Delphi: An Integrated, Language-Directed Performance Prediction, Measurement and Analysis Environment

Daniel A. Reed* David A. Padua
{reed,padua}@cs.uiuc.edu
Department of Computer Science
University of Illinois

Ian T. Foster
foster@mcs.anl.gov
Mathematics and Computer Science Division
Argonne National Laboratory

Dennis B. Gannon
gannon@cs.indiana.edu
Department of Computer Science
Indiana University

Barton P. Miller
bart@cs.wisc.edu
Department of Computer Science
University of Wisconsin

Abstract

Despite construction of powerful parallel systems and networked computational grids, achieving a large fraction of peak performance for a range of applications has proven very difficult. In this paper, we describe the components of Delphi*, an integrated performance measurement and prediction environment that places system design on a solid performance engineering basis.*

1. Introduction

As the scope of high-performance computing expands from regular computations and single, parallel systems to irregular computations and distributed collections of heterogeneous sequential and parallel systems, users increasingly complain that it is difficult or impossible to achieve a high fraction of the theoretical performance peak. In short, current high-performance computing systems are brittle. Small application, system software, or architectural changes often lead to large changes in observed performance.

This performance variability is a direct consequence of resource-interaction complexity and failure to assess the effects of these interactions during system design. Because these interactions involve application and system software, processors, I/O systems, and networks, we believe the only practical solution to this dilemma is to develop integrated performance measurement, analysis, and prediction systems that allow application and system developers to explore the performance implications of software and hardware design choices, both for extant and proposed systems.

An integrated modeling and measurement environment would allow designers to "mix and match" software and hardware components, validating the performance design goals of the complete system against a composition of calibrated models of proposed components and measurements of extant components prior to detailed design construction. Based on this thesis, we are building a new performance environment, called *Delphi* that leverages software from current projects at Illinois, Indiana, Wisconsin, and Argonne, including the Pablo [5], Paradyn [4], and Autopilot [6] performance analysis and measurement environments, the HPC++ and Polaris [1] Fortran compilers, and the Globus [2] metacomputing system.

Delphi's goal is to support measurement, analysis, and performance prediction of multilingual, SPMD, data-parallel, and object-oriented applications executing on both homogeneous, parallel systems and distributed collections of wide-area computing resources (i.e., computational grids). Below, we briefly outline *Delphi*'s components, current research efforts, and planned directions.

2. Delphi Organization

Delphi builds on our experiences with dynamic performance instrumentation tools, integrated compiler support for performance analysis and scalability pre-

*This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), F30602-96-C-0161, and DABT63-96-C-0027 by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

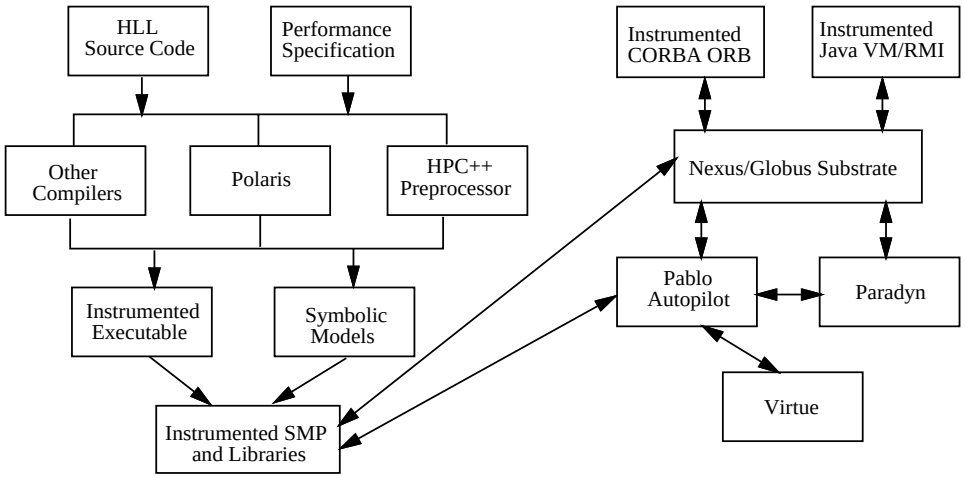

Figure 1. Delphi Organizational Structure

diction, and resource optimization for heterogeneous metacomputing. As Figure 1 suggests, *Delphi* integrates both parallel systems and wide-area metacomputing or computational grids [2] within a single performance measurement and prediction framework.

In this framework, annotating compilers emit detailed data on program transformations and annotate programs with calls to *resource reflectors*. Using this information, symbolic performance models can be used to rapidly assess the effects of software and architectural variations on achievable performance. For more detailed performance assessments, executing annotated applications invokes resource reflectors that either measure resource use or activate resource models.

Below, we describe each of the components of the *Delphi* system: (a) compilers that emit code annotated with symbolic, execution-cost expressions and executable interfaces to performance instrumentation and modeling routines, (b) performance models of key system components, including task schedulers, memory and input/output management, and networks, and (c) flexible measurement and analysis software for assessing performance on extant systems.

3. Annotation and Prediction

The tacit assumption underlying source or object code instrumentation to understand system performance is that the organization and structure of the compiler-generated code are similar to that of the source code. With sophisticated optimizing and parallelizing compilers, this assumption is increasingly false — compilers and restructurers aggressively transform the user-written code. To understand resource demands and the causes for performance variability in transformed code, compilers and performance tools must cooperate to integrate information about the program's dynamic behavior. The compiler must be aware of how its low-level, explicitly parallel code maps to the original source code.

3.1. Source Code Annotation

To integrate compilers with performance modeling and analysis tools, we are extending the Indiana HPC++ [3] and the Illinois Polaris [1] compilers, as well as developing instrumenting parsers, to emit code with embedded calls to instrumentation and resource reflectors and to generate symbolic performance expressions. Currently, C, Fortran 77, and Fortran 90 parsers, the PGI HPF compiler, and the Polaris Fortran restructurer all emit instrumented code whose measured performance can be correlated to hardware performance data.

3.2. Symbolic Prediction

Our model for performance prediction exploits both compiler-derived data on symbolic program variables and performance measurements from selected executions of the compiler-generated code. The latter provide calibration data and input-dependent control flow data. From these two sources, we derive scalability models that consist of symbolic expressions representing the execution complexity of individual program sections.

The high-level symbolic model includes terms for computation time, memory- reference overhead, I/O costs, and communication time; all are based on the performance models of §4. If value ranges are included (e.g., with multiple estimates of cache miss ratios), the symbolic model can produce bounding estimates.

In general, the high-level model includes both static and dynamic phases. The static phase traverses the abstract syntax tree (AST) of the program, symbolically estimating loop ranges and counting operations for each statement in the program. These operation counts are then used as coefficients for the independent variables in the expression.

Using symbolic arithmetic, these expressions are combined to generate execution costs for program basic blocks. The basic block expressions are then combined and augmented with coefficients obtained from the dynamic phase. This phase captures additional data that cannot be deduced during the static phase (e.g., the fraction of true and false branches taken in a particular conditional statement).

4. Performance Models and Libraries

When symbolic predictions and execution-driven models lack the requisite accuracy, performance measurements can provide the quantitative data needed to understand the detailed interactions among system components. As part of the *Delphi* effort, we are developing both instrumented libraries for common resources and compile-time and execution-driven performance models. The instrumented libraries can be combined with the performance models to form a hybrid execution model.

4.1. Memory Access

To provide memory-access time estimates for the computation models of §3, one must calculate the number of cache misses and measure the access time and miss penalty at each level of the memory hierarchy. Currently, we predict the number of cache misses by generating a memory- reference trace and creating a stack-distance histogram. If a particular memory location was previously referenced, the previous reference is removed from the stack, its distance from the top of the stack is recorded, and the reference is moved to the top of the stack. If the reference is new, it is pushed on the stack and a distance of infinity is recorded.

Using the stack-distance histogram, we compute cache hits and misses based on the reference distance. This enables us to simulate program behavior for multiple cache sizes in a single, architecture-independent pass. At present, we are working to increase the efficiency of this technique, making it applicable to truly large application executions.

4.2. Task Scheduling

The tasks or threads of parallel and distributed computations can be scheduled statically (e.g., assigning groups of loop iterations to specific processors) or dynamically (e.g., with a shared run queue). Though not easy, predicting the execution time of static task schedules is far simpler than estimating the behavior of adaptive computations.

In adaptive codes, threads are generated and scheduled at runtime. Moreover, these threads interact with the memory hierarchy in subtle and not-so-subtle ways. For example, if a large number of threads access the same set of memory locations, the effect is to serialize the computation.

As a prelude to developing performance models, we are measuring overhead for thread creation and scheduling and thread/memory interactions on current shared-memory parallel systems. Using this data, we will explore compile-time techniques that can estimate data access collisions.

4.3. Input/Output

For many high-performance applications, the input/output barrier rivals or exceeds that for computation, making design of scalable input/output architectures and file systems a key aspect of any high-performance system. Our experience [7] has shown that software layering is a potential source of performance loss. Hence, we are developing an instrumented versions of multilevel input/output libraries that enable multilevel analysis of I/O requests (i.e., requests from the PACI/ASCI Hierarchical Data Format (HDF 5) library, the interactions of those requests with the underlying MPI-IO toolkit, and their ultimate instantiation as UNIX I/O requests).

Building on our input/output characterizations, we are also developing queueing models of input/output scalability and file striping. These models predict the performance of parallel disk systems as a function of request rates and access patterns, request sizes, and disk hardware parameters [7]. They can be used to estimate I/O system scalability.

4.4. Networks and Computational Grids

With increasing development of applications for heterogeneous, distributed computing grids, performance measurement and prediction must include application tuning for heterogeneous resources. The components of these distributed applications execute on parallel systems or workstation/PC clusters and interact via message- passing systems like MPI, wide-area toolkits like Globus [2], and remote object invocations via CORBA, DCOM, and Java RMIs.

To assess the behavior of distributed applications, we have developed Paradyn extensions for

Globus/Nexus that can capture packet loss rates, inter-arrival jitter, and bandwidth, and we have created Autopilot daemons that monitor wide-area communication behavior. Based on our performance experiments with CORBA, DCOM, and Java RMI, we have also created a multivendor working group to define a high-performance, remote-invocation model for scientific applications. Our goal is to create a remote invocation model that can deliver high performance atop multiple communication protocols and whose behavior can be quantified for use in the models of §3.

5. Flexible Measurement Software

Multilevel performance data is needed both to parameterize symbolic and execution-driven resource models and to understand the behavior of new systems. Paradyn's dynamic instrumentation allows one to instrument and capture data from application processes and threads [9], while they are executing, by dynamically patching object code [4]. Conversely, Pablo's source-code instrumentation [5] allows compilers and parsers to emit instrumented application code, based on compile-time analysis and user specification. Finally, the Paradyn and Autopilot [6] toolkits allow one to capture the behavior of distributed, metacomputing applications via Nexus and wide-area instrumentation.

With Paradyn, Pablo, and Autopilot as a basis, we are developing instrumentation to capture standard software metrics (e.g., context switches, disk accesses, and utilization) [8]. This is complemented by standard hardware metrics (e.g., instruction mixes and cache miss ratios) obtained via microprocessor counters.

6. Presentation and Analysis Tools

Detailed understanding of the effects of architectural and system software design alternatives also requires powerful data-analysis and visualization tools. We are extending the Paradyn and Pablo [4, 5] software for data analysis and display and exploiting the Virtue toolkit for immersive visualization.

The new Pablo interface supports a hierarchy of performance displays, ranging from routine profiles to detailed data on the behavior a single source code line. Similarly, Paradyn displays multilevel metric histograms, correlated with application process and thread behavior. Finally, the Virtue immersive virtual environment [5] shows real-time views of distributed and parallel computations. Via these interfaces, ones can access and load performance data from multiple executions, including different numbers of processors and hardware platforms. This allows one to compare executions and understand component interactions.

7. Conclusions and Future Work

The high performance sensitivity of today's parallel systems to application system features makes application tuning complex. It also makes performance-directed system design counter-intuitive. Integrated systems like *Delphi* and POEMS promise to create a new basis for system performance engineering.

Although we have developed prototypes of all *Delphi* components, much work remains to integrate these components and validate the integrated system on both shared-memory parallel systems and computational grids. This is the challenge for the coming year.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[2] I. Foster and C. Kesselman. *Computational Grids: The Future of High-Performance Distributed Computing*. Morgan-Kaufmann, 1998.

[3] E. Johnson, P. Beckman, and D. Gannon. Portable Parallel Programming in HPC++. In *Proceedings, ICPP International Workshop on Challenges for Parallel Processing*, 1996.

[4] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irwin, K. L. Karavanic, K. Kunchitkapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, (11), Nov. 1995.

[5] D. A. Reed, R. A. Aydt, L. DeRose, C. L. Mendes, R. L. Ribler, E. Shaffer, H. Simitci, J. S. Vetter, D. R. Wells, S. Whitmore, and Y. Zhang. Performance Analysis of Parallel Systems: Approaches and Open Problems. In *Proceedings of the Joint Symposium on Parallel Processing (JSPP)*, pages 239–256, June 1998.

[6] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the High-Performance Distributed Computing Conference*, July 1998.

[7] E. Smirni and D. A. Reed. Workload Characterization of Input/Output Intensive Parallel Applications. In *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[8] A. Tamches and B. P. Miller. Instrumentation of Commody Operating System Kernels. In *Proceedings of the Third Symposium on Operating System Design and Implementation*, 1999.

[9] Z. Xu, B. P. Miller, and O. Naim. Dynamic Instrumentation of Threaded Applications. In submitted for publication, 1999.