-Selection and Pre-Selection. Experimental results show that even a very simple cache model can achieve a high level of optimization with Post-Selection verifying the worth of transformed programs, especially when the number of program executions is limited. Pre-Selection is superior if more program executions are allowed.

1. INTRODUCTION

One of the most important tasks for optimizing compilers is to transform program structures written by a programmer into more efficient ones for a specific target platform. Such program transformations include loop tiling and loop unrolling. These transformations have a significant impact on program performance by increasing locality and exposing instruction-level parallelism, respectively. There are many static algorithms [8, 17, 6] to select the best tile sizes based on a simplified cache model that have obtained good speedups. However, if these transformations are applied together, finding the optimal combination becomes a very complex issue since these two transformations are highly inter-dependent and preferred transformations vary widely across different platforms [3, 15]. If simplified machine models or limited levels of information (such as L1 cache configuration) are used, they fail to model such complex transformation spaces. Overcoming this failure is crucial in situations, such as embedded systems, where code performance is critical and hardware specific optimizations are required. Currently, such optimization is obtained by hand coding in assembly or special-purpose digital signal processing (DSP) compilers. Unfortunately, due the rate of architectural change and code growth of embedded applications, it is becoming very difficult to implement a hardware specific DSP compiler for every change and to optimize large applications on the assembly level. We therefore need to consider adaptive compilers, i.e., those that are able to cope with a changing hardware platform and produce highly optimized code for a specific architecture.

Clearly, no compiler technology based on static analysis only and a hardwired cost model of the target processor is capable of changing to a new architecture while achieving a high level of optimization. What is required is a method where the compiler can receive dynamic feedback regarding its performance and modify its behavior accordingly.

To achieve this goal we have introduced a new compilation approach, *iterative compilation* [13]. In this paper, we incorporate cache models into the iterative compilation system to achieve a high level of optimization of both compiler efficiency and code performance. In this approach, many versions of program with different tile sizes and unrolling factors are generated and their value is determined by actual execution time. Cache models are used to guide the search by selecting good candidates or rejecting bad candidates. Such an approach does not suffer from compile-time undecidability or inaccuracy of static models and with sufficient time it produces highly optimized code for a specific machine taking into account the behavior of the entire machine. This technique is especially suitable for embedded systems where long compilation time can be amortized across the number of products and the lifetime of the applications.

The paper is organized as follows. In Section 2 related and previous work is discussed. In Section 3 we briefly describe the implementation of iterative compilation. Section 4 describes the experimental setup and provides detail of the

search strategies used in iterative compilation. Section 5 presents the experimental results that demonstrate the improved effectiveness of the search due to the inclusion of cache models. In Section 6 we discuss the results and future directions of research.

2. RELATED AND PREVIOUS WORK

There are many papers that deal with tile size selection using a simple cache model [8, 17, 6]. Using these techniques good speedups have been obtained for architectures where the L1 cache is a dominant factor in program performance. In [19] the impact of interactions among multiple levels of memory or parallelism are investigated. Their results suggest that simple cost functions guide only simple situations and with architectural growth single-level cost functions may not optimally guide tile size selection. Recently, program optimizations based on searching have received attention from other authors. Whaley and Dongarra [22], and Bilmes et al. [2] describe systems for generating highly optimized BLAS routines. These systems probe the underlying hardware to find optimal values for blocking factors, unrolling factors etc. Experimentation has shown that these systems are capable of producing more efficient codes than the vendor supplied, hand optimized library BLAS routines. Wolf, Maydan and Chen [23] have described a compiler that also searches for the optimal optimization by considering the entire optimization space. Unlike our approach, their compiler uses static cost models to evaluate various optimizations. Chow and Wu [7] apply 'fractional factorial design' to decide on a number of experiments to run in order to select a collection of compiler switches. They, however, focus on on/off switches and do not consider the choice of parameter values that might have a large range of values. Michael and Rudolf [21] propose a dynamic program optimization system, called ADAPT. As our approach does, many versions of the program are tested and the fastest version is selected. Unlike our approach, code generation and program execution are overlapped and long execution time is required to reach the optimal optimizations. This approach is therefore suitable only for programs with very long execution times. There are several other approaches that adaptively change program behavior during its execution [9, 12]. These approaches cannot be used for embedded systems since programs are stored in Read-Only Memory and the original codes cannot be replaced by the optimized codes. Moreover, in these adaptive approaches, code duplication is applied, resulting in a code explosion that should be avoided in embedded systems where code size has a significant impact on program performance (small code size reduces the frequency of overlays and therefore can vastly improve execution time [18]).

Carr and Kennedy [5] and Carr and Guan [4] compute unroll-and-jam factors in order to minimize the difference in machine and loop balance. However, in [5] it is assumed that all memory references are cache hits, an assumption that is clearly not valid and will degrade the effect of the transformation. Carr and Guan [4] also use a search strategy to decide the best unroll factor, like the present paper. In contrast, our approach uses actual execution times as well as model information to decide upon the optimal unrolling factor and loop tiling sizes.

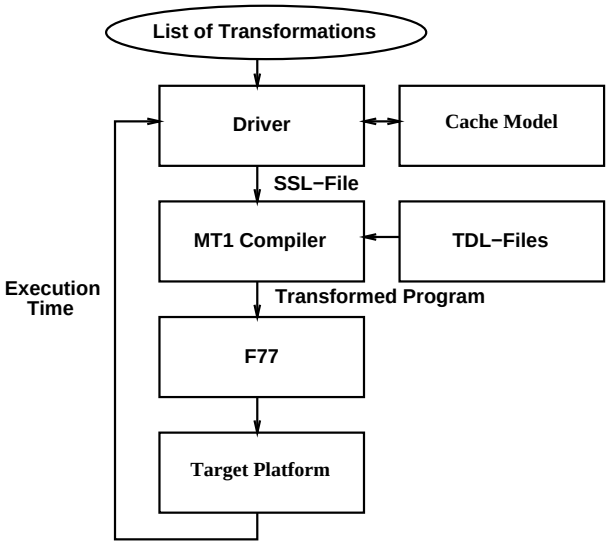

Figure 1: The Compilation Process

In previous work [13] we have shown that iterative compilation is powerful approach capable of outperforming existing tile size selection algorithms and reaching a high level of optimization with reasonable compilation time. We have compared the obtained speedups relative to static tile size selection algorithms: the TSS algorithm [8] and LRW proposed in [17]. In more than 99% of the cases we have considered, iterative compilation outperforms these static techniques. This level of optimization is obtained within reasonable compilation time. On average 100 iterations take less than 12.5 minutes [13] for the benchmarks considered.

3. COMPILATION SYSTEM

Figure 1 shows an overview of the compiler system. The compilation system is centered around a global driver. This driver keeps track of the transformations evaluated and decides which transformations to apply next. The driver reads a list of transformations that need the range of parameters.

The global driver invokes a source-to-source compiler MT1 [1]. MT1 has two mechanisms to control the transformation that is applied to the input program: a Transformation Definition Language (TDL) [1] and a Strategy Specification Language (SSL) [1]. For each transformation included in the list of transformations, the transformation is specified based on a pattern matching in the TDL-files. Next, in order to instruct MT1 to apply a specific sequence of transformations, the global driver constructs an SSL-file that specifies the order of transformations. After a predetermined number of iterations, the global driver stops searching and outputs the transformed program with the shortest execution time.

We have implemented several search algorithms, including a genetic algorithm, simulated annealing, pyramid search, window search and random search. However in the current case, the search space is rather small, 2000 variations of

transformations (see Section 4), and all search algorithms find the same level of optimization within a very small margin. Thus we use only the random search in this paper.

4. EXPERIMENTAL SETUP

In this section, we explain how cache models are used to reduce the number of program executions without degrading a level of optimization. We consider two cache models, an accurate but expensive cache simulator and a less accurate but cheap simple cache model. The cache simulator considers the L1 cache of platforms we have selected. All memory accesses are assumed to be issued to the cache although some memory accesses can be held in registers. The simulator is used to estimate the cache hit rate of the transformed program and the simple cache model estimates the working set size to allow prediction of whether or not a transformed program fits in the cache. These models are combined with feedback information (i.e., execution time in the current case) in two ways: Post-Selection and Pre-Selection.

4.1 Post-Selection

The first step of Post-Selection, is to estimate the cache hit rates and working set sizes of all combinations of tile sizes and unrolling factors in the parameter space. These combinations are then ordered based on a particular ranking strategy and the highest ranked 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 level 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 following 3 strategies using Post-Selection.

- **Post-SIM1**

```
calculate all cache hit rates
current = initial transformation
REPEAT
      next = H_next_highest
      execute(next)
      IF exec_time(next) < exec_time(current)
      THEN current = next
```

First calculate the cache hit rate of all combinations of tile sizes and unrolling factors using the cache simulator. The selection of combinations is solely based on the cache hit rate and combinations with highest cache hit rate ($H_{next\_highest}$: I-th best at iteration I) are selected and executed. The N combinations with the highest cache hit rates are executed and compared. The one with fastest execution time is selected.

- **Post-SIM2**

```
calculate all cache hit rates
current = initial transformation
FOREACH Unroll Factor
      next = H(unroll)_next_highest
      execute(next)
      IF exec_time(next) < exec_time(current)
      THEN current = next
```

In this strategy, we consider the impact of loop unrolling more explicitly. Currently we do not have a static model to determine the best unrolling factor, therefore, we take the tile size that yields the highest cache hit rate for each unrolling factor. First calculate the cache hit rate of all combinations of tile sizes and unrolling factors using the cache simulator. Then for each unroll factor, the combination with the highest cache hit rate is selected and executed. This foreach loop is repeated until N combinations are totally executed. For example, if the unrolling factors are from 1 to 20, the combinations with $N/20$ highest cache hit rates are selected for each unrolling factor and their execution times evaluated. The fastest combination is selected.

- **Post-MOD**

```
current = initial transformation
FOREACH Unroll Factor
      next = WS(unroll)_next_largest ≤ CS × β
      execute(next)
      IF exec_time(next) < exec_time(current)
      THEN current = next
```

In this strategy, we select the largest tile size such that the working set, WS, is within β % of the cache size CS for each unrolling factor. In total, N combinations are executed. For each unroll factor (from 1 to 20), $N/20$ combinations with largest tile size are selected. Then the fastest combination is selected. The value of β can be seen as *effective cache size*.

4.2 Pre-Selection

In Pre-Selection, the random search is used to select the combinations from the parameter space. However, a combination is only executed if the cache models predict a specified level of performance. Combinations of tile sizes and unrolling factors are selected randomly and combinations with poor cache hit rates or improper working set sizes are filtered out reducing the number of program executions. In contrast to Post-Selection, the selection of combinations is done by the search algorithm and the value of transformed program is verified during the search checking whether reduction in execution time is made. We consider the following 2 strategies.

- **Pre-SIM**

```
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, a cache simulator is used to guide the search. One combination is selected randomly, and its execution time is evaluated if the cache hit rate (H_{next}) is within a α% of the current best cache hit rate ($H_{current}$). The search algorithm will stop after N combinations are actually executed or cache hit rates of all combinations are examined. The combination with the smallest execution time is selected.

- **Pre-MOD**

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

In this case, the search space is restricted so that the search algorithm can find good combinations with high probability. Each value of γ and δ determines the lower and upper bound of the working set size. A combination is selected randomly, and its execution time evaluated if its predicted working set size (WS_{next}) is between these boundaries. After executing N combinations or examining all combinations in the parameter space, the search algorithm stops and reports the fastest version.

With these strategies bad combinations with poor cache hit rates or inadequate working set sizes are excluded. We consider $N = 20, 40, 60, 80$ and 100 for each strategy.

4.3 Benchmarks and platforms

We have selected the following benchmarks in our experiments. In order to see the effect of the transformations on different memory access patterns, we use different loop orderings in the same benchmark. 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 orderings in matrix-vector multiplication on data input sizes 2048, 2300 and 2301. We use 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 involves multiplication with a transposed matrix. We use data input sizes of 256, 300 and 301. Finally, we use a Finite Impulse Response filter (FIR) with data sizes of 8192, 8300 and 8301.

We have conducted our experiments on four different platforms: Pentium III, Pentium II, Hewlett-Packard Precision Architecture (HP-PA 7100) and UltraSparc. Note that the platforms we currently use are not representative for DSP, however, we believe that the techniques discussed will prove applicable on DSP platforms. The L1 data cache configurations are shown in Table 1. In total we have collected 162 measurements to produce statistically relevant results.

	Cache Size	Associativity	Line Size
Pentium II & III	16KB	4-Way	32 Byte
HP-PA	128KB	Direct Mapped	32 Byte
UltraSparc	16KB	Direct Mapped	32 Byte

Table 1: L1 Data Cache Configurations

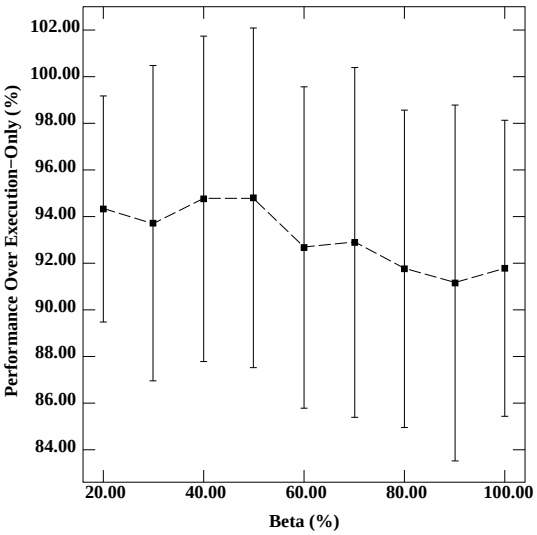

Figure 2: Performance of Post-MOD for 20 Executions

A Fortran compiler with full optimization is used to compile both the original and transformed programs. Unrolling factors from 1 to 20 and loop tiling with tile sizes from 1 to 100 were used in these experiments.

In the following, we compare with the result from iterative compilation executing 400 combinations without a cache model, and not with the best speedups that can be achieved using the present transformations. The search algorithm described in [13] that only uses execution time is called the Execution-Only Algorithm. We do this because in real situations, Execution-Only Algorithm represents the best possible estimate of execution time for each combination. Moreover, for a restricted set of benchmarks, we have generated the entire search space and we found that on average the result from Execution-Only Algorithm is within 2% of the best possible speedup using the present transformations.

5. RESULTS

First we show the experimental results from which we can determine the best parameters for each cache model,α, β, γ and δ. The level of optimization that can be obtained with cache models is then discussed.

5.1 Determining Parameters

Figure 2 shows how close Post-MOD approaches the speedup obtained by Execution-Only Algorithm as a function of β

Strategy	20	40	60	80	100
Pre-SIM	145	326	436	523	604
Pre-MOD	650	1183	1420	1800	2000

Table 2: Total Numbers of Steps Considered at Each Execution

on average with standard deviations. In this experiment only 20 combinations are executed in Post-MOD ($N = 20$) while 400 combinations are executed in Execution-Only Algorithm.

To determine the value of α we have done detailed experiments in [14]. We found that the average speedup is slightly increasing with α until α reaches 99.9%. For $\alpha > 100\%$, however, it drops very sharply. If α is set more than 100%, Pre-SIM is too selective excluding most of the search space, actually executing very few combinations. We fine tuned the value of α and found that $\alpha = 99.9\%$ performs best.

Second, we find that *effective cache size* is very small ranging from 40% to 50% as shown in Figure 2. We select $\beta = 50\%$ instead of $\beta = 40\%$ since it finds more benchmarks that reach to 100% of the maximum speedup of Execution-Only Algorithm. Experiments also reveal that the combination of $\gamma = 40\%$ and $\delta = 50\%$ is optimal.

5.2 Analysis: Levels of Optimization

Figure 3 shows how close each strategy approaches the speedup obtained by Execution-Only Algorithm for which 400 combinations are actually executed. Average performance of all benchmarks is plotted with standard deviation. Figure 4 shows detailed performance for each strategy. The x-axis shows the number of program executions while the y-axis shows percentages of benchmarks that reaches to certain levels of optimization (from 80% to 100% of Execution-Only Algorithm). From these graphs we can deduce how many program executions we need for each strategy if we require that at least n% of the benchmarks reach at least m% of the maximal improvement. For instance in Figure 4 (d) we can see that with 100 program executions Pre-SIM yields 65% of total benchmarks ($162 \times 0.65 = 105.3$ benchmarks) reaching 100% of speedup obtained by Execution-Only Algorithm.

Firstly, considering the difference between Pre-Selection and Post-Selection, Post-Selection is better when cost models capture the dominant factor of performance. However, if the cost models fail to do so, Pre-Selection is the better approach. Since Pre-Selection models visit the search space according to the search algorithm, they visit a wider range of the search space and find better combinations that cannot be predicted by static models. Considering Pre-SIM, we find that Pre-SIM is the most effective strategy except at 20 executions where Post-MOD is best (see Figure 4 (d)). Pre-SIM is more flexible than Post-SIM1 and Post-SIM2, visiting a large area of the search space based on the Execution-Only Algorithm while effectively rejecting bad combinations that exhibit poor cache hit rates. Also, in the simple cache models, Pre-MOD yields better performance than Post-MOD does when more than 40 combinations are executed (see Figure 3). Considering that Pre-Selection yields better performance than Post-Selection, it is worthwhile to have some randomness for guiding the search when the information to decide the best sequence of transformations is limited or inaccurate. In Post-Selection other search algorithms such as GA and SA that decide next combination based on the execution time can be also used.

Secondly, comparing Post-SIM1 and Post-SIM2, we find that Post-SIM2 yields better performance than Post-SIM1, especially if the number of executions is limited. As shown in Figure 3 Post-SIM2 achieves 94.9% of Execution-Only Algorithm with 20 executions, while Post-SIM1 needs more than 40 executions (93.8% with 40 executions) to find a high level of optimization. As widely regarded, the hit rate of the L1 cache is one of the most important factors for the efficiency of the resulting code. However, in the current case, it is apparently not a good approach to estimate the resulting code performance only from L1 cache hit rate. One important factor that Post-SIM2 takes into account and Post-SIM1 does not is the presence of loop unrolling. Post-SIM2 knows that loop unrolling is also applied and that the range of unroll factor is from 1 to 20. It selects combinations that have the highest cache hit rate for each unroll factor, i.e., the selection strategy has the flavor of a directed search based on the knowledge that unrolling is used. The costs of simulation are the same for both cases, however, this extra information provided to Post-SIM2 significantly improves the ability to find good combinations.

Thirdly, we find that Post-MOD and Pre-MOD are quite good models to guide the search. Especially when the number of program executions is limited, Post-MOD performs best among all strategies at 20 executions. Although Post-MOD and Pre-MOD are not the best after 20 program executions, they always perform better than Post-SIM1 on average. One reason the accurate cache model does not perform so well is that it fails to take into account of the impact of loop unrolling and other levels of memory hierarchy, such as the L2 cache. Another reason is 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. For a real processor, a burst of misses will stall the machine for a significant number of cycles whereas occasional misses may be hidden. From Figure 4 we see that Post-SIM1 is the worst strategy that requires a large number of executions. It is shown that the accurate cache simulator even fails to predict the resulting code that reaches 80% of Execution-Only Algorithm if the number of execution is limited (in Figure 4 (a)). It requires 100 program executions to be comparable to other strategies.

Finally, in table 2, the total numbers of steps in Pre-Selection are shown. In Pre-Selection many combinations are visited according to the search algorithm for which the execution time is not evaluated. For example, in Pre-SIM 145 combinations are considered and only 20 combinations with high cache hit rate are actually executed. On average 85.6% and 95.6% of total steps are filtered out in Pre-SIM and Pre-MOD, respectively. This results suggest the cost of total compilation process. In Pre-Selection, hit rates or working

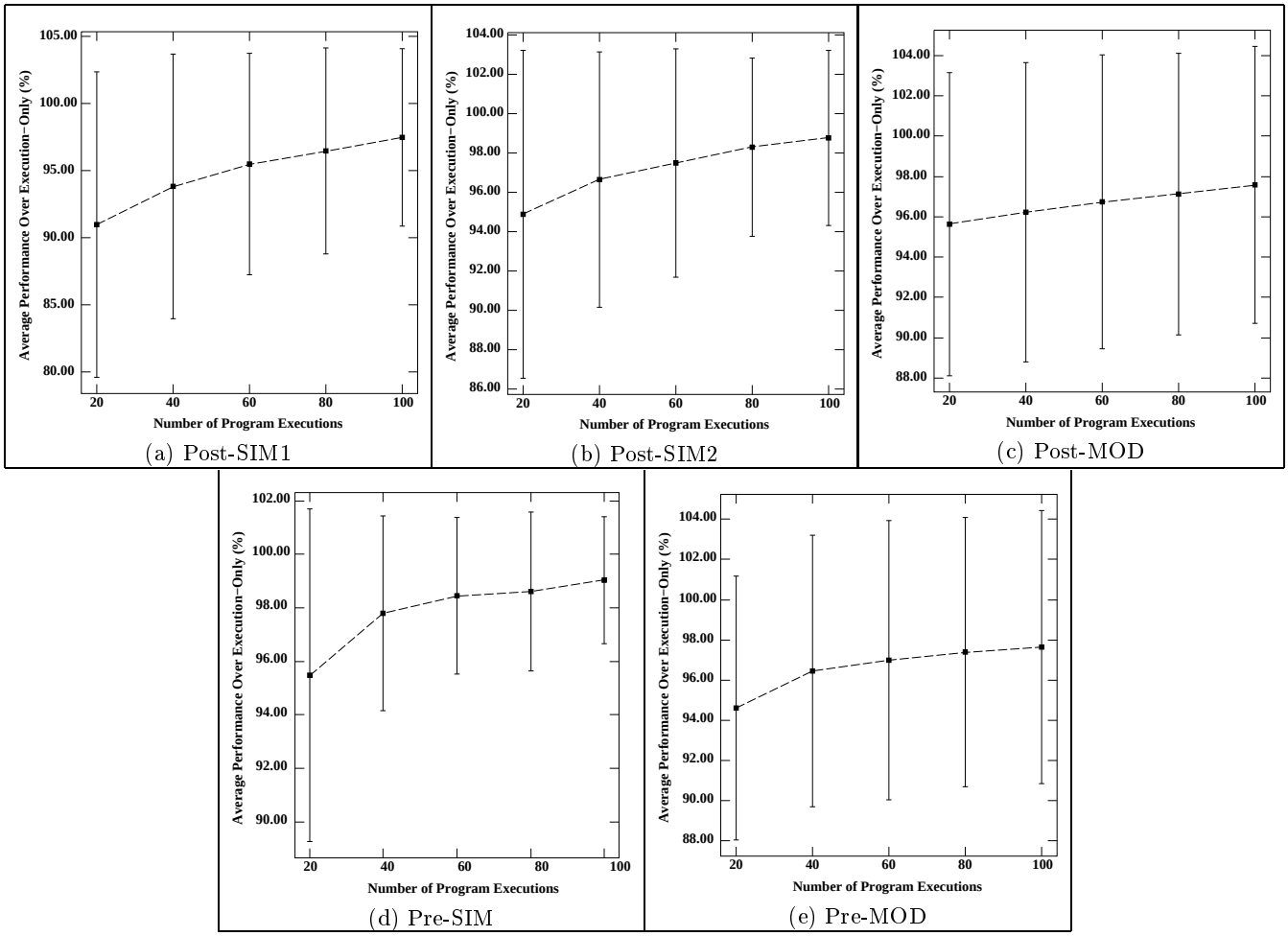

Figure 3: Average Performance over Execution-Only Algorithm

set sizes of all combinations need to be calculated beforehand. Thus the cost spent for the cache models is fixed regardless to the number of program executions and therefore Pre-Selection is costlier than Post-Selection.

6. DISCUSSION

In the case of embedded and DSP applications, there are usually small routines that contain one or more compute intensive loops. Hence aggressively optimizing these routines can reduce the overall execution time of the application significantly. These routines are prime candidates for being tackled by the proposed approach. Furthermore, the compiler can keep track of how well it is able to optimize certain parts of the code and switch to an iterative approach in case it is unable to find good optimizations statically. Currently, we use only execution time as a metric for performance. However, other factors such as code size, which is very important for embedded systems, can be considered also.

In this paper, we use fixed input data sizes. If profiling shows that the kernel is heavily biased towards a certain input data size, this approach is appropriate. However, in many cases profiling will yield a distribution of input data sizes. In this case we cannot simply optimize for one single data input size. In such a case we need to optimize the program so that the average execution time is minimized [16]. We have found that there are many combinations of transformations that yield good speedups and iterative compilation finds effective combinations for a range of input data sizes. Other optimizations, such as array padding, can be used to produce a code that is less sensitive to data sizes. Certain data sizes, often powers of two, may cause a significant number of cache conflict misses. Array padding that reduces conflict misses can be easily included in our search space.

We use $\alpha = 99.9\%$ for Pre-SIM. Although there is some randomness in this strategy, the resulting performance is still dominated by the behavior of the L1 cache in the case of general purpose processors that we have considered. If other architectures, such as DSPs in which other factors also have a significant impact on resulting code because of their complicated memory hierarchy, are considered, a different value of α might be used. In such a case, a lower value must be used to visit the search space more freely so that we can find the best sequence of transformations that is not predictable only from the information of the L1 cache.

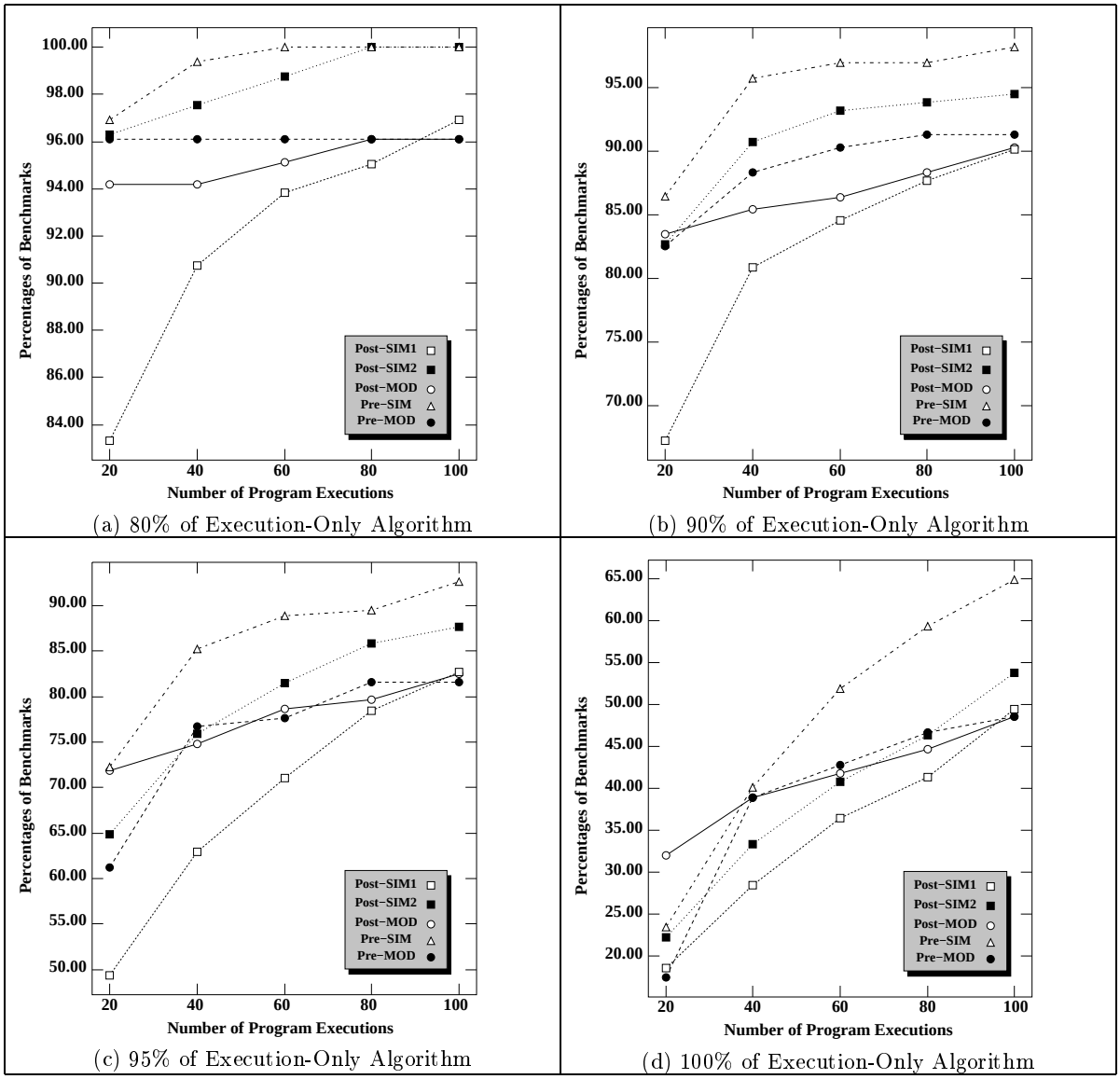

Figure 4: Percentages of Benchmarks for Each Cache Model

Note also that we use a full cache simulator to measure the impact of the transformations at compile time. As our results show, it is very powerful to find the best combination of transformations within a few executions. However, having a full cache simulator is very expensive and not suitable for our approach. It is in fact more expensive to simulate many of these benchmarks than execute them. The cache simulator was used in this study to identify the limits to filtering based on a very accurate cache model versus a simpler and faster one. We are considering other modeling techniques, such as Cache Miss Equations [11] or similar techniques proposed in [20, 10] that yield very accurate cache miss rate with much lower cost. Using these techniques we believe that the cost of Post-SIM1, Post-SIM2 and Pre-SIM will be as low as that of Post-MOD and Pre-MOD. We intend to quantify the amount of time saved relative to Execution-Only Algorithm using these strategies. We are also currently investigating an analytical model from which we can obtain the best unrolling factor.

Finally, the search should be steered by available knowledge, both application domain and target domain specific. We need to collect profiling information that can be exploited using his knowledge and determine its effectiveness in improving the search efficiency.

7. CONCLUSION

To achieve both compiler efficiency and code performance, we have investigated the incorporation of static cache models into iterative compilation. We found that iterative compilation can be more effective with the help of cache models by reducing the required number of executions while achieving

a high level of optimization. According to our results, multiple levels of information should be considered to guide the search more effectively. When the number of program executions is limited, Post-Selection is more effective verifying the transformed program that is predicted by static models. If more program executions are allowed, Pre-Selection is superior, visiting wider range of the search space while excluding bad candidates. Iterative compilation appears to be especially promising for embedded systems where highly machine specific optimizations are required to yield a good code performance.

8. REFERENCES

[1] A.J.C Bik, P.J. Brinkhaus, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Transformation Mechanisms in MT1. Technical Report no. 1999-21, LIACS, Leiden University, 1999.

[2] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. *Proc. ICS'97*, pages 340–347, 1997.

[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. *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998. Workshop organised in conjunction with PACT'98.

[4] S. Carr and Y. Guan. Unrolling-and-Jam Using Uniformly Generated Sets. In *Proc. MICRO-30*, pages 349–357, 1997.

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

[6] J. Chame and S. Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proc. the 13th ACM International Conference on Supercomputing*, pages 492–499, 1999.

[7] K. Chow and Y. Wu. Feedback-Directed Selection and Characterization of Compiler Optimizations. In *Proc. 2nd Workshop on Feedback-Directed Optimization*, November 1999. In conjunction with MICRO-32.

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

[9] P. Diniz and M. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. *Proc. Programming Languages Design and Implementation*, pages 71–84, 1997.

[10] B.B. Fraguela, R. Doallo, and E.L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 221–231, October 1999.

[11] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703–746, July 1999.

[12] R. Gupta and R. Bodik. Adaptive Loop Transformations for Scientific Programs. *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, October 1995.

[13] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. To appeare in Proc. PACT2000.

[14] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. Incorporating cache models in iterative compilation for combined tiling and unrolling. Technical Report no. 2000-10, LIACS, Leiden University, 2000.

[15] 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 Note in Computer Science, pages 121–132, May 1999.

[16] 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.

[17] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *ASPLOS IV*, pages 63–74, April 1991.

[18] C. Lefurgy and T. Mudge. Code compression for DSP. Technical Report CSE-TR-380-98, University of Michigan, November 1998. Presented at CASES-98 Workshop.

[19] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Qantifying the Multi-Level Nature of Tiling Interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

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

[21] M.J. Voss and R. Eigenmann. ADAPT: Automated De-Coupled Adaptive Program Transformation. In *Proc. of the International Conference on Parallel Processing*, August 2000.

[22] R.C. Whaley and J.J. Dongarra. Automatically Tuned Linear Algebra Software. *Available through* `http://www.netlib.org/atlas/`, 1998.

[23] 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.