Final Technical Report: Tools for the Development of High-Performance Energy Applications and Systems

February 2010 to January 2014

SC0004061/FG02-10ER25972 (PRJ36WV)

Barton P. Miller

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

1 Technical Accomplishments

The Paradyn project has a history of developing algorithms, techniques, and software that push the cutting edge of tool technology for high-end computing systems. Under this funding, we are working on a three-year agenda to make substantial new advances in support of new and emerging Petascale systems. The overall goal for this work is to address the steady increase in complexity of these petascale systems.

Our work covers two key areas: (1) The analysis, instrumentation and control of binary programs. Work in this area falls under the general framework of the Dyninst API tool kits. (2) Infrastructure for building tools and applications at extreme scale. Work in this area falls under the general framework of the MRNet scalability framework.

Note that work done under this funding is closely related to work done under a contemporaneous grant, "Foundational Tools for Petascale Computing", SC0003922/FG02-10ER25940, UW PRJ27NU.

1.1 Process Control Component Library

Dyninst has the ability to monitor and control another process. This functionality is critical to a wide variety of tools including those for performance profiling, debugging, system monitoring, testing, and security sandboxing. Having a stable, multi-platform library can significantly simplify the development of such tools.

This year, we focused on implementation of a new tool component, the *ProcControlAPI*. A primary goal of this effort was to build a high-level tool component for controlling and monitoring processes during execution. Based on prior experience with OS debugger interfaces, we know that these interfaces can be complicated, subtle and error prone. The ProcControlAPI is designed to abstract and hide these complexities and present a simple interface to users.

A secondary goal of ProcControlAPI was to improve DyninstAPI portability, stability, and scalability by replacing the existing process control infrastructure. Our initial development of the ProcControlAPI was on the Linux platform for both 32- and 64-bit systems. Following this port, we began port of ProcControlAPI port to the BlueGene family which (along with other efforts) will eventually allow DyninstAPI to support BlueGene systems.

The port to Linux and FreeBSD is completed, and the BlueGene port is in final testing. Concurrent with this effort, we are about halfway through a Windows/x86 port. Windows provides a relatively clean and simple process control interface, but has a radically different thread and signal model, so generalizing the API to Windows has some distinct challenges.

In addition, Dyninst has been restructured so that it is based on ProcControlAPI. In the course of designed the ProcControlAPI, we redesigned the internal event and threading structure to handle the myriad of debug interfaces present in different operating systems. As a result, Dyninst now has a significantly simpler and cleaner internal design (as will any tool that uses the ProcControlAPI).

1.2 Support for New Code Patching Interfaces

We have been designing a new component, called the PatchAPI, that encapsulates the part of Dyninst that coordinates the code patching and code generation for our binary instrumentation. This component is now completed and in general use. The new PatchAPI has plug-in interface that handles the full complexity of the Dyninst code patching and the simplified requirements of Self-Propelled Instrumentation. In addition, for Dyninst, it supports third-party functionality (working from a control process operating on an application process) and for Self-Propelled, it supports first-party functionality (working inside the same address space as the application).

This new API has already found wider use at LLNL in experiments there on optimizing floating point operations in executable programs. The PatchAPI allows new sources of code to be patched into a binary using the current power of the Dyninst infrastructure. In this case, we are able to support raw binary code fragments to be inserted into to a program with the same level of ease and protection as we have used for our current AST (abstract syntax tree) interface.

1.3 More Efficient and Safer Instrumentation

Binary instrumentation modifies a binary program, either pre-execution or during execution. This technique can be used to insert monitoring code, such as for performance analysis, attack detection, cyberforensics, or behavior monitoring; to modify program data, such as for dynamic transactional memory; or to perform code replacement, such as for performance steering. For the past several years, we have been developing the Dyninst binary analysis and instrumentation system. By combining these capabilities we have made binary instrumentation more efficient and precise, and enabled a class of tools that combine analysis with instrumentation. Dyninst distinguishes itself from other instrumenters through its abstract interface; its emphasis on anywhere, any-time (AWAT) instrumentation; and its low overhead that is proportional to the number of instrumented locations. Our recent work allows Dyninst to provide AWAT instrumentation with proportional cost.

Dyninst allows users to instrument anywhere in the binary. Other binary instrumentation approaches allow users to insert instrumentation at instructions or specific types of control ow edges. Surprisingly, instrumenting instructions or edges is not sufficient to capture program behavior based on structural characteristics such as functions or loops. Consider the case of instrumenting the entry of a function; such instrumentation should execute only once per function call. Other binary instrumenters do not directly support instrumenting function entries; instead, users instead instrument the first instruction in the function (assuming sufficient information, such as a symbol table, is available to correctly identify that instruction). However, the first instruction of functions with no preamble code may be executed multiple times per function invocation. Therefore, such instrumentation would execute multiple times per invocation. This problem also holds for instrumenting other control flow features of a program, such as function exits and loop entries, iterations, and exits. We address this problem by allowing users to specify instrumentation locations in terms of the control ow graph (CFG) in addition to at the instruction level. We define classes of instpoints to describe interesting locations in the CFG (e.g., function entries and exits), and users insert instrumentation by annotating these instpoints with instrumentation code.

Dyninst allows users to insert, remove, or modify instrumentation at any time in the execution continuum: pre-execution (binary rewriting), before code has executed for the first time, or while the instrumented code is currently executing (dynamic instrumentation). Binary rewriting offers several benefits over dynamic instrumentation, such as amortizing the cost of instrumenting a binary over multiple executions or eliminating the need for having the instrumenter present during execution. In contrast, dynamic instrumentation allows users to instrument as the program executes. Other binary instrumentation approaches support either static instrumentation or dynamic instrumentation, but not both. Furthermore, other dynamic instrumenters cannot guarantee that instrumenting currently executing code will take immediate effect.

Our approach allows users to use the same tool to instrument both statically and dynamically. Furthermore, during dynamic instrumentation any changes are guaranteed to take immediate effect. During binary rewriting we present the user with a CFG that was derived using static analysis; this CFG supports both analysis and instrumentation. If we are performing dynamic instrumentation we also report run-time events, such as changes to the CFG or system events (e.g., thread creation or process exit). For example, if we are instrumenting a binary that is statically obfuscated or generates code at runtime, we will discover the new code before it executes and report that new code to the user. Dyninst imposes proportional cost by imposing overhead only when instrumented code is executed; furthermore, removing instrumentation also removes its cost. This can significantly reduce overhead if the number of instrumented locations is small (e.g., a subset of functions in the program) or if instrumentation can be removed as the program runs. Binary instrumenters rely on a technique we call relocation to insert both instrumentation and any additional infrastructure code the instrumenter requires (e.g., dynamic analysis code). Relocation moves code to a new location where it can be expanded to include new code (instrumentation and analysis code) and transforms it to preserve its original behavior; as a result of this transformation the relocated code frequently executes slower than the original. Other approaches use dynamic analysis to discover all executed code, and thus must relocate all executed code whether or not it is instrumented. As a result, they impose overhead even on uninstrumented code.

By using static analysis to derive the CFG rather than dynamic analysis we can greatly reduce the amount of uninstrumented code that must be relocated. In conventional binaries, we only relocate instrumented code. This relocation may be performed on a basic block or function basis, depending on the density of instrumentation. As a result, any uninstrumented code executes natively with no overhead; this is also true for code whose instrumentation is removed. For binaries where the CFG is incomplete (e.g., JIT compilers or self-unpacking malware) we rely on relocation to enable dynamic analysis, but unlike other approaches we only relocate the specific locations (e.g., indirect jumps or calls) that we know are incompletely analyzed.

As a result of our analysis and instrumentation techniques, Dyninst provides greater precision of instrumentation without incurring additional cost. For example, in experiments performed on the SPEC benchmark suite we incurred an average 71% execution overhead when instrumenting every basic block, as opposed to the 128% imposed by PIN. This is a worst-case example of our techniques, since it does not leverage the proportional cost of our approach, and we show that the cost we impose decreases as fewer locations are instrumented while other instrumenters impose cost even when executing uninstrumented code. We can also instrument malware that executes incorrectly or crashes when instrumented with other instrumenters, although the overhead we impose is higher than our overhead on conventional binaries due to the tamper-resistance features used by these binaries.

1.4 Binary Code Analysis in Support of Finding Security Problems

Software security vulnerabilities are an ever-present threat, and one way of defending against attackers who would exploit these vulnerabilities is to detect exploits before an attacker can. Analysts performing this task often work with binary code, where little human-readable information about a program is available. Some automated software testing techniques broadly search a binary program for common vulnerabilities, such as buffer overflows, and ways to exploit them. We are leveraging our work in binary code analysis and instrumentation to target this search to efficiently generate exploits for specific vulnerabilities that may be more sophisticated in nature. Our current focus is on finding exploit input that will drive a program to a known vulnerability. We are extending the technique of *concolic execution,* where expressions representing the effect of input on control flow are accumulated as a program executes, and then modified and solved for different input that explores a different control flow path. While most existing work tries to explore as many of a program's control flow paths as possible, we only need to find a single path that leads to a known vulnerability as fast as possible.

Our work so far has been on implementing a basic concolic execution system. Currently, our system handles a small number of simple test programs. Given one of these x86 binary programs and the address of an instruction within the program, representing a vulnerability, our system will automatically generate an input string for the program that causes it to execute the instruction. Since we are using a simple depth-first search over a program's control flow space, we currently manually constrain the size of an input string to make this search feasible. Our next major focus will be on more advanced path search strategies that will make this search truly automatic and allow us to scale to larger programs. We are also working on handling network servers, which are common targets for attackers.

1.5 A New Standard for Deploying Tools in Leadership Class Systems

This work was done in collaboration with the University of New Mexico and Lawrence Livermore National Lab.

In high-performance computing (HPC) environments, system sizes continue

to grow dramatically. On a recent Top 500 list, 286 (or 57.2%) of the entries have over 8,192 cores, compared to 18 (or 3.6%) just 5 years ago. On the most recent list, four have over 200K cores, six others have over 128K cores, and four more have over 64K cores. Lawrence Livermore National Laboratory (LLNL) has its 1.6 million core system, Sequoia. Further, exascale systems are projected to have on the order of tens to hundreds of millions of cores within the current decade.

Capability class and other very large application instances that utilize large portions of the entire system, as well as HPC system software and tools, must scale to these massive sizes.

All distributed software systems require a bootstrapping phase in which their processes are started and some basic information is exchanged. Given an allocation of physical computational nodes, we define distributed software infrastructure bootstrapping as the procedure of instantiating the infrastructure's composite processes on the computational nodes and exchanging the information that these processes require to complete their setup and to enter their primary operational phases[1].

An efficient bootstrapping process can be critical, and inefficiencies in this process can become an impediment for software deployment and utility. For example, a few seconds are generally the upper bound on acceptable delay for interactive operations. At current scales, it can take several minutes to deploy an interactive software tool, even when the tool can perform its key functions much more quickly. Our experiences with our own Stack Trace Analysis Tool (STAT) demonstrated this problem: a full-scale instance of STAT on the Lawrence Livermore National Laboratory's BlueGene system could take minutes to start-up and subsequently, less than a single second to perform its analysis. Efficient bootstrapping can also be critical in the many-task computational model. In this model, applications are decomposed into many (thousands and sometimes millions of) tasks, and processes are launched continuously on the available computational resources throughout the application's execution. The finer-grained the tasks, the greater the impact of inefficient bootstrapping.

In this effort, we designed and developed a set of abstractions and mechanisms that address the challenges of scalable software system bootstrapping and deficiencies in current bootstrapping approaches. We presented the lightweight infrastructure-bootstrapping infrastructure (LIBI), a reference implementation of our system for launching distributed applications. LIBI is not intended to replace existing resource managers (RMs); LIBI is designed to provide a more intuitive and flexible system bootstrapping interface and mechanisms for portably leveraging RMs. LIBI also provides efficient and scalable rsh-based process launch for situations in which RMs are unavailable or cannot be used for one reason or another.

LIBI makes the following contributions:

1. We developed a system for classifying process launching and bootstrapping mechanisms. This classification gives us a way to compare different bootstrapping facilities, particularly with respect to performance and scalability;
2. We defined the architecture, design and implementation of our LIBI software prototype; and
3. We constructed an evaluation of our LIBI prototype, which demonstrated how LIBI can improve large scale software system bootstrapping.

1.6 Improving the Determination of Software Authorship

Information as to who wrote a given piece of code, authorship, is used to analyze software quality, improve software maintenance, and perform software forensics. Current tools approximate line level authorship by assuming that the last person to change a line is its author, while ignoring all earlier changes. In this effort, we showed how to mine a code repository for the development his-

1. Technically, bootstrapping is not complete until the processes act upon exchanged information; this final activity is infrastructure-dependent.

tory 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 method abstracts 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 then use the history to attribute each character of the line to the responsible author and assign contribution weights. We have implemented this new functionality as an extension to git.

The methods used by current tools (git-blame, svnannotate and CVS-annotate) for obtaining line level authorship loses information. A line of code may be changed multiple times by different 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 effort, we defined the *repository 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 used 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 showed that the line-level model performs consistently better than the file-level model when evaluated on effort-aware metrics.

This work 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.

1.7 Tool User Support

A critical part of our project is to distribute tools that results from our research, design, and implementation efforts. Projects all over the world depend on the software that we produce. As part of this effort to distribute software, we carry out thorough software testing and provide support to our users. The testing and user support make easier for other groups to adopt our software, and our staff researchers play an invaluable role in this process.

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] Benjamin Welton and Barton Miller, "The Anatomy of Mr. Scan: A Dissection of Performance of an Extreme Scale GPU-Based Clustering Algorithm", *Computer Sciences Technical Report, submitted for publication,* January 2014.

[2] Barton P. Miller, Dorian C. Arnold, Michael J. Brim, Philip C. Roth, Evan Samanas, Benjamin Welton and Bill Williams, "Building on Lessons Learned From Over a Decade of MRNet Research and Development", *Extreme Scale Programming Tools Workshop,* Denver, November 2013.

[3] Benjamin Welton, Evan Samanas, and Barton P. Miller, "Mr. Scan: Extreme Scale Density-Based Clustering Using a Tree-Based Network of GPGPU Nodes", S*upercomputing 2013 (SC2013),* Denver, November 2013.

[4] 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

[5] Dong H. Ahn, Michael J. Brim, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Matthew P. LeGendre, Barton P. Miller, Adam Moody, Martin Schulz, "Efficient and Scalable Retrieval Techniques for Global File Properties", *27th IEEE International Parallel & Distributed Processing Symposium,* Boston, MA, May 2013.

[6] Joshua D. Goehner, Dorian C. Arnold, Dong H. Ahn, Gregory L. Lee, Bronis R. de Supinski, Matthew P. Legendre, Barton P. Miller, Martin Schulz, "LIBI: A Framework for Bootstrapping Extreme Scale Software Systems", *Journal of Parallel Computing* **29**, 3, March 2013, pp. 167-176.

[7] Wenbin Fang, James A. Kupsch, and Barton P. Miller, "Automated Tracing and Visualization of Software Security Structure and Properties", *9th International Symposium on Visualization for Cyber Security (VizSec),* Seattle, October 2012.

[8] Kevin A. Roundy and Barton P. Miller, "Hybrid Analysis and Control of Malware Binaries", *Recent Advances in Intrusion Detection (RAID),* Ottawa, Canada, September 2010.

[9] 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.

[10] N.E. Rosenblum, B.P. Miller and X. Zhu, "Extracting Compiler Provenance from Program Binaries", *9th ACM SIGNPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE),* Toronto, June 2010.

[11] Michael J. Brim, Luiz DeRose, Barton P. Miller, Ramya Olichandran, and Philip C. Roth, "MRNet: A Scalable Infrastructure for Development of Parallel Tools and Applications", *Cray Users Group Conference (CUG),* Edinburgh, Scotland, May 2010.

[12] Dorian C. Arnold and Barton P. Miller, "Scalable Failure Recovery for High-performance Data Aggregation", *International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, April 2010.

4 Students Supported and Student Progress

Graduate Students Supported:

Wenbin Fang	**Xiaozhu Meng**
Todd Frederick	**Evan Samanas**
Salini Kowsalya	**Benjamin Welton**

No post doctoral or undergraduate students were supported during this period.

5 Project Web Site

```
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 moni-

toring 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).