Compile-time Based Performance Prediction*

Calin Cascaval, Luiz DeRose, David A. Padua, and Daniel A. Reed

Department of Computer Science
University of Illinois at Urbana-Champaign
{cascaval,derose,padua,reed}@cs.uiuc.edu

Abstract. In this paper we present results we obtained using a compiler to predict performance of scientific codes. The compiler, Polaris [3], is both the primary tool for estimating the performance of a range of codes, and the beneficiary of the results obtained from predicting the program behavior at compile time. We show that a simple compile-time model, augmented with profiling data obtained using very light instrumentation, can be accurate within 20% (on average) of the measured performance for codes using both dense and sparse computational methods.

1 Introduction

In this paper we present the compiler-related part of the Delphi project whose goal is to predict performance of codes when executed on multiprocessor and distributed platforms. The project will integrate several components, such as compilers, resource managers, dynamic instrumentation tools, and performance prediction and visualization tools, into a unique environment targeted for high performance computing.

The part of the project presented in this paper focuses on compiler techniques for performance prediction of parallel codes and machines. By using these techniques, a compiler becomes a useful tool to support machine design and program tuning either by itself or as part of performance visualization tools, such as SvPablo [8]. Furthermore, accurate compile-time performance prediction techniques are of crucial importance for the compiler itself to select the best possible transformations during the code optimization pass.

Powerful techniques for performance prediction are quite important due to the difficulties programmers and compilers encounter in achieving peak performance on high-end machines. The complexity is exacerbated in new architectures, which have multi-level memory hierarchies and exploit speculative execution, making the achievement of a large fraction of the peak performance of these systems even harder.

* This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

One of the main challenges we face is to find a general methodology that allows the performance prediction of arbitrary codes, including irregular applications, for any arbitrary input data. In this paper, we describe a simple strategy for performance prediction that we have implemented in the Polaris source-to-source translator [3]. The strategy leads to symbolic expressions generated by the compiler for each loop of the program being analyzed. These expressions integrate the prediction of CPU execution time and the prediction of cache behavior and can make use of both compile-time and profiling information.

Examples of optimizing transformations that should be controlled by performance prediction results are: parallelization of loops based on the tradeoff between the amount of work in the loop and the overhead of running the loop in parallel; loop interchange, a classical loop optimization that affects data locality and parallelism overhead in non-obvious ways. Even a gross estimate of the performance of a program segment can enable the compiler to decide if the optimizing transformation is beneficial.

Our work is performed at the high level language representation, which allows us to keep the models simple and relatively hardware independent. We have validated our performance prediction models on sequential programs using the SPEC95 benchmarks and some sparse matrix codes on the SGI's R10000 and Sun's UltraSparc IIi processors. It was quite surprising to see how well our simple strategy worked for our test cases.

The remainder of this paper is organized as follows. In Sect. 2 we describe our compiler-driven prediction model. In Sect. 3 we present the experimental results. In Sect. 4 we discuss related work, and we conclude in Sect. 5.

2 Compiler Driven Prediction

In the approach described in this paper, the compiler extracts information from the source code and generates symbolic expressions representing the execution time of different program segments. For the cases where we do not have compile-time information, either run-time profiling [6] provides the necessary data or simple approximations [1] can be used.

The target computer can be an existing machine or a proposed architecture. Although for many users the total execution time will be the main figure of interest, we believe that other values, such as number of cache misses, can be used profitably to drive compiler transformations. Along the same lines, for architectures with multiple heterogeneous processors, such as FlexRAM [16], compile-time performance prediction can be used to decide where to execute the code: in the main processor which has a larger, slower cache and less bandwidth to the main memory, or in the Processor-In-Memory (PIM) which has a smaller but faster cache and it is implemented in the memory chip.

In Fig. 1, we present an overview of the prediction environment. In this environment, the compiler analyzes the source code and synthesizes the symbolic expressions representing performance data from the application. It also instruments the code to extract profiling information. The code can be run with differ-

ent data sets to extract the profiling data used as parameters for the performance expressions. The compiler uses the profiling data to evaluate the performance expressions – although, symbolic expressions comparisons are also possible, which in turn will be used to control program optimizations. In our environment, since Polaris is a source-to-source parallelizing compiler, the optimized program will be an optimized and parallelized Fortran source. Also, the profiling data can be used along with hardware costs to resolve the symbolic expressions and the results displayed using a performance visualization tool.

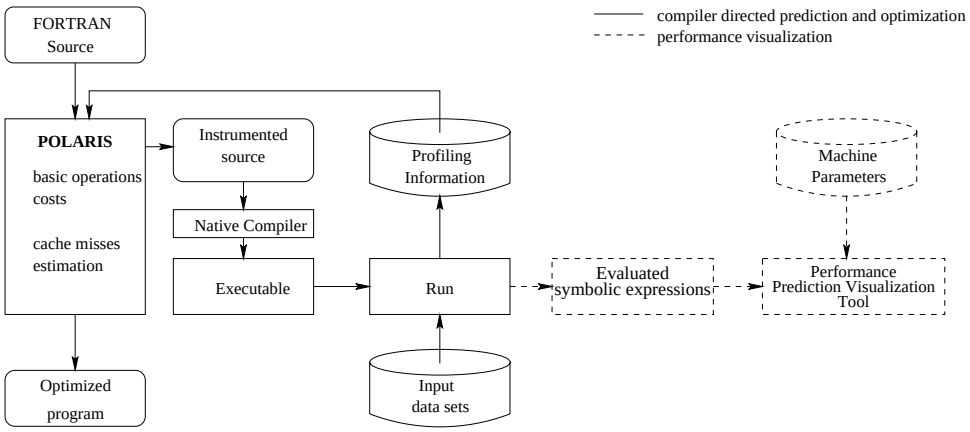


Fig. 1. Compiler-based prediction environment

We present first the machine model supported by our performance prediction model, then the targeted application domain and, finally, the prediction model itself.

2.1 Models

Machine Model The architecture supported by our prediction model assumes one or more CPUs that execute code stored in memory. The processor could be a simple in-order single issue architecture or a more complex, multiple issue out-of-order processor. The memory consists of a hierarchy of memory levels (caches), each having different characteristics.

Program Model The type of applications to be evaluated in this project are scientific Fortran programs. The codes used in our experiments include dense matrix computations from the SPECfp95 benchmark suite [18], and sparse matrix codes from SpLib, a library of sparse functions for linear algebra [4].

Since we want to handle a wide range of codes, our compiler must be able to handle programs with complex control flow, such as imperfectly nested loops,

loops with early exits and conditionals, and while loops. We are willing to trade some prediction accuracy for the ability to handle efficiently any program structure. The approach that we have taken, detailed in the following section, is to have several models that we can apply depending on the amount of compile-time information available. In case we do not have enough compile-time information, run-time profiling can be used to obtain the necessary data.

Performance Prediction Model The performance prediction model uses analytical and experimental information. The compiler synthesizes performance expressions and instruments the code to extract values unknown at compile-time. The total execution time is decomposed as follows:

$$T_{total} = T_{CPU} + T_{MEM} + T_{COMM} + T_{I/O} \tag{1}$$

where T_{CPU} is the computation time spent by the processor itself, T_{MEM} is the time spent accessing the memory hierarchy, T_{COMM} is the interprocess/thread communication time, and $T_{I/O}$ is the time spent doing I/O. We consider (1) to be a reasonable decomposition that allows us to concentrate on each subsystem (CPU, memory, I/O) and work with simpler models for each.

We shall detail the how we estimate T_{CPU} and T_{MEM} separately after we describe some of the common features. Each term in (1) consists of a symbolic expression, i.e., a mathematical formula expressed in terms of program input values and perhaps some profiling information, such as branch frequencies. The expressions also involve parameters representing characteristics of the target machine and are a function of the source code and input data. To factor in the data sets, the compiler will place very light instrumentation in the program, and the user or the prediction environment will run the program such that it can collect information on the input data, and substitute it in the symbolic expressions.

For example, the loop in Fig. 2 performs a sparse matrix vector multiplication ($\boldsymbol{Y} = \boldsymbol{A} \times \boldsymbol{X}$) and the matrix $\boldsymbol{A}$ is stored in compressed sparse row storage (row i is stored in consecutive locations of $\boldsymbol{A}$ beginning at $ia(i)$ and the column of the element stored in $\boldsymbol{A}(k)$ is $ja(k)$). Since the number of iterations of loop L2 depends on the input data, the compiler will not be able to estimate how many times statement S2 is executed without information on the input data. To obtain this information, the compiler will add instrumentation after loop L2 to compute the number of iterations, place a symbolic variable for the number of iterations in the prediction formula, and let the dynamic phase compute the actual value that depends on the data set. This interaction is illustrated by the path marked with solid arrows in Fig. 1.

2.2 Compile-time Prediction

In the static phase of the prediction, the compiler goes through the Abstract Syntax Tree (AST) of the program and collects information about operations. Recall that all our evaluations are done at source code level. The compiler combines

```
L1          do i = 1, m
S1              Y(i) = 0.0d0
L2              do k = ia(i), ia(i+1)-1
S2                  Y(i) = Y(i) + A(k) * X(ja(k))
                end do
            end do
```

Fig. 2. Sparse matrix vector multiplication

this information in symbolic expressions for statements, loops, and routines, depending on the level at which we want to predict performance. The compiler also places instrumentation statements for the dynamic phase wherever it does not have enough information, such as loop bounds, branch frequencies, etc.

Basic Operations The sub-model for T_{CPU} in (1) estimates the time spent by the processor doing computation. It counts the number of operations executing in the CPU's functional units, including load and store operations assuming no cache misses. In addition, it considers as basic operations intrinsic functions, such as $SQRT$ (many current processors have functional units that execute square root operations) or trigonometric functions, as well as function calls and loop overheads.

To reduce the number of independent variables in the symbolic expressions, operations may be grouped together based on the operation type and the data size on which they operate. For example, we group together single precision addition and multiplication since, on most current architectures, these instructions have similar latencies being executed in the same or identical functional units. We distinguish between multiplication and division since the division operation usually has longer latency than other operations. However, the grouping is not fixed so that we can accommodate other architectures with different designs. T_{CPU} can be expressed as:

$$T_{CPU} = CycleTime \times \sum_{i}^{\#groups} (k_i \times C_i), \tag{2}$$

where k_i are symbolic expressions representing the number of operations in group i, and C_i represents the hardware cost for the operations in group i. The hardware costs C_i can be obtained either from the processor's manual and design specifications, or by using microbenchmarking [21]. The latter is usually the most convenient way to get the values associated with intrinsic functions and loop overheads.

Using simple symbolic arithmetic, these expressions are combined to generate the cost of operation in each statement. The expressions corresponding to the statements are then combined and augmented with coefficients obtained from the dynamic phase (e.g., loop bounds and branch frequencies), to produce cost

expressions for several levels of granularity in the program (blocks of statements, loops, procedures) until a unique expression is generated for the entire program.

Although this is a very simple strategy, it has proven reasonably accurate when no compiler optimizations are applied, as can be seen in the experimental results presented in Sect. 3. Dependence graph information could be used to improve prediction accuracy on modern processors that exploit instruction level parallelism (ILP).

In order to accurately predict the performance for optimized codes we have to apply, or at least approximate, the optimizations performed by the native compiler in our restructurer. We have chosen to approximate these optimizations by using the following heuristics applied at high level source code:

- *Eliminate loop invariants.* This is a simple optimization applied by all optimizing compilers and it can be done at high level.
- *Consider only the floating point operations.* This is based on the observation that, in scientific codes, the useful computation is done in floating point, and in optimized code integer operations are used mostly for control flow and index computation and are usually removed from the innermost loops. We do take into account the control flow in the form of loop overheads.
- *Ignore all memory accesses that are not array references.* The reason for this heuristic is that scalar references occur relatively infrequently in scientific codes and, if they do, current architectures often have enough registers to buffer them.
- *Overlap operations.* For multiple issue architectures with multiple functional units, we must allow operations in different categories to overlap execution. For example, on the R10000 processor, there can be 6 instructions issued in one cycle: 2 integer operations, 2 floating point operations, 1 memory operation and 1 branch.

Using these approximations we obtain a lower bound on the processor's execution time.

Memory Hierarchy Model The term T_{MEM} in (1) estimates the time spent accessing memory locations in the memory hierarchy. As we mentioned before, when estimating the the execution time of basic operations we assume all memory references are cache hits in the first level cache. However, many accesses are not served from the first level cache, in part because applications have data sets much larger than the cache.

T_{MEM} can be expressed as follows:

$$T_{MEM} = CycleTime \times \sum_{i}^{\#levels} (M_i \times C_i), \tag{3}$$

where M_i represents the number of accesses that miss in the i^{th} level of the memory hierarchy, and C_i represents the penalty (in machine cycles) for a miss in

the i^{th} level of the memory hierarchy. C_i is computed using microbenchmarking, as in [19].

The compiler computes the number of array references in each statement and aggregates the values across blocks of statements, loops, and procedures. We do not include scalar references in this version of the model because, as mentioned above, we assume that the register files and the first level caches in most current processors are large enough to hold the majority of scalars. We propose two models to estimate the number of cache misses, which are used according to how much compile-time information is available.

Stack Distances Model. This model is based on the stack histogram obtained from a stack processing algorithm [5, 17] and requires accurate data dependence distance information [2]. The stack processing algorithm works on a program trace, where the elements in the trace can be memory locations, cache lines or virtual pages, and builds a stack as follows: for each element in the trace, check if the element is present in the stack. If it is, store the distance from the top of the stack where the element was found (the stack distance) and move the element on the top of the stack. If the element is not found, store ∞ as the stack distance and push the element on the top of the stack. The stack histogram is the distribution of the number of references at different stack distances. Stack algorithms have been used to simulate memory and cache behavior of programs [15, 17, 23–25].

We have devised a compile-time algorithm to compute the stack histogram. The compiler will label each data dependence arc (including input dependences) with the number of distinct memory locations accessed between the source and the sink of the dependence. The dependence with the minimum label will give the stack distance δ for the reference that is the sink of the data dependence arc. We also compute the number of times a static reference is accessed, A_δ. The number of cache misses for the cache at level i with cache size S_i is computed using the formula:

$$M_i = \sum_{\delta=S_i}^{\infty} A_\delta \tag{4}$$

In order to take into account spatial locality, we adjust the number of accesses when the dependence between two references that could potentially share a cache line satisfies the sharing conditions [5].

Indirect Accesses Model. When data dependence vectors are not available, we must approximate the number of cache lines that are spanned by the array references using other methods. This situation occurs mostly when indirect accesses are present in the code; therefore, we call this model the *indirect accesses model*, although it can be applied to any loop. In this model, we obtain the number of misses by multiplying each array reference A_j with the array element size e_j, and dividing the sum by the number of bytes for each block size in the memory

hierarchy. Thus, M_i can be expressed as:

$$M_i = (\sum_j A_j \times e_j)/BlockSize_i \quad (5)$$

When indirect accesses are involved, we need to apply a correction to the estimation since there could be many accesses to the same element. For example, many of the accesses to array X in statement S2 in Fig. 2 can map to the same element, depending on the value of the array element $ja(k)$. Therefore, we approximate the number of accesses with the minimum between the number of accesses and the size of the array. If the size is not known at compile time, the compiler can obtain it by run-time profiling.

For both models, the expressions are computed symbolically and we use profiling data to replace the parameters that depend on the input data set.

2.3 Putting it All Together

The static phase of the prediction is completed when the compiler instruments the code with the symbolic expressions for performance estimation. In the next phase, dynamic prediction, we compile the instrumented code using a native compiler and run the code in order to gather the run-time profiling data. Note that to obtain profiling data, the code need not run on the architecture for which we are predicting performance, therefore, the prediction model is not constrained by the existence of the architecture.

After the dynamic phase is completed, results are merged with the architectural parameters determined through microbenchmarking (or supplied by the user in case the architecture under study is not available) and can be used to guide compiler optimizations, and/or to be displayed using a performance prediction visualization tool, as mentioned at the beginning of this section.

3 Experimental Results

We have implemented the prediction model using the Polaris source-to-source restructurer [3]. Polaris contains implementations of most of the classical optimizations. It also has a dependence analysis pass that computes distance vectors whenever possible. We are using Polaris to analyze codes from the SPECfp95 benchmarks [18] and the Indiana University SpLib package, which is part of the Linear System Analyzer [4].

In Table 1 we present a summary of the loops analyzed and estimated by Polaris for the SPECfp95 benchmarks. For each benchmark, in the first two columns, we show the total number of loops that are present in the program and the number of loops that do not containing I/O, thus the loops that we considered in this paper. The next columns show the distribution of the estimated loops based on the amount of compile-time information available. "Full" means that Polaris was able to compute the data dependence distance vectors for all array references in the loop. "Partial", means that while all the dependence distances

were computed, some of the dependences have non-constant distances. For both these cases we can apply the stack distances model to predict the number of cache misses. For the second case we assume that accesses take place at the minimum distance. "Missing" represents the case in which Polaris could not compute the dependence distances for all the array references, and therefore the indirect accesses model is the only strategy that can be applied to predict performance. "Profiling" is the case in which the compiler needs run-time data due to unknown branch frequencies. For multiply nested loops, each loop of the

Benchmark	Total Loops	Estimated Loops	Compile-time Information			Profiled
			Full	Partial	Missing	
APPLU	168	149	137	8	0	4
APSI	298	231	175	29	7	20
HYDRO2D	165	158	124	15	6	13
MGRID	57	47	46	0	1	0
SU2COR	117	82	56	11	1	14
SWIM	24	24	24	0	0	0
TOMCATV	16	12	11	1	0	0
TURB3D	70	60	40	2	13	5
WAVE5	362	334	223	51	19	41
Total	1277	1097	836	117	47	97
Percentage		85.90%	76.21%	10.67%	4.28%	8.84%

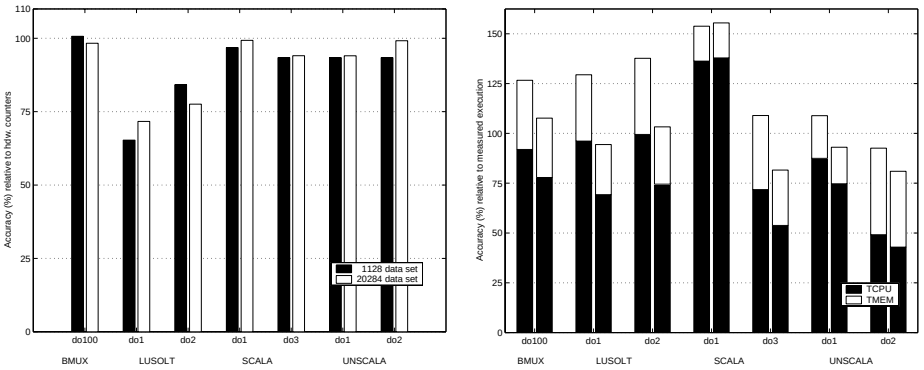
Table 1. Compile-time Stack Distances Accuracy

nest counts in only one category. For example, if the innermost loop of a doubly nested loop can be analyzed precisely, but the outermost can not, the innermost loop will be counted in the "full" column, and the outermost will be counted in the "missing" column. Both loops are part of the total and estimated loops.

To validate our model, we have conducted experiments on the MIPS R10000 processor and the UltraSparc IIi processor. The R10k processor is a 4 issue out-of-order superscalar architecture, with 32 KB on-chip L1 cache and 1 MB–4 MB off-chip L2 cache. In one cycle, the R10K can execute up to two integer instructions, two floating point instruction, one memory operation and one branch. Both caches are two way set-associative. The UltraSparc is a 4 issue in-order superscalar processor. It can execute up to two integer, two floating point, one memory and one branch operation per cycle. The caches, both the 16 KB on-chip L1 cache and the 256 KB off-chip L2 cache, are direct mapped.

In the following figures we present the prediction accuracy of our strategy. The accuracy is computed as the predicted execution time divided by the measured execution time. For cache predictions the accuracy is computed as the predicted number of misses divided by the actual number of misses measured using hardware counters. Thus, in both cases, the closer to 100%, the better the prediction.

In Fig. 3 we present the accuracy of predictions using the indirect accesses model for unoptimized loops in the SpLib package. Each bar in the graphs represents the performance accuracy for one of the following data sets: a sparse matrix of 1128 $\times$ 1128 with 13360 non-zero elements (the left bar in each group), and a 20284 $\times$ 20284 sparse matrix with 452752 non-zero elements (the right bar in each group). Figure 3(a) displays prediction accuracy for the L1 cache with respect to actual cache misses measured using the hardware counters on the R10k processor. The accuracy is within 25% for most of the loops. The loop `do1` in subroutine `LUSOLT` shows somewhat less accuracy, due the fact that our model does not account for conflict misses. Figure 3(b) shows execution time prediction accuracy. The accuracy of the prediction remains within 25% for most of the loops. The results for the optimized codes are somewhat worse, especially for the small data set. We attribute this to interloop reuse, which our indirect accesses model does not capture.



(a) L1 cache prediction using the indirect accesses model. Each bar represents a different data size.

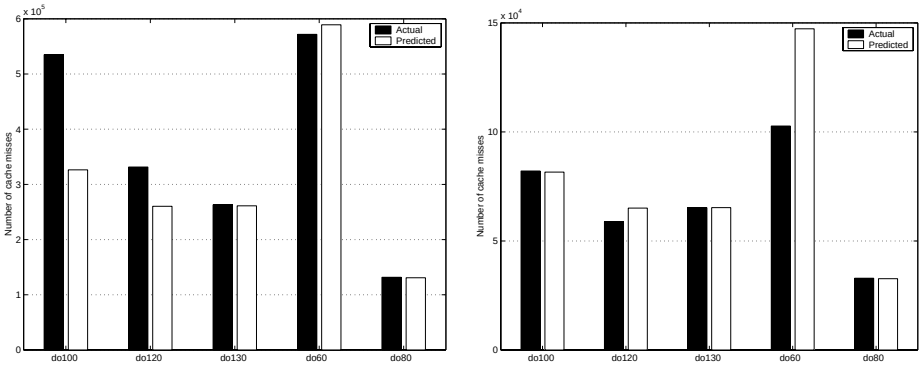
(b) Execution time prediction using the indirect accesses model. For each loop the two bars represent different data sets.

Fig. 3. Prediction accuracy for loops in SpLib for two input data sets on the R10k

To predict the execution time for the SPECfp95 benchmarks we have used the following strategy: for all the loops that we can estimate (loops that do not contain I/O) we use the compiler to estimate the number of operations and then use (2) to obtain T_{CPU}. We also estimate the number of cache misses, for both levels of cache, using both the stack distances model and the indirect accesses model. If the stack distances model cannot be applied because not enough compile-time information is available, we apply the indirect accesses model. We convert the number of misses into execution time using (3). To get the overall figures for the benchmarks, we multiply the predicted execution time for each

loop by the number of times it is executed, and then sum these times to get the execution time. We compute the actual execution time similarly, using measured execution times for each loop.

In Fig. 4 we present the cache estimation accuracy for both levels of cache on the R10000 processor for a set of loops in the TOMCATV benchmark. The left bars (black) represent the actual number of cache misses measured using hardware counters, while the right bars (white), represent the predicted number of cache misses. The prediction is very accurate for most of the cases. The two exceptions, the prediction for loop `do100` in the L1 cache does not capture all the inter-array conflict misses; and the prediction for loop `do60` in the L2 cache does not capture some of the spatial locality. However, as we shall see in Fig. 5, it will not affect adversely the overall execution time.

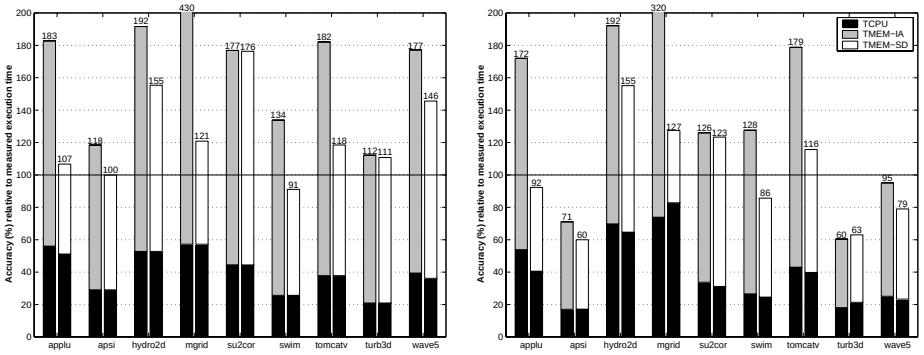


(a) L1 cache (b) L2 cache

Fig. 4. Cache prediction accuracy for the TOMCATV benchmark on the R10k

We present results for codes optimized using the default level of optimization for the native compiler, which is O3 for the SparcWorks Fortran compiler and O2 for the MipsPro Fortran compiler. In Fig. 5 we present prediction accuracy for the SPEC95fp benchmarks. For each benchmark we show two bars, one which uses the indirect memory access estimation model (left), and the other one which uses the stack distances model (right). The two sections of each bar represent the percentage of CPU execution time and the percentage of memory hierarchy access time, respectively.

The results in Fig. 5(a) show prediction accuracy for the R10k processor. The indirect accesses model does not perform well in these codes since it does not account for much of the reuse present in dense computations. The stack distances model prediction (right bars) is, for most of the benchmarks within 20% of the actual execution time. The three exceptions, HYDRO2D, SU2COR



(a) MIPS R10000 (b) UltraSparc IIi

Fig. 5. Execution time prediction accuracy for the SPECfp95 benchmarks. The left bars show results using the indirect access model, while the right bars show accuracy using the stack distances model

and WAVE5 exhibit interloop reuse and our compile time model does not capture it at the present time. However, we have used a run-time implementation of the stack distances model to quantify the amount of interloop reuse, and with that method the prediction accuracy was 142% for HYDRO2D, 108% for SU2COR and 110% for WAVE5. With those figures, the average prediction error for the entire set of benchmarks is 14%.

The chart in Fig. 5(b) displays results for the UltraSparc processor. For this processor we tend to under-predict the execution time, the average prediction error being 27%. One reason is that both levels of cache on this processor are direct mapped. Since our compiler model works for fully associative caches, there are many conflict misses that we do not capture. However, there are methods such as the one presented in [15], that allow approximating the number of cache misses for a set associative cache from the number of misses for a fully associative cache. Again, using the run-time implementation of the stack distances model, the average prediction error for the entire set of benchmarks on the processor drops to 20%.

4 Related Work

The need for good tools to estimate the application performance has been, and continues to be, high on the desire list of many application developers and compiler writers. While most of the performance prediction estimates have relied on simulations, there have been a few attempts to predict performance at compile time, mostly for applications running on multiprocessor systems. We summarize

a few of these attempts, and explain how our work makes use of those results and how our work contrasts with some of the other methods.

Saavedra et al. [19–21] has done extensive work in the area of performance prediction for uniprocessors. In [21], the authors present the microbenchmarking concept to measure architectural parameters. We use the same microbenchmarking approach to estimate operation costs (including intrinsic functions) and cache latencies. In [20], they present an abstract machine model that characterizes the architecture and the compiler. Their early model does not consider memory hierarchy effects. They do consider such effects in [19], but not using compile-time prediction. Their estimation of the number of cache misses relies on the measurements for the SPEC92 benchmarks presented in [13].

Sarkar [22] presents a counter-based approach to determine the average execution time and the variance in execution time for programs. His approach relies on a *program database* to collect information about execution frequencies of basic blocks. However, in estimating the execution time of the program, he assumes known basic block costs. We could use the same control flow based method to count frequencies of execution for the basic blocks, including their optimizations to minimize the number of counters; however we also would need to compute the basic block costs, and the memory hierarchy penalties. We could create the program database from our symbolic expressions.

Fahringer [9–11] describes P^3T, a performance estimation tool. He uses the Vienna Fortran Compilation System as an interactive parallelizing compiler, and the *WeightFinder* and P^3T tools to feedback performance information to both the compiler and the programmer. While our goals are similar, the works differ in the approach taken. Fahringer uses pattern matching benchmarking based on a library of kernels to estimate execution time. A program is parsed to detect existing kernels and pre-measured run-times for the discovered kernels are accumulated to yield the overall execution time. The prediction relies heavily on the quality and completeness of the kernel library. To estimate cache behavior the author classifies the array accesses with respect to cache reuse. An estimated upper bound of the number of cache lines accessed inside a loop is computed. Misses for loops can be aggregated for predicting procedures and entire programs.

We cannot readily compare the accuracies of the two methods, as Fahringer presents experimental results for HPF programs only and, therefore, uses message passing, while our work is targeted to programs running on shared memory machines.

Ghosh *et al.* [14] have introduced the *Cache Miss Equations* (CMEs) as a mathematical framework that precisely represents cache misses in a loop nest. They count the cache misses in a code segment by analyzing the number of solutions present for a system of linear Diophantine equations extracted from reuse vectors, where each solution corresponds to a potential cache miss. Although solving these linear systems is difficult, the authors claim that mathematical techniques for manipulating the equations allow them to relatively easily compute and/or reduce the number of possible solutions without solving the equations.

One of the first attempts to use profiling information in order to improve compiler optimizations was trace scheduling for very long instruction word (VLIW) processors [7, 12], which uses profile data to optimize execution of the most probable code execution paths. More recently, Chang *et al.* [6] have developed an optimizing compiler that uses profiling information to assist classic code optimizations. The compiler contains two new components, an execution profiler and a profile-based code optimizer. The execution profiler inserts probes into the input program, executes the input program for several inputs, accumulates profile information, and supplies this information to the optimizer. The profile-based code optimizer uses the profile information to expose new optimization opportunities that are not visible to traditional global optimization methods.

5 Conclusions

The prediction environment presented in this paper integrates a compiler derived model with run-time instrumentation to estimate performance for scientific codes. We have presented the compiler-driven performance prediction model, which although very simple, is able to predict performance of codes in the SPECfp95 benchmark suite with 20% error margin on complex architectures. This model can also be extended to shared multiprocessor codes because our compiler is a parallelizing source-to-source translator.

We use the results of the compiler-driven performance prediction in two ways. First, the compiler can use the performance prediction results to guide optimizations. Examples are deciding if a parallel loop is worth running in parallel based on the sequential execution time and the overhead to run the loop in parallel. Second, the performance prediction results can be displayed using performance visualization tools and the performance data can be related back to the high level source code based on information provided by the compiler.

References

[1] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation '93*, pages 300–313, 1993.

[2] U. Banerjee. *Dependence analysis*. Kluwer Academic Publishers, 1997.

[3] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, December 1996.

[4] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. *The Linear System Analyzer*, chapter PSEs. IEEE, 1998.

[5] C. Cascaval and D. A. Padua. Compile-time cache misses estimation using stack distances. In preparation.

[6] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic compiler code optimizations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[7] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of ASPLOS II*, pages 180–192, Palo Alto, CA, October 1987.

[8] L. DeRose, Y. Zhang, and D. A. Reed. SvPablo: A multi-language performance analysis system. In *10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools'98*, pages 352–355, Palma de Mallorca, Spain, September 1998.

[9] T. Fahringer. Evaluation of benchmark performance estimation for parallel Fortran programs on massively parallel SIMD and MIMD computers. In *IEEE Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing*, Malaga, Spain, January 1994.

[10] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Press, 1996.

[11] T. Fahringer. Estimating cache performance for sequential and data parallel programs. Technical Report TR 97-9, Institute for Software Technology and Parallel Systems, Univ. of Vienna, Vienna, Austria, October 1997.

[12] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C(30):478–490, July 1981.

[13] J. D. Gee, M. D. Hill, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. In *Proceedings of the IEEE Micro*, pages 17–27, August 1993.

[14] S. Ghosh, M. Martonosi, and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proceedings of ASPLOS VIII*, San Jose, CA, October 1998.

[15] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[16] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.

[17] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[18] J. Reilly. SPEC95 Products and Benchmarks. *SPEC Newsletter*, September 1995.

[19] R. Saavedra and A. Smith. Measuring cache and tlb performance and their effect on benchmark run times. *IEEE Transactions on Computers*, 44(10):1223–1235, October 1995.

[20] R. H. Saavedra-Barrera and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report CSD 92-715, Computer Science Division, UC Berkeley, 1992.

[21] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers*, 38(12):1659–1679, December 1989.

[22] V. Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation '89*, pages 298–312, Portland, Oregon, July 1989.

[23] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Trans. Comp. Sys.*, 13(1), 1995.

[24] J. G. Thompson and A. J. Smith. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems*, 7(1), 1989.

[25] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3), 1991.