Integrating a Debugger and a Performance Tool for Steering

Krishna Kunchithapadam
krishna@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI USA 53706

Abstract

Steering is a performance optimization idiom applicable to many problem domains. It allows control and performance tuning to take place during program execution. Steering emphasizes the optimization and control of the performance of a program using mechanisms that are external to the program. Performance measurement tools and symbolic debuggers already independently provide some of the mechanisms needed to implement a steering tool. In this paper we describe a configuration that integrates a performance tool, Paradyn, and a debugger to build a steering environment.

The steering configuration allows fast prototyping of steering policies, and provides support for both interactive and automated steering.

1 Introduction

Performance optimization is a non-trivial activity that programmers perform. The canonical performance optimization cycle (Figure 1) can be divided into four steps:

- *Instrumentation and Data Collection*: where programmers or tools instrument programs to collect performance data.
- *Analysis and Visualization*: where programmers use analysis and visualization tools to display and interpret the collected performance data.

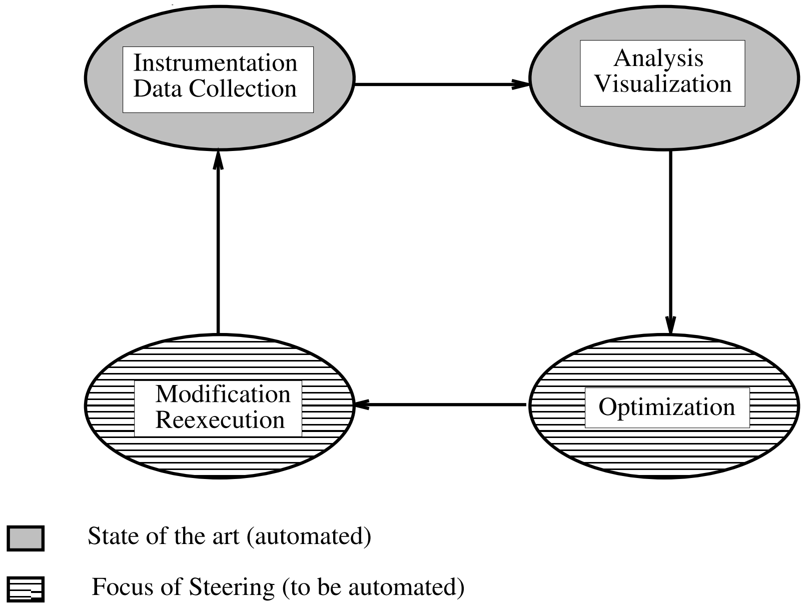

Figure 1: The Performance Optimization Cycle.

- *Optimization*: where programmers choose ways to improve the performance of their programs.
- *Modification*: where programmers make optimizations either to the source code or to running programs.

Many performance tools have been built to assist programmers in automating the steps of Instrumentation and Data Collection, and Analysis and Visualization. However, programmers for the most part currently use ad-hoc and manual techniques during the steps of Optimization and Modification. *Steering* is an idiom for performance optimization that focuses on the Optimization and Modification steps of the performance optimization cycle.

Steering is different from adaptive algorithms in that it is an idiom that emphasizes the *external* optimization and control of programs. In adaptive algorithms, programmers write optimization code that is linked with their application. At runtime, the optimization code makes performance optimization decisions and modifies the execution of the application.

In a technique that uses external control, the application does not contain code that makes optimization decisions. External mechanisms are used to both make optimization decisions and to modify the application itself to effect any changes.

The primary advantage of external optimization and control is that it allows programmers to prototype their optimization policies without going through the potentially long compilation-reexecution cycle each time the optimization code needs to be changed. External control also does not preclude the future inclusion of a suitable optimization policy into the application for production use.

Even when steering is the performance optimization idiom used by programmers, the mechanisms and tools used for steering are often not generic. In this paper, we show how to combine the mechanisms of a performance tool that does external measurement of programs and a debugger that provides external modification of programs into a steering tool that can be used for external optimization and control. In addition to the steering configuration, we also identify a programming model based on the notion of tunable performance knobs that allows a programmer to separate the functionality of an application from its optimization. The use of tunable knobs in an application can simplify the specification and even the automation of performance steering.

1.1 Goals of Steering

Steering includes the set of activities that are performed at program execution time and that modify the subsequent behavior of the program. Support from compilers and performance tools can simplify the programmer's task of steering a program. For example, compilers may generate data dependence information into symbol tables that a tool like a debugger can use in modifying a program. Information about compiler optimizations allows programmers to choose appropriate steering modifications for their programs. Performance tools provide a steering tool with performance data about a program; the steering tool can then use this information to initiate steering actions when needed.

The steering tool also needs the support of programmers to effect steering changes to an application. Since steering emphasizes external control, the programmer needs to indicate to the steering tool what parts of the program should be modified (and in what manner) to steer a program. Programmers also need to structure their applications so that steering actions can be specified to a tool in a generic manner. We design a programming model based on the notion of tunable performance knobs that allows both external control of an application's performance via a steering tool and the simple specification of steering actions using a predicate-action idiom.

In this paper, we present the design of a generic steering tool by making use of mechanisms already present in some performance measurement tools and in symbolic breakpoint debuggers. The mechanisms for steering need to provide support for *when* a program should be steered, *what* should be modified to effect the steering, and *how* the change should be made to the executing program. Performance tools can help answer the when and what questions while a debugger can help answer the how question. A steering tool that is built by integrating a performance tool and a debugger can be used for both interactive and automated performance steering.

A performance measurement tool is used in the Instrumentation and Data Collection, and Analysis and Visualization steps of the performance optimization cycle. The programmer uses the results of the analysis to decide on an optimization. The steering tool then invokes a symbolic debugger that allows the programmer to make modifications to an application to effect the steering changes.

A program needs to be suitably structured for a programmer to be able to steer its performance. The design of programs for steering (for example, through the use of tunable performance knobs) is part of our current research, but is outside the scope of this paper.

1.2 Examples of Steering

There are many examples of performance steering. Some of the techniques developed for steering are ad-hoc or domain-specific, while others are of wider applicability. In this section, we will illustrate the technique of steering using a small set of examples.

Loop convergence control: A simple example of performance steering is in the domain of numerical computations. An iterative algorithm may use a loop convergence criteria to terminate. If the convergence threshold can be changed to improve performance without affecting the correctness of the algorithm, then the program may dynamically change the convergence criterion for better performance. More importantly, if a large number of iterative algorithms are combined in a multi-stage computation, a faster convergence at an earlier stage even at the cost of a somewhat poorer solution may be compensated for by a later stage.

Load balance, degree of parallelism: Load balance is an important determinant of performance in irregular applications[2]. It may also be possible to modify the performance of an algorithm by changing the amount of parallelism that is used. A tool that allows a programmer to dynamically create or destroy threads and to move tasks between different threads can be used to improve the performance of the application.

Performance knobs in databases: Some database servers are designed to have a set of tunable performance knobs that control the allocation of resources in the server[1]. The server periodically executes a tuning algorithm to choose setting for the performance knobs (and hence resource allocations) that improves the performance of the server or satisfies some goal-requirements of the transactions that arrive at the server. This is an example where the application is structured to facilitate performance steering by the separation of the functional and performance code, and by the use of tunable knobs.

Array distributions: The performance of data-parallel programs is sensitive to the layout of array data across the nodes of a parallel machine[4]. Choosing good data layouts is non-trivial. A tool that measures array data related communication, presents such communication information to a programmer and accepts programmer hints for redistributing arrays into a more optimized layout performs steering. In this application domain, the steering tool needs support from the compiler and the runtime system. It is also possible for the steering tool to automate the process of choosing optimized data layouts using knowledge of the problem domain.

Changing algorithms: It is possible in suitably structured applications to change algorithms being used at runtime. For example, if a program invokes an algorithm via a function pointer, a steering tool can be used to change the function pointer to point to an function that is more suitable for the current performance context. Algorithms can also be changed by loading different dynamic libraries.

In the next section we describe the mechanisms present in the Paradyn performance tool that are relevant to performance steering. We then describe a steering configuration that uses the mechanisms of Paradyn and a symbolic debugger for performance steering.

2 Paradyn as a Performance Tool

Paradyn[5] is a performance tool based on a novel mechanism for automating the search for performance bottlenecks and low-overhead instrumentation techniques[3]. Paradyn is also extensible—any tool external to Paradyn may access the performance information that Paradyn collects via a well-defined *visualization* interface. The combination of the search mechanism, dynamic instrumentation, and the visualization interface allows us to use Paradyn as the performance measurement tool in a steering environment. This section provides a description of mechanisms of Paradyn that are relevant to performance steering.

2.1 The Metric-Focus Abstraction

Paradyn models an application as a collection of resources. Any program object for which performance data can be collected is a candidate resource. Examples of program resources include processes, machines, procedures, code blocks, variables, locks, barriers, message tags, memory, caches. Resources are arranged in hierarchies by resource type. For example, a resource type of *Procedure* contains resources corresponding to the different procedures in a program, and each procedure contains resources corresponding to different code blocks that comprise the associated procedure. Similarly, a resource type of *SyncObject* contains resource types *MessageTag, Barrier,* and *Lock*, each of which contain resources corresponding to the individual instances of message tags, barriers, and locks respectively. A *focus* is a set of resources, one from each resource hierarchy. An example of a focus is *message tag* 9999 *used in procedure* foobar *of process* 2345 *on machine* host12.

Performance metrics correspond to numeric values (in suitable units) that characterize some aspect of the performance of a program. Metrics may be computed for any valid focus. Examples of metrics include *CPU time* and *message count*.

Paradyn collects and provides performance data in *metric-focus* combinations. An example of a metric-focus pair is *message count for message tag* 9999 *used in procedure* foobar *of process* 2345 *on machine* host12. Metric-focus combinations are used by Paradyn to guide the search for performance bottlenecks. External tools also obtain performance data from Paradyn's visualization interface in terms of metric-focus lists. The steering tool is organized as an external visualization process that communicates with Paradyn, using the metric-focus abstraction to present performance data to a user.

2.2 Dynamic Instrumentation

Along with the complexity of characterizing the behavior of a large application comes the problem of measuring performance data that is needed to perform the above characterization. As the size and execution-time of a program increases, so does the volume of performance data that can be collected. It is often not possible nor profitable to collect comprehensive performance data about a program; it is difficult to store large volumes of data and to present them to a user in a suitable form. Moreover, any software technique for collecting performance data from an application perturbs the application—the greater the data collected, the larger the potential perturbation. Large instrumentation overhead may even result in performance data being collected in an environment that no longer matches that of the uninstrumented program, effectively rendering any performance measurement and analysis useless.

Paradyn uses *dynamic instrumentation* to reduce the volume of performance data that is collected, by postponing performance measurement till the demand for it arises. Dynamic instrumentation modifies the code and data segments of an executing program to collect performance metrics for foci. Since the instrumentation is made at execution time, programmers need not modify their source code, or use special compiler or linker options to build their binaries.

Paradyn uses dynamic instrumentation in conjunction with its automated search mechanism to focus instrumentation to restricted parts of a program. When a program starts to execute, dynamic instrumentation is used to collect very high-level performance information at a low volume and perturbation overhead. As the search along metric-focus pairs is refined, dynamic instrumentation is used to refine (by addition, deletion and modification) the instrumentation in the application to collect detailed performance data, but for a smaller program focus. The combination of a refined focus and detailed performance data keeps the instrumentation overhead and volume of performance information under control.

Finally, dynamic instrumentation can also measure its own overhead. This instrumentation overhead is presented as a performance metric to Paradyn's search mechanism. This allows the search process to control the amount of instrumentation added to a program (and the associated overheads) using a cost model.

A steering tool uses dynamic instrumentation to enable the collection of performance data needed to trigger steering activities, and to disable the collection of the data at the end of a steering session.

2.3 The Visualization Interface

Performance data that Paradyn (or any performance tool) collects can be used in many ways. Paradyn uses the data in conjunction with its search mechanism to search for performance bottlenecks. The same performance data can be visualized in different ways. Rather than provide a fixed set of visualizations, Paradyn provides access to the performance data it collects via the *visualization interface*. Any external process that wishes to consume performance data (not necessarily for visualization) can initiate the collection (via dynamic instrumentation) of performance data for valid metric-focus lists using this interface.

A steering tool can therefore use this interface to prototype steering policies without having to modify any code in Paradyn. Such an interface also imposes minimal restrictions on the structure of the steering tool.

3 The Steering Configuration

The steering configuration that we propose integrates the mechanisms of Paradyn and a debugger. Paradyn is used to collect performance data; optionally the steering tool can also use the performance search mechanisms of Paradyn during steering. The command interface of the debugger is used to make changes to a program and effect steering changes. During a steering session, there is a transfer of control between Paradyn and the debugger. The steering tool manages the control transfer, but the user of the steering tool needs to specify to the steering tool when to transfer control from Paradyn to the debugger. The user hints can be coded into the steering tool or be specified as a set of rules that are interpreted by the steering tool. The dynamic instrumentation interface of Paradyn requires a small modification to allow a debugger to attach to a program and perform steering changes.

Figure 2 describes the steering configuration.

3.1 Control of the Steering Tool

The steering configuration above does not include any policies for steering an application; it is a collection of mechanisms that can be used by the programmer. The simplest use of the steering tool is for interactive steering. However, the same configuration can also be used for automated steering.

In interactive steering, Paradyn measures the performance of an application and searches for types of bottlenecks specified by the programmers. When a performance bottlenecks is detected, the steering tool alerts the programmer and transfers control to the debugger. The programmer can then make steering changes to the application and transfer control back to Paradyn. Alternatively, the programmer can bypass the search mechanisms of Paradyn and write per-

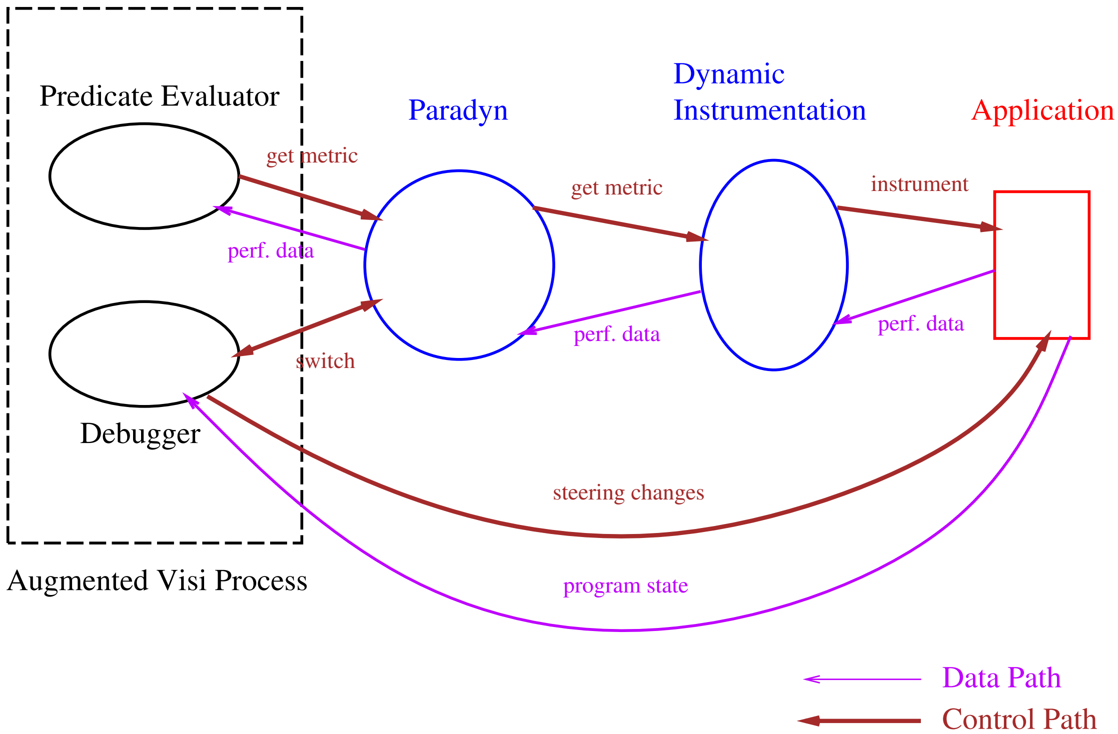

Figure 2: The Steering Configuration.

formance search strategies into the steering tool using the performance data available via the visualization interface of Paradyn.

In automated steering, the programmer provides hints to the steering tool about what constitutes a performance problem, and how changes should be made to the application (i.e. what debugger commands should be invoked) to improve performance. The steering tool can use any algorithm to detect when the performance problems specified by the programmer occur and then automatically invoke the steering actions.

4 Conclusions

Performance steering is a runtime-based performance optimization idiom that is being used in many application domains that emphasizes the external optimization and control of programs. A steering tool needs mechanisms to determine when a program should be steered, what needs to be changed, and how the steering should be effected. Performance measurement tools and debuggers already independently provide some of these mechanisms.

Performance measurement tools provide performance data and help answer the *when* question. Debuggers provide commands to modify the state of an executing program. Debugging commands can be used to answer the *how* question. Programmer hints specify what needs to be changed in a program to effect steering.

We show a steering configuration that combines the use of Paradyn as the performance measurement tool and any symbolic debugger as the modification tool. We also suggest a programming model based on tunable performance knobs that allows steering actions to be specified in a generic manner and simplifies the task of steering programs.

References

[1] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards Automated Performance Tuning for Complex Workloads. In *Proceedings of the 20th International VLDB Conference*, Santiago, Chile, September 1994.

[2] G. Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon-toward interactive parallel programs: The online steering of a molecular dynamics application. In *Proceedings of the 3rd International Sym-*

posium on High-Performance Distributed Computing, San Francisco, CA, August 1994.

[3] Jeffrey K. Hollingsworth, Jon Cargille, and Barton P. Miller. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proceedings of the 1994 Scaleable Highe Performance Computing Conference*, pages 841–850, Knoxville, TN, May 1994.

[4] Krishna Kunchithapadam and Barton P. Miller. Optimizing Array Distributions in Data-Parallel Programs. In A. Nicolau, D. Gelernter, D. Gross, and D. Padua, editors, *Languages and Compilers for Parallel Computing, LNCS*. Springer-Verlag, 1994.

[5] Barton P. Miller, Jeffrey K. Hollingsworth, R. Bruce Irvin, Jonathan Cargille, Krishna Kunchithapadam, Karen Karavanic, Tia Newhall, and Mark Callaghan. The Paradyn Performance Measurement Tools. In *Review for the IEEE special issue on Parallel and Distributed Systems*, October 1994.