Final Technical Report: Building a Community Infrastructure for Scalable On-Line Performance Analysis Tools around Open|Speedshop

September 2009 to September 2013

SC0002155 (PRJ27NS)

Barton P. Miller

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
bart@cs.wisc.edu

1 Technical Accomplishments

Peta-scale computing environments pose significant challenges for both system and application developers and addressing them required more than simply scaling up existing tera-scale solutions. Performance analysis tools play an important role in gaining this understanding, but previous monolithic tools with fixed feature sets have not sufficed. Instead, this project worked on the design, implementation, and evaluation of a general, flexible tool infrastructure supporting the construction of performance tools as "pipelines" of high-quality tool building blocks. These tool building blocks provide common performance tool functionality, and are designed for scalability, lightweight data acquisition and analysis, and interoperability. For this project, we built on Open|SpeedShop, a modular and extensible open source performance analysis tool set.

The design and implementation of such a general and reusable infrastructure targeted for petascale systems required us to address several challenging research issues. All components needed to be designed for scale, a task made more difficult by the need to provide general modules. The infrastructure needed to support online data aggregation to cope with the large amounts of performance and debugging data. We needed to be able to map any combination of tool components to each target architecture. And we needed to design interoperable tool APIs and workflows that were concrete enough to support the required functionality, yet provide the necessary flexibility to address a wide range of tools.

A major result of this project is the ability to use this scalable infrastructure to quickly create tools that match with a machine architecture and a performance problem that needs to be understood. Another benefit is the ability for application engineers to use the highly scalable, interoperable version of Open|SpeedShop, which are reassembled from the tool building blocks into a flexible, multi-user interface set of tools. This set of tools targeted at Office of Science Leadership Class computer systems and selected Office of Science application codes.

This was a collaborative project with Krell Labs (Jim Galarowitz, PI). the University of Maryland (Prof. Jeffrey Hollingsworth, PI), Oak Ridge National Laboratory (Dr. Philip Roth, PI), and Lawrence Livermore National Laboratory (Dr. Martin Schulz, PI). Below, we describe the contributions made by the team at the University of Wisconsin. The project built on the efforts in Open|SpeedShop funded by DOE/NNSA and the DOE/NNSA Tri-Lab community, extended

Open|Speedshop to the Office of Science Leadership Class Computing Facilities, and addressed new challenges found on these cutting edge systems.

Work done under this project at Wisconsin can be divided into two categories, new algorithms and techniques for debugging, and foundation infrastructure work on our Dyninst binary analysis and instrumentation toolkits and MRNet scalability infrastructure. Note that in the infrastructure area, this project shares common development with several of our other funded DOE Office of Science Efforts. Therefore, work done in that area (Section 2 of this report) is also reported in those Progress Reports.Below we present several areas in which we have made technical contributions. The publication list (Section 3) is a more complete representation of our work.

1.1 Binary Rewriting on IBM BlueGene P and Q

We continue to work on supporting static binary instrumentation (known as *binary rewriting*) under Dyninst. Our recent work includes supporting both statically and dynamically linked executables, porting rewriting to work on the IBM BG/P. Static executables are important as they are frequently used on systems such as IBM BlueGene and Cray XT. They the operating system to deliver a binary to the compute nodes with no additional library or other dependences.

We now handle both 32 and 64 bit executables on both IBM Linux and BlueGene compute nodes. The compliers and loaders on BG are different from Linux, so this adds an extra dimensions of complexity.

1.2 Instrumenting Difficult Binaries

A major part of a program's behavior is its interaction with the operating system. Knowing how the program interacts with the operating system can significantly increase the analyst's ability to understand the semantics, intent, and performance of the program, and is of particular importance in malware analysis. Most programs do not directly invoke the system calls that define the operating system interface. Instead, they call wrapper functions provided by standard system libraries; such wrapper functions invoke the required system call or calls, often through the use of trap instructions. We developed a robust library fingerprinting technique that identities wrapper functions based on their interaction with the system call interface. Library fingerprinting restores names to library code that has been statically linked into program binaries by identifying recognizable characteristics *fingerprints* of standard library functions. A direct benefit of identifying wrapper functions is that we help to understand and add meaning to complex functions that call these routines.

Existing library fingerprinting techniques use simple pattern matching to identify functions. For example, the widely used IDA Pro disassembler stores patterns similar to byte-level regular expressions that it has derived from existing libraries. These patterns can be used only to find library code that is nearly bytewise identical to the library from which the patterns are derived; such approaches are brittle in the face of code produced by different compiler versions or build options, and do not generalize well across library versions. Our goal is to generate patterns that are tolerant of these naturally occurring binary differences. There is an essential tension, however, between specificity and generalizability: signatures must capture the details that differentiate functions, but must abstract away minute differences between versions.

We introduced semantic descriptors that capture the high-level semantics of wrapper functions using the characteristics of system call invocations. Our approach yields two contributions. First,

we use semantic descriptors to create fingerprints for wrapper functions; these fingerprints, based on the essential, invariant characteristics of system calls, can generalize across different library versions and compiler variations. Second, we define a extendible pattern matching algorithm that identities specific wrapper functions based on library fingerprints. We use our technique to form fingerprints for the GNU C Library (glibc) and to restore wrapper function names to stripped program binaries, which informs our high-level work flow:

1. We obtain binaries for several versions of the glibc library as a reference set; these binaries are obtained from several Linux distributions, and are compiled using several compiler toolchains. The code variations among these libraries allow us to determine the generalizability of semantic descriptor-based fingerprints.
2. Using the ParseAPI parsing library, we extract an instruction representation and control ow graph (CFG) for each exported function in a particular glibc binary. Exported functions define the public interface of the GNU C Library, where we anticipate behavioral stability across library versions. During parsing, we identify wrapper functions by recording direct invocations of system calls.
3. For each wrapper function, we construct a semantic descriptor based on the invariant characteristics of the invoked system call(s), and record the function fingerprint. Invariant characteristics include the system call name and any concrete parameter values.
4. To identify wrapper functions in program binaries, we search for functions that directly invoke system calls and construct a semantic descriptor; we then compare the descriptor against a database of library fingerprints. The fingerprint matching algorithm is a two-stage process that tolerates slight variations in fingerprints.

We have implemented semantic descriptor-based library fingerprinting as an extension to unstrip, a tool that discovers functions in stripped binaries and adds generic names to the symbol table. Our extensions allow unstrip to add meaningful names to GNU C library wrapper functions that are discovered in stripped binaries. The benefit of unstrip is that it creates a new binary with a symbol table, so any tool that depends on a symbol table will be able to leverage this information.

1.3 Accurate Attribution of Contributions in Code Repositories.

Information as to who wrote a given piece of code, authorship, is used to analyze software quality, perform software forensics, and improve software maintenance. Previous tools approximated line level authorship by assuming that the last person to change a line was its author, while ignoring all earlier changes. In this research, we showed how to mine a code repository for the development history of a line of code to assign contribution weights to multiple authors. Using these contribution weights, we can attribute a line to the most responsible author in binary code forensics, directly apply the weights to model source code familiarity, and trace back to earlier commits to determine when bugs were introduced in software quality analysis. Our new techniques abstract code repositories as a graph representing the development dependencies between commits. We perform a backward flow analysis based on the results of an enhanced line differencing tool between adjacent commits to extract the development history of a line of code. We used the history to attribute each character of the line to the responsible author and assign contribution weights. We then implemented this new functionality as an extension to git.

The methods used by previous tools (git-blame, svnannotate and CVS-annotate) for obtaining line level authorship loses information. A line of code may have been changed multiple times by differ-

ent developers to fix bugs, to conform to interface changes, or to tune parameters. These changes compose the history of a line of code. For each line of code, current tools report the last commit that changed the line and the author of that last commit. These tools take the last snapshot, while missing the earlier stages of the development history. Therefore, even when the last commit changes only a small fraction of a line of code, the author of the last commit still is credited for the entire line.

In this project, we defined the r*epository graph, structural authorship,* and *weighted authorship* to help overcome these limitations. The repository graph is a directed graph representing our abstraction for a code repository. In the graph, nodes are the commits and edges represent the development dependencies. For each line of code, we define structural authorship and weighted authorship. Structural authorship is a subgraph of the repository graph. The nodes consist of the commits that changed that line. Development dependencies between the subset commits form the edges. Weighted authorship is a vector of author contribution weights derived from the structural authorship of the line. The weight of an author is defined by a code change measure between commits, for example, best edit distance. We use these two models to extract the development history of a line of code and derive precise line level authorship.

To evaluate our new models, we implemented structural authorship and weighted authorship as a new git built-in tool: *git-author.* We conducted two experiments to show how often the new models will produce more information and whether this information is useful for analysis tools that are based on code authorship information. In the first experiment, we ran git-author over the repositories of five open source projects and found that about 10% of the lines were changed by multiple commits and about 8% of the lines were changed by multiple authors. Analysis tools lose information on these lines when they use the current methods for line level authorship. In the second experiment, we used git-author to build a new line-level bug prediction model. We compared our line-level model with a representative file-level model on our data sets derived from the Apache HTTP sever project. The results show that the line-level model performs consistently better than the file-level model when evaluated on effort-aware metrics.

This project made the following contributions:

1. The structural authorship model that extracts the development history of a line of code and overcomes the fundamental weakness of current tools.
2. The weighted authorship model that assigns contribution weights to each change of the line and produces precise line-level authorship attribution.
3. The tool git-author that is a new built-in tool in git and implements the structural authorship and the weighted authorship model.
4. A study of five open source projects that characterizes the number of lines changed by multiple commits and multiple authors.
5. A line-level bug prediction model that performs consistently better than the file-level model.

2 Software Distributions

Under this funding, we contributed to the Dyninst and MRNet software distributions. These distributions are being used widely and continue to have a major impact in academia, research labs, and industry. We summarize the state of these distributions.

2.1 Dyninst

Dyninst is a suite of component libraries that provides comprehensive treatment of binary programs. DyninstAPI's capabilities are divided into three categories: 1) analysis, 2) modification/instrumentation, and 3) control.

Dynist provides both detailed control-flow analysis and data-flow analysis of binary code. It provides program abstractions in a platform-independent representation for such constructs as instructions, basic blocks, loops, and functions. For data flow, Dyninst supports both program slices and symbolic evaluation. The effectiveness of Dyninst binary analysis techniques has been maintained over the years, in spite of the fact that binary code from modern compilers grows significantly more complex over time. Aggressive optimizations being quite standard. It is common to find

- non-contiguous code layout, even within a single functions,
- functions that share code (e.g., from multiple entry points),
- functions without stack frames (i.e., no stack set-up or tear-down code), and
- functions with no return (e.g., from tail-call optimization).

Dyninst handles all these cases properly. Furthermore, Dyninst is opportunistic about the information available in an executable file. If symbol are available, Dyninst will use them. Dyninst, however, also operates sensibly on stripped binaries (e.g. no symbols). In addition, if debugging information is available, Dyninst will make use of it. If such information is not available, Dyninst tries to compensate by employing it's arsenal of code analysis techniques.

Dyninst's program modification facilities allow the user to edit a program's control flow graph, while maintain well-defined behaviors. One extremely important class of modifications is instrumentation. The instrumentation features of Dyninst allow inserting code at almost any instruction boundary. Instrumentation is described in terms of platform independent abstract syntax trees, build from DyninstAPI classes.

Dyninst can modify programs either statically or dynamically. Static modification is accomplished by rewriting a binary program or library. Dynamic modification is accomplished by modifying a program during execution. Dyninst's runtime code generator produces the machine code for the user's custom modification. Dyninst then dynamically inserts the newly-generated instrumentation code at runtime.

Dyninst's process control facilities are an important adjunct to Dyninst's other capabilities. Dyninst's process control facilities allow the Dyninst system to both control and monitor processes. Process control facilities usually take the form of process start, or process attach. In addition, Dyninst includes a facility for manipulating breakpoints, Dyninst's process monitoring facilities capture process events such as process and thread creation, process/thread termination, and exceptions.

All these features are captured in the upcoming Version 4.2 release of Dyninst, available from our web sites (mentioned in Section 5).

The following subsections contain details on each component tool kit in Dyninst.

2.1.1 DyninstAPI

This is the parent toolkit from which many of the other components were derived. It is still quite useful for building analysis, instrumentation, and control tools. Dyninst provides analysis, instrumentation, modification, and control of binary programs. The key strength of Dyninst is a collection of clean platform-independent abstractions to represent a binary program, both its static (code) characteristics and its dynamic (execution) characteristics.

It works on binaries that are statically or dynamically linked, executables and libraries, and with or without symbols. Dyninst provides almost identical analysis, instrumentation, and modification interfaces for statically analyzing, instrumenting, and modifying a binary *(binary rewriting)* and dynamically doing the same *(dynamic instrumentation)*. While it was originally a monolith that contained all the functionality as internal classes and methods, it is now a relatively thin layer built on top of the below toolkits.

2.1.2 ProcControlAPI

ProcControlAPI provides a portable interface to process start, control, and status monitoring. We note that this is quite an intricate component as it has to work consistently with several radically different operating system models for processes, threads, signals and exceptions, and address space structure). Doing so correctly and efficiently required detailed understanding of the process control and interfaces on all the supported platforms. From the user's point of view, the ProcControlAPI provides clean platform independent abstractions for the above models.

Recent additions to the ProcControlAPI include being able to work with large process groups in a single operations. This addition allows efficient support of process control and monitoring using the IBM BG/Q CDTI node debug interface.

2.1.3 SymtabAPI

SymtabAPI is a portable interface to both the processing and understanding of the symbol, header and debugging data of object files and libraries, and for the updating and generation of new binaries (to support binary rewriting). The rapidly evolving (almost frantically evolving) ELF and DWARF standards provides a challenge to maintain this interface (and increase the value of this toolkit). Note that the libelf and libdwarf libraries provide insufficient information to be a complete solution. For example, libdwarf provides a low-level interface to DWARF information but does not interpret it, so SymtabAPI provides a parser that sits on top of libdwarf and constructs function and variable information

2.1.4 ParseAPI

ParseAPI is a portable library to parse executable code in basic high level control abstractions, including instructions, basic blocks, functions, and loops. These abstractions are captured in the produced Control Flow Graphs (CFG's) and Call Graphs.

2.1.5 InstructionAPI

InstructionAPI is a portable interface to decoding instructions providing cross platform abstraction for the basic instruction operation, operands, and modes, and instruction semantics. It also provides a string representation of the instruction (for disassembly), and access to processor specific register and addressing modes.

2.1.6 StackwalkerAPI

Portable interface to walk run time stacks in a first- and third-party structure. First-party stack walks are when the library is in the same address space as the application programs, often being triggered by timers or breakpoints; third-party stack walks are when the library is in a separate tool (such as is the case for a debugger) and used the ProcControlAPI to access the application programs. Stackwalker includes the ability to understand stacks that include frames from signal or exception handlers, instrumentation tools, kernel calls, and optimized call frames. It supports a variety of analysis modes, including using code parsing results to define stack frame height.

2.1.7 DynC

DynC is a C-based language for defining code instrumentation snippets using the Dyninst toolkit objects. Using DynC can substantially simplify generating instrumentation code. DynC provides a clean abstraction of address spaces (adopted from the Cinquecento Programming Language) that allow variables used in a snippet be in the tool's address space, the applications address space (with the ability to name and use the program's functions and variables), or tool-local temporary space.

2.1.8 DataflowAPI

DataflowAPI is a portable interface to produce dataflow information for a binary program or library. This dataflow information includes forward and backward slices, symbolic evaluate of values in registers, liveness analysis for locations, and stack height analysis (i.e., how large is the current stack frame at any given program counter?).

2.2 MRNet

The desire to solve large-scale science problems in areas of national and global significance, including climate modeling, computational biology, and particle simulation, has driven the development of increasingly large parallel computing resources. Unfortunately, performance, debugging, and system administration tools that work well in small-scale environments often fail to scale as systems and applications get larger. In response to these deficiencies, we developed a tree-based overlay network infrastructure, MRNet, for building tools and applications that can scale to the largest of computing platforms, including current extreme-scale Cray and IBM BlueGene systems that contain millions of processor cores. MRNet makes operations such as command and control, and data collection and reduction, efficient at large scale.

Typically, tools are organized using a tree structure, where a single tool front-end interacts with a large set of tool back-ends (often called tool daemons). This structure is commonly referred to as a master-slave architecture. Tool back-ends are responsible for data collection and application control, when applicable. The tool front-end often provides the interface to users, and is responsible for analysis of data collected at the back-ends. For tools using this structure, the front-end quickly becomes a bottleneck due to centralized computation and communication with all back-ends. In addition, many application programs can use the same hierarchical structure to achieve extreme scale. MRNet provides a scalable solution for these tools and applications by interposing a tree-based overlay network (TBON) of processes between the tool front-end and back-ends.

The TBON is used to distribute tool activities normally performed by the front-end across the overlay processes, thus reducing analysis time and keeping the front-end load manageable. MRNet takes advantage of the logarithmic performance properties of trees to provide scalable multicast

communication and data aggregation. Tools and applications built using MRNet send and receive data between front-end and back-ends on logical data flows called streams. Data flowing on streams is encapsulated as packets, which are synchronized and aggregated using built-in or user-defined filters. MRNet's general-purpose abstractions allow tools to completely control how communication and computation is performed. Furthermore, MRNet lets tools and applications define the TBON topology and the placement of processes on distributed hosts. MRNet supports any tree topology, and provides a utility for easily generating common topology structures such as balanced and k-nomial trees

3 Publications

[1] Emily R. Jacobson, Andrew R. Bernat, William R. Williams, and Barton P. Miller, "Detecting Code Reuse Attacks with a Model of Conformant Program Execution", *International Symposium on Engineering Secure Software and Systems (ESSOS),* Munich, Germany, February 2014.

[2] Xiaozhu Meng, Barton P. Miller, William R. Williams, Andrew R. Bernat, "Mining Software Repositories for Accurate Authorship", *29th IEEE International Conference on Software Maintenance,* Eindhoven, Netherlands, September 2013

[3] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller, "Labeling Library Functions in Stripped Binaries", *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE),* Szeged, Hungary, September 2011.

[4] Emily R. Jacobson, Michael J. Brim, and Barton P. Miller, "A Lightweight Library for Building Scalable Tools", *Para 2010: State of the Art in Scientific and Parallel Computing,* Reykjavik, Iceland, June 2010.

4 Students Supported and Student Progress

Graduate Students Supported:

Emily Jacobson	Daniel McNulty
Salini Kowsalya	Xiaozhu Meng
Rohit Koul	

No post doctoral nor undergraduate students were supported during this period.

5 Project Web Sites

`http://www.paradyn.org`
`http://www.dyninst.org`

6 Outreach and Transitions

We continue to work with many key groups in government labs, research organizations, academia, and industry. Some of these interactions include:

- *Paraver, Barcelona Supercomputer Center, Prof. Jesus Labarta:* Paraver is a performance visualization and analysis tool based on traces. It can provide extremely detailed visualizations of performance behaviors on a wide variety of computational platforms.

Paraver uses MRNet for finding equivalence classes of traces among large numbers of processes, Paraver's Extrae binary program instrumenter is based on the DyninstAPI.

- *TAU, University of Oregon, Prof. Alan Malony:* TAU (Tuning and Analysis Utilities) gathers performance data through instrumentation of functions, methods, basic blocks, and statements. It also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumenter tool based on the Program Database Toolkit (PDT), or dynamically using DyninstAPI. TAU's profile visualization tool provides graphical displays of the performance analysis results, in aggregate and single node/context/thread forms.
- *Scalasca, Juelich Supercomputer Center, Dr. Bernd Mohr:* Scalasca is also a trace, analysis, and visualization tool for large-scale parallel systems. It combines runtime summaries to provide a performance overview with a description of detailed behavior described by event tracing. The traces are analyzed to identify wait states that occur such as for unevenly distributed workloads. Scalasca uses parallel trace-analysis to analyze results for large scale systems.

  Scalasca's COnfigurable Binary Instrumenter (COBI) is based on the DyninstAPI, Scalasca uses MRNet to help scalably process its trace data (such as for identifier unification).
- *STAT, Lawrence Livermore National Lab, Dr. Bronis de Supinski:* STAT (Stack Trace Analysis and debugging Tool) is the product of a LLNL and Wisconsin collaboration to produce a focused, easy to use debugging tool. It provide stack traces for programs running at extreme scale. Traces are collected and visualized in a fraction of a second, even on systems with a million or more application processes. It has been used in production to find difficult bugs at scale and has been used for finding system errors during the acceptance testing of the Sequoia IBM BG/Q system.

  STAT uses MRNet for its scalable collection, reduction, and visualization of traces, and StackwalkerAPI for collecting individual stack traces.
- *ATP, Cray Inc., Dr. Luiz Derose:* ATP (Abnormal Termination Processing) is Cray's post-mortem product for collecting information about programs that crash. It provides detailed state information about the crashed program, including stack traces.

  MRNet is used in ATP to scalably collect and reduce the traces. In addition, StackwalkerAPI is used to collect stack traces on the crashed processes. MRNet is used in ATP to scalably collect and reduce the traces. In addition, StackwalkerAPI is used to collect stack traces on the crashed processes. MRNet is distributed as a separate tool by Cray as a support partner product.
- *SystemTap, Red Hat Inc., Joshua Stone:* SystemTap is Red Hat's diagnostic monitoring tool for the kernel, services and application programs.

  Red Hat has adopted the DyninstAPI as their instrumentation mechanism for non-kernel monitoring in Red Hat Enterprise Linux (their flagship supported product). In addition, Red Hat is distributing Dyninst in its own right in RHEL.
- *VampirTrace, Technische Universitaet Dresden, Dr. Andreas Knuepfer:* VampirTrace is an open source library that allows detailed tracing of parallel applications that use message passing (MPI) and threads (OpenMP, Pthreads). VampirTrace is capable of tracing GPU accelerated applications and generates exact time stamps for all GPU related events.

  Vampir can instrument executables using the DyninstAPI with Vampir's `-vt:dyninst` option.

- *Open|Speedshop, Krell Labs, James Galorowicz:* Open|Speedshop is a project supported by the DOE NNSA Tri-Labs, to provide an open source, portable, and extensible performance monitoring and visualization tool for leadership class systems.

  Open|Speedshop uses Dyninst for both static and dynamic instrumentation of programs and MRNet for its scalability infrastructure.
- *MATE Autonomous University of Barcelona Prof. Ania Morajko:* MATE (Monitoring, Automatic and Tuning Environment) is an autotuning environment that monitors program behaviors and dynamically modifies the program or is runtime in response to those behaviors. MATE can adjust such characteristics as the number of threads, socket protocols,

  MATE uses Dyninst for its instrumentation and MRNet for control of its daemon processes and scalable collection of performance data.

In addition to the above list of projects, Dyninst has become a frequently used tool kit for cyber security research projects. It is being used for code analysis in cyber forensics and for modifying code to make it more difficult to attach (so called *hardening* of a binary program).