The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools

Luiz DeRose
laderose@us.ibm.com

Advanced Computing Technology Center
IBM Research
Yorktown Heights, NY 10598 USA

Ted Hoover Jr.
hoov@us.ibm.com

Tools Development Dept.
IBM
Poughkeepsie, NY 12601 USA

Jeffrey K. Hollingsworth *
hollings@cs.umd.edu

Computer Science Department
University of Maryland
College Park, MD 20742 USA

Abstract

The complexity of both parallel architectures and parallel applications poses several problems for the development of performance analysis and optimization tools. In this paper, we describe the motivations and the main aspects of the design of the Dynamic Probe Class Library (DPCL), an object based C++ class library that provides an infrastructure to reduce the cost of writing instrumentation for performance tools. Additionally, we present some of the performance tools built on top of DPCL, which demonstrate the power and flexibility of the library.

1 Introduction

As parallel architectures become more complex, due to deep-memory hierarchies, clustered SMPs, and more intricate distributed interconnects, application developers face new and more complex performance tuning and optimization problems. Moreover, the sensitivity of parallel system performance to slight changes in application code, together with the large number of potential application performance problems (e.g., load balance, data locality, or input/output) and continually evolving system software, make application tuning complex and often counter-intuitive. Thus, it is not surprising that users of parallel systems often complain that it is difficult to achieve a high fraction of the theoretical peak performance, and are constantly asking for more application performance analysis tools. However, also due to the complexity of parallel systems, programming tools are becoming expensive to build and maintain. Moreover, the available tool development resources are shrinking rapidly. In summary, more performance tools are needed, but fewer tools can be created.

When observing the costs associated with development of performance tools, it is clear that one of the major costs is in developing the instrumentation. The specific cost of developing the instrumentation depends on the type of instrumentation chosen and has to be played against the user effort required to use the tool, but in general, writing the instrumentation package requires substantial interaction across different system components, such as operating system, run-time libraries, and compilers. This interaction requires substantial communication, coordination, and cooperation across development groups, which tend to be expensive. The exact fraction of the cost will vary between tools, but often the direct and indirect costs of writing the instrumentation software dominates the cost of the tool.

In order to address these issues, IBM decided to invest resources into building a general-purpose infrastructure that flexibly supports the generation of arbitrary instrumentation. This infrastructure allows the reuse of large portions

*Jeff Holingsworth has been visiting the Advanced Computing Technology Center at IBM Research on Sabbatical for the past five months.

of the instrumentation system, such that the cost of writing performance tools could be reduced. In this paper we present the *Dynamic Probe Class Library (DPCL)*, an object based C++ class library that provides the necessary infrastructure to allow tool developers and sophisticated tool users to build parallel and serial tools through a technology called *dynamic instrumentation*. DPCL takes the basic components needed by tool developers and encapsulates them into C++ classes. Each of these classes provides the member functions necessary to interact and instrument a running application with software patches called probes.

The remainder of this paper is organized as follows: In §2, we begin by discussing the motivation for the development of DPCL, the dynamic instrumentation approach, the Dyninst project, and the main components of DPCL. In §3 we present some of the prototype tools being developed using DPCL. In §4 we address some issues regarding current and future work with DPCL. Finally, §5 summarize our conclusions.

2 Dynamic Probe Class Library

2.1 Motivation

DPCL evolved from the application development needs of the high performance computing community. In the past the development of parallel application tools such as debuggers and performance tools have traditionally focused on extending the serial development model to very large, highly scalable, parallel applications. In most cases, each solution built a unique infrastructure to support the requirements of the tools being developed. Over time, the effort of maintaining multiple unique tool infrastructures exceeded the effort of developing new features within the tools. A new approach was needed.

A second motivation was the realization that many of the techniques used in the debugging and tuning of serial applications did not scale to the dimensions required by the HPC community. Debugging using just simple print statements embedded within code can easily generate large quantities of data on a massively parallel application. An approach was needed that allowed data to be gathered from an application that was more flexible than the traditional approach. This requirements lead to the adoption of dynamic instrumentation technology being developed at the University of Wisconsin [4] and University of Maryland [1]. Dynamic instrumentation provides the flexibility for tools to insert probes into applications as the application is running and only where it is needed.

The reality is that development of performance tools for HPC is extremely hard. The compiler technology and programming models change constantly, the entry point for developing tools is too high for Independent Software Vendors to develop competitive offerings, and the investment in tools is too low to solve the problem. Thus, a common approach was needed to support a wide variety of tools that could be powerful, robust, and scalable.

DPCL represents an initial solution to the problem. By using DPCL as a foundation for tools development, the tools community can build a common tool infrastructure and reduce the time to market for new tools. An added benefit is in the portable nature of the tools that can be developed. This will allow tools builders to adapt more quickly to the changing technology and programming models.

DPCL is beneficial to developers and users at several levels. First, tool developers can focus on developing tools that address the requirements that they are trying to fulfill, without having to worry on developing the instrumentation infrastructure. Second, tools researchers can focus on the research by using the DPCL infrastructure to build prototypes quickly. A desirable side effect of reducing instrumentation cost is that it becomes cost effective to experiment with more speculative analysis techniques. Novel and innovative ideas can be evaluated inexpensively. Thus, trial and error is less costly. Third, users that need specialized tools can also benefit of DPCL by having an infrastructure to develop the required tools in-house with less effort. Finally, tool users will ultimately have more choices of better and portable tools.

2.2 Dynamic Instrumentation

Traditionally, instrumentation systems have had to either optimize their instrumentation for minimal overhead or maximum information. To keep overhead low, few events can be recorded and the amount of information per event must also be kept low. However, this could mean that key events may not have been recorded. Likewise, if too much instrumentation is inserted, the overhead may be so high that it is no longer representative of the un-instrumented program's execution behavior.

Also, many instrumentation systems require that programs be re-compiled to be instrumented. While this is generally possible, for large applications it can be time consuming. Even worse, for third party libraries and applications users where the source code may not be available, re-compiling will not be possible.

An alternative is to allow a program to be modified while it is executing, and not have to re-compile, re-link, or even re-execute the program to change it. This dynamic instrumentation approach provides several practical benefits compared with traditional static compilation. For example, if we are measuring the performance of a program and discover a performance bottleneck, it might be necessary to insert additional instrumentation into the program to understand the problem.

Dynamic instrumentation allows flexibility in gathering data, which can be used to focus attention on specific items of interest, increase accuracy by reducing interference caused by gathering unwanted data, or increase convenience to the user by delaying the decision point for instrumentation until run-time. With dynamic instrumentation the instrumentation code need only to reside in the application as long as it is needed to gather data. When a problem is suspected the instrumentation can be inserted into the application to gather data needed to verify the problem. Once the problem is verified the instrumentation can be replaced with more detailed instrumentation to establish the cause of the problem. If the initial guess turns out to be incorrect, the original instrumentation can be replaced with new instrumentation that examines other possible causes.

2.3 Dyninst

The Dyninst Application Program Interface (API) permits the insertion of code into a running program [1]. Using this API, a program can attach to a running application, create a new bit of code and insert it into the application. The program being modified is able to continue execution and doesn't need to be re-compiled, re-linked, or even restarted. The next time the modified program executes the block of code that has been modified; the new code is executed in addition to the original code. The API also permits changing subroutine calls or removing them from the application program.

Runtime code changes are useful to support a variety of applications including debugging, performance monitoring, and supporting the composition of applications out of existing packages. Depending on the use, the code can either augment the existing program with ancillary operations such as measuring the application performance or adding additional print statements, or alternatively, it can be used to alter the semantics of the program by changing the subroutines executed or manipulating application data structures. The second type of change is most useful for either performance steering, or other debugging applications. The API supports both of these uses.

To allow the benefits of runtime code modification to be made available to a broader user community, the API provides a machine independent interface to binary modification. Traditionally, post compiler instrumentation tools have provided interfaces that allow machine or assembly language level code to be inserted. Instead, the Dyninst interface is more analogous to a machine independent intermediate representation of the instrumentation as an abstract syntax tree. This allows the same instrumentation code to be used on different platforms. For example, consider instrumentation code to monitor the behavior of a database system (i.e., tracking commit and abort operations). The instrumentation code would be specific to the particular database system, but because the instrumentation is machine independent, it would work with any machine architecture where the database system was installed.

2.4 DPCL

One of the goals of DPCL was to build upon the base function provided by the Dyninst API and extend it using a robust scalable design. This required the design of DPCL to consider the following requirements:

- Provide an infrastructure that allows tools to probe single process applications up to very large MPI applications running across 4096 processors.
- Provide a secure infrastructure.
- Provide thread safe instrumentation to support threaded processes.
- Provide multiple probe types to support a variety of tools.
- Provide the ability for probes to communicate back to the tool.
- Provide support for C, C++, and Fortran applications.

The Dynamic Probe Class Library (DPCL) is a value-added layer built on top of the Dyninst API. While Dyninst provides a good substrate for tools development, its focus is on modifying a set of running processes on a single machine. The DPCL layer adds additional support for multiple nodes in a parallel machine, a security layer to authenticate inter-node instrumentation requests, and a data transport layer to gather data from nodes in a parallel computer and provide them to a single front-end process.

While Dyninst works on multi-processor machines since it uses the native platform's debugger interface (ptrace or procfs) it does not allow requests to cross between nodes in a parallel computer. Likewise, it lacks support for identifying the processes from separate nodes that are part of a parallel job. DPCL provides features that allow it to interact with the Parallel Operating Environment (POE) on IBM SP systems to identify the processes of a parallel job (the user simply needs to identify the process id the front-end processes associated with the job). Once it has this information, it starts instrumentation processes on each node that is running a process from the parallel job.

There are also differences in the security models between Dyninst and DPCL. Dyninst relies on the host operating system's security for ptrace and procfs system calls. These calls restrict debuggers to only be attached to processes owned by the same user as the one running the debugger. However, with DPCL's multi-node feature, it is necessary to add additional security. DPCL extends the security

model provided by Dyninst by extending security requirements across SP system or cluster. DPCL must ensure that DPCL based tools cannot create or connect to parallel applications that are not owned by the tool user. Multiple levels of security are supported by conforming to the level of security that is enforced within the system. DCE based security is the strongest level of security provided. If the SP system is configured using DCE the DPCL daemons must acquire the DCE credentials of the tool user before being able to instrument processes. This is done when the DPCL client first establishes a connection to a remote node. The DPCL super daemon initiates a secure conversation with the DPCL client library to forward the user credentials. Upon receiving and validating the user credentials, the DPCL super daemon will spawn a DPCL daemon using these credentials. This allows the user to only access the processes that are owned by the tool user. If the SP system is not configured to enforce usage under DCE security DPCL will provides a level of simple user authentication to processes on remote nodes.

The third value-added component of DPCL is a data transport interface to move data gathered on each node to the front-end node. Using this interface, a tool developer can write instrumentation code that makes a simple library call to send data, and the DPCL system will handle buffering the data, and moving it to the front-end node.

2.4.1 DPCL Structure

DPCL is a C++ library that encapsulates the client server infrastructure needed to manage large parallel applications (See Figure 1). It consists of a client library from which end user tools can be created, a run time library that is used by DPCL for instrumentation and communication, a daemon that interfaces with the Dyninst library to instrument and manage user processes, and a super daemon to manage security and client connections to the DPCL daemons.

The client library provides a set of C++ classes that allows tools to connect, examine, and instrument single processes and large applications. This is done through the creation of probes to be inserted at set of possible instrumentation points that include function entry points, exit points, and call sites. Three different types of probes are available:

Point probes: probes that are inserted at available instrumentation points within the application and executed when the point is reached during processes execution.

One shot probes: probes that are executed at the current point in time regardless of where the process is executing

Phase probes: probes that are associated with a phase timer in which the probe is executed each time the timer expires regardless of where the application is running.

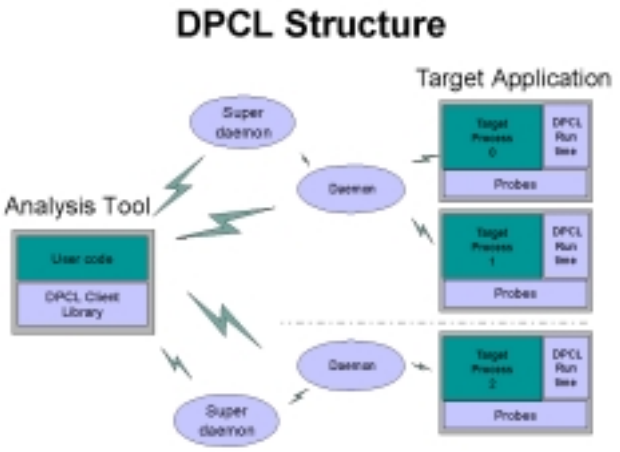

Figure 1. DPCL Structure

For more complex probes a "ProbeModule class" is available to dynamically load external modules into the application and to be called from other probe expressions.

The role of the Super Daemon is to establish secure connections to the DPCL daemons. There is one Super Daemon per server node and used by all DPCL based tools connecting to that node.

The DPCL daemon performs multiple roles. First, it interfaces with the Dyninst library to directly instrument processes. It also manages and forwards any probe communications back to the client library in addition to extending Dyninst capabilities by allowing probe modules to be dynamically loaded. There is one DPCL daemon created per user on each server node.

An additional capability provided by DPCL is the ability for probes to send messages back to the end user tool. When probes are built using probe expressions or written in the form of probe modules, calls to a special DPCL function called "Ais_send()" can be defined that allow any probe to send a message back to the tool. When probes are inserted into an application the tool has the option of defining a call back function to receive these messages. This allows tools to optionally use data that is collected from within the application as the application is running.

2.4.2 Scalability

The scalability requirements of DPCL required the design to consider how to manage a potentially large set of processes in such a way that is efficient and would not generate excessive memory requirements on end user tools developed using DPCL. Consider the case of a tool that is designed to gather a hardware counter profile from each of the functions within a SPMD style application consisting of 128 MPI tasks. A tool would potentially need to collect all the

instrumentation points representing each function entry and exit points across the entire application and subsequently insert probes at each of the points. The number of points to be collected could easily number in the thousands requiring the tool to maintain large data structures to manage all the points and the probes after they have been inserted.

DPCL was designed to allow tools to intelligently manage scenarios such as this. First, when DPCL examines the source structure of the application it only looks at each process down to the module level. If the tool needs to examine the source structure below the module level it must "*expand*" the module to retrieve the additional information. This significantly reduces the amount of information forwarded to the client side of the DPCL. This also allows the tool to incur the overhead of extracting low-level source structure and instrumentation points for only the modules it is interested. Second, DPCL recognizes that the application being instrumented is SPMD by only maintaining one copy of the source object tree representing the process on the client side. This requires the tool to look at only one copy of the source object tree of the 128 processes to find valid instrumentation points. Instrumentation points found within the tree can be used across any of the processes in the application. Next, instead of adding probes to each of the processes within the application, a single instance of the application object can be used to apply the probes across the entire application using a single call to the application object member function.

In summary, DPCL was designed from the beginning to be scalable and extendable. This allows DPCL to support a large variety of tools and allow it to be easily extended to support additional tool requirements over time.

3 DPCL Based Tools

One of the DPCL goals is to increase innovation in tools development. In this section we demonstrate the flexibility provided by DPCL, by briefly describing some of the tools being developed or extended with DPCL. We first present an update to the classic print statement as a debugging tool. Next, we describe a dynamic profiler tool, followed by a graphical user interface for dynamic instrumentation. Finally, we discuss the interface of an existing tool (Paraver) with DPCL.

3.1 Dynamic Printf "a Modern Alternative"

It is a well-known fact that print statements are still the most widely used debugger technique. Thus, one of the first example tools ever developed using DPCL was the *Dynamic Printf*, which demonstrates DPCL's flexibility, in addition to showing users that use print statements as their primary technique in debugging and tuning applications a modern alternative. The purpose of the tool is to take an application that is running and allow the user to dynamically insert and remove print statements into an application to examine the values of variables. This was done to save the user from stopping the application, updating the source code to add printfs, recompiling, and then restarting the application only to find that the wrong values were printed.

Although it is provided as a demonstration sample, it highlights some key features of DPCL. First, the tool can attach to a running application and examine its source structure object hierarchy. Application variable objects (called *Data Objects*) are contained within the object hierarchy. Data objects can be used in probe expressions to pass references to these variables into probe module functions. By inserting probes into the application, variable values are printed eliminating the need to hard coding print statements.

3.2 DynaProf

DynaProf is a dynamic profiling tool for serial, MPI, OpenMP, and mixed mode programs. It provides a command line interface, shown in Figure 2, similar to gdb. Dynaprof allows users to install probes at selected functions call entry and call exit points, in order to provide inclusive and exclusive profiling data, as well as one level inclusive call tree information.

Dynaprof is modular in that the instrumentation it inserts has a well defined format, much like loadable kernel modules on AIX and Linux. Users can start a new application or attach to a running application. Currently, three probes are provided, one for wall clock profiling of functions, one for hardware counter profiling of functions, and one for statistical sampling of the program counter on the basis of hardware counter or cycle counter overflows.

3.3 Graphical User Interface for Dynamic Instrumentation

DPCL is currently being used to develop tools that give end users control on how probes are inserted into applications. By providing visual representations of parallel applications and their source structure in a scalable way, users would be able to have direct control on where the probes are placed in their application. Figure 3 shows this visual representation. The graphical user interface is divided in three main panes. The left pane displays the process list information and allows the user to select the processes to be instrumented. The middle pane shows the application source tree, for selection of the functions to be instrumented. Finally, the right pane is used for probe selection.

Adding a probe to an application is defined by intersection of the selection of set of processes to be instrumented, selection of an object in the source structure hierarchy, and

```
(dynaprof) help
gdb         Run gdb in the current directory: gdb [args].
make        Run make in the current directory: make [args].
load        Load an executable: load [exe [args]].
attach      Attach to a running executable: attach [exe pid1].
poeload     Load a POE application: poeload [exe [args]].
poeattach Attach to a running POE application: poeattach [pid␣of␣poe].
use         Load instrumentation code into the process: use ¡module¿.
run         Continue/Run the instrumented process.
info        Display information about the process.
list        Print info about the executable, modules or functions:
                list [modules—functions] [pattern].
instr       Instrument functions with current probe:
                instr [¡modpat¿—¡module funpat¿].
quit        Abort the current process and exit.
help        Display this text.
?           Display this text.
(dynaprof)
```

Figure 2. DynaProf help command

Figure 3. A graphical user interface for probes placement

the probe being selected. In this case the probe selection may not represent a single probe to be inserted at single instrumentation point within the application, but potentially a set of probes that are inserted as a group to perform a single action. For example: If a user first selects a single task within a multi task MPI job, then selects the source object that represents a module within the application, and finally one of the hardware counter probes, the result of adding a probe is that all the functions contained within the module on the selected task will have a counter probe inserted at the entry and exit points. By providing a higher-level view, the tool can free the user from having to search for individual instrumentation points.

3.4 Paraver

In addition to developing new DPCL based tools, we are also collaborating with tools developers to integrate the DPCL technology into existing tools. One such example is

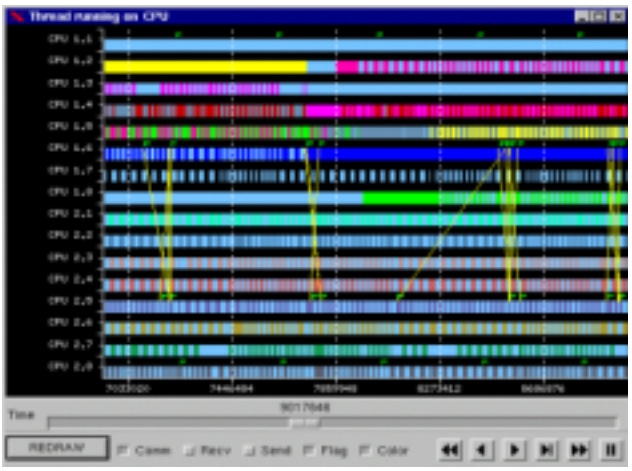

Figure 4. Performance trace visualization of a mixed mode application with Paraver

the collaboration between the Advanced Computing Technology Center (ACTC) at IBM Research with the European Center for Parallelism in Barcelona (CEPBA) for the integration of DPCL into Paraver [2], a flexible parallel programming and visualization tool for performance analysis of sequential and parallel applications (see Figure 4).

The Paraver tracing environment uses DPCL for instrumentation of MPI, OpenMP, and mixed mode programs. In message passing applications all MPI calls are instrumented, while in OpenMP programs, DPCL is used to instrument the start and end of all parallel regions. Additionally, in both cases, the enclosing user functions are also instrumented, and the user can also provide a file with function names for dynamic instrumentation. The Paraver tracing environment uses the DPCL instrumentation to collect detailed quantitative data of the program performance, including hardware counters information.

4 Future Work

4.1 Debugger Support

The version of DPCL that has currently been released provides good support for developing measurement and monitoring tools. However, to implement debuggers, some additional features are required. In particular, information about mapping line numbers to instructions, variable information (names of local variables, their location in memory and type), and fine-grained instrumentation are needed. Compilers provide the first two types of information when they are invoked with the debugger flag. All that is required is that DPCL read this information from the binary, and provide an interface to tool users. Currently, the representation

of this information in executables varies between platforms and even among compilers on the same platform. By providing a common interface to this information, it is possible for people to write portable debuggers.

Additionally, the instrumentation granularity of DPCL needs to be improved to allow arbitrary instrumentation points. Currently, it is possible to instrument procedure entry, exit and call sites. However, for debugging (such as a conditional breakpoint), we need to be able to insert instrumentation code at any location in the application. By providing a common interface to this information, it is possible to write portable debuggers.

4.2 Open Source

DPCL is distributed as part of IBM's Parallel Environment for AIX as a licensed program product. The next step in its development is to make it available under an open source license to allow it to evolve under guidance of a broader development community. To fulfill its goal as a tool development platform it must be able to support tools across a wide variety of hardware and operating system. Establishing an open source community will enable tool developers and researchers to share the benefits and costs of extending the capabilities of DPCL, while protecting their tool development investments. Work is being done to host a web based collaborative open source project as part of the IBM developerWorks Open Source Zone.

The first area currently being addressed is multi-platform support. Work is underway to port DPCL to Linux to improve the tools development capabilities on Linux PCs and clusters. Areas that need to be addressed include support to the ELF/DWARF style executables, in addition to supporting open standards for additional parallel application execution environments such as MPICH [3] from Argonne National Laboratory.

4.3 DPCL - Dyninst compatibility

Another area being addressed is the convergence of multiple Dyninst implementations that currently exist. When DPCL was originally developed as a prototype it was based on the University of Maryland version of Dyninst. To enable some of the extended features required by DPCL new features were added to the Dyninst API and developed on an independent base from the University version. Although explicit steps were taken to maintain compatibility between the two versions of the API over time, each version evolved in slightly different directions. Work is underway to address any remaining compatibility issues to allow DPCL to be ported to additional platforms using the University version of Dyninst as its base.

5 Conclusions

In this paper, we described the motivations and the main aspects of the design of the Dynamic Probe Class Library (DPCL), an object based C++ class library that provides the necessary infrastructure to reduce the cost of writing instrumentation for performance tools. DPCL allows the dynamic instrumentation of serial, shared memory, and message passing applications, requiring only the information found on the executable.

In addition to reducing the cost of tools development, because DPCL is based on dynamic instrumentation technology, it also reduces the intrusion cost of instrumentation, increases flexibility and usability of tools, and provides a mechanism for interoperability among tools.

References

[1] B. R. Buck and J. K. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 1994.

[2] European Center for Parallelism of Barcelona (CEPBA). *Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual*, November 2000. http://www.cepba.upc.es/paraver.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message-Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[4] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.