

A Compiler Approach to Performance Prediction using Empirical-based Modeling

Pedro C. Diniz

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292, USA,
pedro@isi.edu

Abstract. Performance understanding and prediction are extremely important goals for guiding the application of program optimizations or in helping programmers focus their efforts when tuning their applications. In this paper we survey current approaches in performance understanding and modeling for high-performance scientific applications. We also describe a performance modeling and prediction approach that relies on the synergistic collaboration of compiler analysis, compiler-generated instrumentation (to observe relevant run-time input values) and multi-model performance modeling. A compiler analyzes the source code to derive a discrete set of parameterizable performance models. The models use run-time data to define the values of their parameters. This approach, we believe, will allow for higher performance modeling accuracy and more importantly to more precise identification of what the causes of performance problems are.

1 Introduction

Despite the tremendous peak performance of high-end computing architectures, they deliver abysmally poor performance for current scientific and engineering applications. As these applications have millions of lines of source code and manipulate vast amounts of data, manual instrumentation and program understanding are infeasible. The standard approach to the problem of performance understanding relies on tools that profile the code execution and provide aggregate measures of performance. While it is useful to know that a given `do` loop has substantial *L1 cache misses* or *TLB misses*, that information gives little insight on how to remedy the performance problems.

The effects of compiler optimizations exacerbate this problem. Compilers apply a wide variety of transformations making it very difficult to map the effects of the code into high-level programming abstractions developers can reason about. We believe the key to address the problems of performance understanding and prediction is to develop techniques that take into account the effects of compilers at the instruction level and reason about the mapping of the instructions to the target architecture. To be useful such tools must be automated and must retain as much information about the high-level programming abstractions as possible.

This paper describes a performance modeling and prediction approach that uses traditional compiler analysis techniques, both at the source code level and assembly level, in collaboration with empirical performance modeling techniques. The compiler isolates sequences of instructions (called basic blocks in the compiler parlance) and maps them to high-level programming abstractions. Associated with each basic block the compiler builds a set of discrete performance models tailored for specific run-time execution scenarios. For example a *cache miss* or a *TLB miss* may lead to severe pipelining execution problems. In order to determine which of the set of models to apply the compiler generates and executes a skeleton of the application. The execution of the skeleton, will allow the compiler to extract the relevant run-time data which was identified statically. The compiler then feeds the information gathered by the skeleton and derives the actual model parameters and frequency of application of each model to predict the overall performance of the original application with its real data.

If successful this approach would provide, we believe, not only more accurate performance prediction but as a by-product, an understanding of what the cause-effect relations of program constructs on the performance are. The proposed performance modeling techniques can also be used as part of a fully automated program optimization tool. Such a tool would iterate over a given section of the code trying out several transformations observing which sequences of transformations would lead to better predicted performance before committing to the application of such transformations.

The remainder of this paper is organized as follows. In the next section we survey current approaches to performance modeling and understanding in high-performance scientific applications. Section 3 describes in more detail the modeling approach proposed in this paper. Section 4 describes three applications of the proposed approach followed by a brief discussion of the research challenges in section 5. Finally we summarize this presentation in section 6.

2 State-of-the-Art

We now describe current approaches in the area compiler optimizations specifically by addressing performance-aware compilation systems and feedback-directed optimizations. We also describe various efforts in performance modeling and prediction for large parallel codes on current and future processor architectures.

2.1 Performance-Aware Compilation

In many instances the lack of statically available information may prevent the compiler from applying program transformations more aggressively or simply from applying them all together. Several systems address this problem by a combination of static information and run-time testing. The inspector/executor approach dynamically analyzes the values in index arrays to automatically parallelize computations that access irregular meshes [1]. Speculative approaches optimistically execute loops in parallel, rolling back the computation if the parallel execution violates the data dependences [2]. Dynamic compilation systems

enable code generation at run time [3, 4] allowing the compiler to exploit knowledge about input values and hence generate more efficient code. The Dynamo system [5] continuously optimizes code based on performance data gathered incrementally at run-time.

As an alternative, researchers have developed approaches in which the compiler generates for selected computation sections a limited set of compiled versions each of which corresponds to the application of a particular set of program transformations. At run-time the generated code selects which version of the code to choose based on a set of compiler generated predicates or even by evaluating the performance of each alternative implementation and choosing the one with highest performance. This approach can be done entirely automatic as in [6] or with the involvement of the programmer to specify application specific adaptation and optimization strategies as in [7].

There are several on-going research projects in empirical optimization of scientific libraries through basic performance driven selection of multiple code variants (*e.g.*, ATLAS [8] or PhiPAC[9]). In these projects the compiler generates many implementations of the same computation, *e.g.*, *matrix-multiply*, for different optimization strategies in a purely off-line fashion and then selects, based on previous executions, which version performs best for each target architecture.

The GrADS[10] project aims at extending the notion of compile-time and run-time by creating a malleable object code that can be configured to a wide variety of resource availability scenarios. A configurable object program contains, in addition to the application code, strategies for mapping the application to different collections of resources and a resource selection model that provides an estimate of the performance of the application on a specific collection of resources. The GrADS approach project also relies on notions of performance contract to specify when reconfiguration of application code or resources should be triggered to maintain acceptable performance levels.

2.2 Performance Prediction

Other researchers have developed static estimators to guide the application of program transformations with the ultimate goal of improving performance.

For example in [11] the authors have developed a series of empirical models for the impact of data distribution in the performance of parallel applications on distributed memory machines. The system first "trains" the estimator with known communication and data partition patterns found in kernel routines and use the resulting estimator models for complete applications.

In the context of the POLARIS system researchers have also developed a methodology for statically predicting the performance of applications using a combination of static analysis and profiling information[12]. The approach uses source code analysis information to derive analytical expressions defining the number of expected *L1 cache misses* and basic arithmetic operators as a function of loop bounds and uses architecture characteristics (*e.g.* number of functional units and pipelining depth). Using these analytical expressions and real run-time data such as loop bounds the compiler can predict the overall execution time.

2.3 Performance Modeling & Understanding

Other researchers have also modeled application performance based on the high-level definition of the problem rather than directly on the code implementation. Kerbyson *et.al* [13] describe a series of modeling case studies where they use the data and computation partitions as well as their own empirical model for communications to define analytical expressions that track very well the observed performance on multiprocessor machines.

Other approaches have focused on modeling program behavior for large-scale parallel and distributed applications (*e.g.*, [14, 15]). This work aims at understanding the sources of the applications' performance. In [16] the authors developed a set of simple models to capture the memory and communication behavior of each application. These metrics are capture include the cache-miss ratios and memory bandwidth via simulation and then convolved with the target architecture parameters to determine the expected performance. This approach aims at attaining better accuracy than *back-of-the-envelope* calculations without the extreme cost of cycle-level accuracy.

Other researchers have also studied the scalability of parallel scientific computations by empirical measurements using statistical instrumentation with currently available performance monitoring tools regarding computation and communication. In [17] the authors use the observed metrics to refine the explanations of the factors that influence application performance and scalability.

3 Empirical Modeling

Performance modeling techniques offer an alternative way of enabling the compiler to derive and possibly select a set of program transformations. This modeling approach is particularly valuable in scenarios in which the application can take an extremely long time to execute making profiling impractical or when attempting to predict the performance on future architectures.

3.1 Overview

The modeling approach described here and depicted in figure1 relies on collaboration between static compiler analysis and run-time data, but unlike post-mortem profiling uses cues from the execution of a skeleton of the original code to derive better performance prediction models.

During a first phase the compiler identifies the application's control-flow-graph (CFG) isolating the basic blocks of instructions and retaining the mapping between relevant instructions (such as loads and stores) to the high-level programming constructs such as array accesses and arithmetic expressions. While in general this seems a daunting problem given the various internal and target-specific compiler transformations, we believe it is possible to develop effective techniques that derive a mapping that will allow the compiler to provide useful information to the programmer. The recent experience with the HPCView[18]

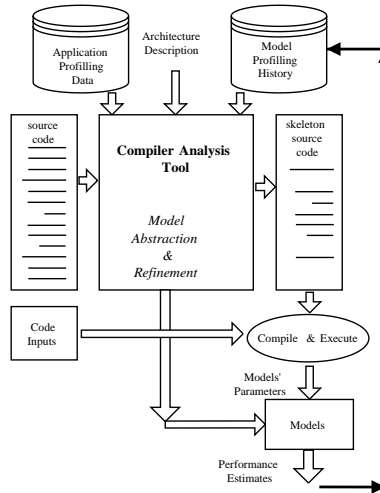


Fig. 1. Performance modeling approach.

tool supports this claim. During this phase the compiler collects static information for each basic block such as the number of memory references and address calculations along with floating-point and integer operations¹

For each basic block the compiler also derives a set of possible execution scenarios. These scenarios are based on the values of particular run-time variables that are bound to significantly affect the performance. For example a given loop might exhibit very distinct performance if the arrays it accesses are not in cache and/or if the stride of the accesses is larger than the cache line size. While in some cases the compiler can understand statically which of the array references are bound to create performance problems in other cases the stride of the access depends on the values of other variables. In other cases, such as conflicting misses the compiler can only determine this information once the array have been laid out in the code's virtual address space. Based on the assumptions for each scenario the compiler then derives a performance model. This model is parameterizable by run-time values such as loop bounds; array access stride or even the layout of the various arrays and their relative position in memory.

In order to determine the exact values of the various parameter models the compiler must determine the exact run-time values of the program variables that can affect the parameters of the various performance models for each program section. To derive these values the compiler generates a *skeleton* program based on the original code and runs it with the original code inputs. This *skeleton* program bypasses some of the code of the original application as the exact outputs are not relevant but retains the values of a set of variables (*e.g.*, loop bounds) that are important for modeling purposes. The exact set of variables

¹ Collecting such static metrics by looking at the source is bound to lead to terribly inflated metrics as compilers apply program transformations such as common code elimination enabled by similarities in array indexing.

whose values need to be captured by the *skeleton* program are derived statically by compiler analysis. The compiler then executes the *skeleton* program and collects the actual data values. Using the run-time values the compiler selects which of the pre-analyzed models to apply for each code section. Using the resulting output metrics for each selected model (by evaluation analytical performance expressions associated with that model using actual run-time values) the compiler derives execution estimates for the performance contributions of each region.

3.2 Program Skeletons

In many of these situations analyzing the structure of the computation and run-time variable values such as loop bounds and array access strides provides valuable information for the purpose of performance modeling and prediction of run-time execution. This data can be obtained, in many cases, by simple inspection of a subset of the variable the program manipulates. The example in the figure 2 illustrates a case where the out of 4 array accesses, only 1 has variable stride and is bound to create substantial *cache* or *TLB misses*.

In the *skeleton* code the `do` loop is eliminated and instead the code extracts the relevant information to apply to an execution the model. It saves in an internal data structure the values of `n`, `mstride` and records which arrays have long and small access strides. The arithmetic functions `abs` and `sqrt` as well as the loop have been removed and the corresponding execution time can be accounted by using target architecture dependent constants. When encountering control-flow that is dependent on computed values, the compiler must use profile-based data about the frequency with which each of the branches was taken or retain in the skeleton code the computation that conditional statements depend on. While in the worst case scenario a given conditional statement would force the execution of the entire application, we believe that many computations will not elicit this behavior. As a fallback position the compiler can rely on accurate profiling [19] information regarding the frequency of the execution of each branch and bypass the information the execution of a skeleton code provides.

<pre> mstride = ... do ip=1,n b(ip) = two*vx(ip) + abs(...) b(ip) = sqrt(...)*face(mstride*ip) enddo </pre>	<pre> mstride = ... call save(address,mstride,n) if(mstride .gt.cache_line) then call record_long_stride(address,array,"face") else call record_unit_stride(address,array,"face") endif call record_unit_stride(address,array,"a") call record_unit_stride(address,array,"b") call record_unit_stride(address,array,"vx") </pre>
---	--

Fig. 2. Skeleton Extraction Example (original code on the left, skeleton on the right).

3.3 How to Derive Models

The model for each basic block of instructions in the code is derived by looking at the data dependences between the data required by each instruction. Internally the compiler builds a data-flow graph for each basic block and determines the set of data that is generated in register for each block as well as the set of data values generated to be used by other blocks.

Using the target architecture description in terms of the number of functional units, pipelines and their depth the compiler can derive a set of performance analytical expressions for a set of scenarios and determine for each of these scenarios what the expected performance is in terms of consumed clock cycles and peak performance the execution of the code section would take. For example, if one of the various memory references in the basic block causes a *TLB miss* that leads to a pipeline stall (due to data dependences) this leads to a substantial decrease in overall performance. Another scenario could explore the performance consequences of the references to the sparse array not being in cache. For each of the scenarios, the corresponding model can be obtained either empirically and/or by using target architecture cycle level accurate simulations.

The compiler analyzes a discrete set of such scenarios and enumerates the corresponding models. When generating the skeleton code the compiler also generates code to abstract, and keep track at run time, of the portion of the processor state that is relevant to each model. While in general this approach can lead to a full blown functional-level processor architecture simulation we believe it is possible to develop simple models for selected components of the architecture whose poor performance provide important insight into the overall program behavior. Tracking the values of consecutive array accesses by inspection of the corresponding indices, for example, allows compilers to signal which of the array references will clearly lead to potential performance problems.

4 Applications

4.1 Compiler Optimizations

Compiler writers have a wide range of program transformations at their disposal to attempt to improve the quality of the generated executable code. Unfortunately it is not easy to determine statically what the best sequence of program transformations is.

A compilation and program optimization system could use the approach outline above by using performance profiling information of previous runs to refine the performance models. This knowledge would help the compiler to select which optimization strategies are likely to produce better results. Because the performance of generated codes can varies widely with distinct data input characteristics, the compiler could use the performance models to select a set optimization strategies geared towards different data settings. The compiler would generate multiple code version of the same computation and select at run-time which of the selected code versions to use based on the assumptions for each code version.

4.2 Generating Performance Assertions

In the quest for understanding the performance bottlenecks of their applications researchers have developed the mechanism of performance assertions [20]. The current practice calls for the programmers to manually specify what the performance assertions would be.

Using this mechanism programmer must examine selected portions of their code and determine manually what a reasonable performance expectation a given segment of the code should deliver. When violated, the corresponding performance violation handling code (typically a write statement) will indicate the location and nature of the violation. Besides being tedious and error prone this process is highly non-portable. A given performance assertion might be acceptable to one target architecture but very unrealistic in another. This leads to an excessive number of spurious performance assertions violations which detracts the programmer from its main purpose find the real performance bottleneck problems. The automated approach proposed in this paper would aims at deriving the performance assertion directly from architecture specifications only raising performance assertion exceptions when a given threshold metrics say 10% of peak performance were predicted and providing additional information about why the performance model is reaching that particular performance level.

4.3 Interactive Performance Understanding

Ultimately it is the programmer who can profoundly impact the performance of its application. We foresee the application of the techniques proposed in this paper as part of interactive performance understanding systems that allow programmer to understand which data structure are substantially impacting performance and provide insight why and what to do about it. For example, programmers often use pointer variables for extreme flexibility. In the contexts of tight numeric intensive loop dereferencing pointers to retrieve/store data from/to memory might lead to severe pipeline stalling. Based on poor performance estimates suggested by its pipeline models the compiler could suggest the programmer to restructure its code by converting pointer access in the tight numeric loop to array accesses to a temporary array variable which the programmer loads (using a pointer-based only loop) before the numeric loop is executed.

5 Research Challenges

While appealing this approach raises several implementation and design challenges in building an automated and effective performance modeling and prediction systems for large scientific codes. First, how accurate can this approach be? If the models are too simple the effort might not warrant the benefits; if too expensive the quantity and quality of the parameters might be as hard to extract as examining the impact of each instruction. Is there a meaningful middle-ground? Second, can basic blocks of instructions be meaningfully mapped to high-level programming constructs so as to provide good high-level program information

about performance problems? Third, what is the precision of this approach in the presence of more sophisticated architectural features that are so hard to model? Forth, is it feasible for a static compiler analysis to generate a code skeleton to extract a set of meaningful parameters for each model? Can the implementation effectively capture a limited set of context representative of a wide set of execution scenarios? Finally, and given the large scale nature of the target applications, how does this approach scale?

6 Summary

In this paper we described an approach that relies on the synergistic collaboration of static compiler analysis, compiler-generated instrumentation (to observe relevant real run-time input values) and multi-model performance modeling of sequences of instruction for the target architecture (derived empirically and calibrated and validated off-line by cycle-accurate simulations). While there are many challenges to a practical implementation of the proposed approach we believe it is possible to build a program analysis tool that can deliver realistic performance estimates that are useful to programmers in understanding the source of their applications performance issues.

References

1. Saltz, J., Berryman, H., Wu, J.: Multiprocessors and run-time compilation. *Concurrency: Practice & Experience* **3** (1991) 573–592
2. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loop with privatization and reduction parallelization. In: *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'95)*, ACM Press (1995)
3. Auslander, J., Philipose, M., Chambers, C., Eggers, S., Bershad, B.: Fast, Effective Dynamic Compilation. In: *Proc. ACM Conference on Programming Language Design and Implementation (PLDI'96)*, ACM Press (1996)
4. Engler, D.: VCODE: A retargetable, extensible, very fast dynamic code generation system. In: *Proc. of the ACM Conference on Program Language Design and Implementation (PLDI'96)*, ACM Press (1996)
5. Bala, V., Duestervald, E., Banerjia, S.: Dynamo: A Transparent Run-time Optimization System. In: *Proc. of the ACM Conference on Programming Languages Design and Implementation (PLDI'00)*, ACM Press (2000)
6. Diniz, P., Rinard, R.: Dynamic feedback: An effective technique for adaptive computing. In: *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'97)*, ACM Press (1997)
7. Voss, M., Eigenmann, R.: High-Level Adaptive Program Optimization with ADAPT. In: *Proc. of the ACM Conference on Principles and Practice of Parallel Processing (PPoPP'01)*, ACM Press (2001)
8. Whaley, C., Dongarra, J.: Automatically tuned linear algebra software. In: *Proc. of Supercomputing (SC'98)*. (1998)
9. Bilmes, J., Asanovic, K., Chen, C.W., Demmel, J.: Optimizing matrix multiply using phipac: a portable high-performance ansi-c coding methodology. In: *Proc. of the ACM International Conference on Supercomputing (ICS'97)*. (1997)

10. Kennedy, K., Mazina, M., Mellor-Crummey, J., Cooper, K., Torczon, L., Berman, F., Chien, A., Dail, H., Sievert, O., Angulo, D., Foster, I., Gannon, D., Johnsson, L., Kesselman, C., Aydt, R., Reed, D., Dongarra, J., Vadhiyar, S., Wolski, R.: Toward a framework for preparing and executing adaptive grid programs. In: Proc. of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002). (2002)
11. Bala, V., Fox, G., Kennedy, K., Kremer, U.: A static performance estimator to guide data partitioning decisions. In: Proc. of the third ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'91), ACM Press (1991)
12. Cascaval, C., DeRose, L., Padua, D., Reed, D.: Compile-time based performance prediction. In: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC'99), Springer-Verlag (1999) 365–379
13. Kerbyson, D., Alme, H., Hoisie, A., Petrini, F., Wasserman, H., Gittings, M.: Predictive Performance and Scalability Modeling of a large-scale Application. In: Proc. of the Supercomputing Conference (SC'01). (2001)
14. Snively, A., Wolter, N., Carrington, L.: Modeling application performance by convolving machine signatures with application profiles. In: Proc. of the IEEE 4th Annual Workshop on Workload Characterization, Austin (2001)
15. da Lu, C., Reed, D.: Compact application signatures for parallel and distributed scientific codes. In: Proc. of the Supercomputing Conference (SC02). (2002)
16. Snively, A., Carrington, L., Wolter, N.: A framework for performance modeling and prediction. In: Proc. of the Supercomputing Conference (SC'02). (2002)
17. Vetter, J., Yoo, A.: An empirical performance evaluation of scalable scientific applications. In: Proc. of the Supercomputing Conference (SC'02). (2002)
18. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: A Tool for Top-Down Analysis of Node Performance. *Journal of Supercomputing* **23** (2002) 81–104
19. Ammons, G., Ball, T., Larus, J.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'97), ACM Press (1997)
20. Vetter, J., Wooley, J.: Performance assertions. In: Proc. of the Supercomputing Conference (SC'02). (2002)