Final Technical Report: Foundational Tools for Petascale Computing

December 2009 to December 2013

SC0003922/FG02-10ER25940 (PRJ27NU)

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, "High-Performance Energy Applications and Systems", SC0004061/FG02-10ER25972, UW PRJ36WV.

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 Extreme Scale Harnessing of Hybrid GPU/CPU Computation

We investigated techniques for density-based clustering of multi-billion point datasets such as geospatial data. Specifically we developed a clustering technique that uses a hybrid computing model combining large-scale multicast/reduction overlay networks operating with nodes equipped with high-end GPGPUs. This hybrid computation allows for clustering of extremely large datasets in an efficient manner. We introduced a new clustering algorithm, Mr. Scan, and an end-to-end implementation of this algorithm that we showed can efficiently scale to billions of points on a leadership class supercomputer.

Clustering is the act of classifying data points, where data points that are considered similar are contained in the same cluster and dissimilar points are in different clusters Clustering helps researchers and data analysts gain insight into their data, e.g., identifying and tracking objects such as gamma-ray bursts in sky observation data, monitoring the growth and decline of forests in the United States and identifying performance bottlenecks in large-scale parallel applications [12]. We

focus on a type of clustering algorithm called density-based clustering, which classifies points into clusters based on the density of the region surrounding the point. Density-based clustering detects the number of clusters in a dataset without prior knowledge and is able to find clusters with non-convex shapes.

Datasets such as the Sloan Digital Sky Survey and geolocated tweets from Twitter are useful to cluster but are too large (i.e., billions of data points) to be practically computed on a small or medium-sized parallel computer (100's to 1000's of nodes) by any non-trivial clustering algorithm. These large data sizes require the largest-scale parallel systems that are in use today. However, there are few distributed density-based clustering algorithms designed to run on these large-scale systems. Existing distributed density based algorithms typically reduce the quality of the output when compared to the single-node version, or they do not scale to the sizes needed for these datasets.

Mr. Scan is our implementation of the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering algorithm. DBSCAN is the most widely cited density-based clustering algorithm and has been shown to be well-suited for data analysis in many fields, e.g., the analysis of laser ablated material and tracking population movement by use of geotagged photographs. The benefits DBSCAN has over other clustering algorithms are that it has the ability to nd irregularly shaped clusters, it distinguishes data points that are considered noise (i.e., points in low density regions) from clusters, and it is able to cluster data where the number of clusters in the dataset is not known in advance. These features come at a cost, since the computational complexity of DBSCAN is *O(n2)* where *n* is the number of points in the input dataset. This complexity is a result of the calculation of a *n x n* matrix containing the distances between all points. This matrix can be replaced with a spatial tree index which reduces the cost of distance calculations leading to an average case complexity of *O(n log n)*.

Mr. Scan is the first implementation of DBSCAN that can scale up to 6.5 billion data points and the first distributed DBSCAN algorithm that incorporates the use of GPGPUs. It uses a programming paradigm that organizes processes into a multi-level tree with an arbitrary topology. In this multi-level tree paradigm, DBSCAN calculations are done on the GPGPU leaf nodes and these results are combined on non-leaf nodes. Mr. Scan is also the first clustering algorithm to use this programming paradigm to our knowledge. **Using this multi-level tree design we demonstrate the capability to cluster 6.5 billion points using 8,192 GPGPU nodes in 17.3 minutes.**

The ability to cluster billions of data points with DBSCAN can only be realized if the key obstacles to scaling DBSCAN are overcome: load balancing, cluster merging, and distributing data advantageously. The running time of DBSCAN increases as a function of spatial density of the input data points, which causes a load imbalance when compute nodes contain regions of varying density. We modify DBSCAN to find the most dense regions and infer their membership in a cluster without evaluating the points inside these dense regions. Results from DBSCAN compute nodes must be merged accurately without requiring the entirety of each cluster. We resolve this by requiring a small, bounded number of representative points per cluster to perform a merge. Finally, data must be distributed in a manner that balances DBSCAN's clustering operation and the overhead of merging clusters. We achieve this with a heuristic that spatially decomposes the data into partitions to balance the merge overhead. Each partition contains roughly equal point counts to aid in balancing DBSCAN clustering time.

1.2 Scalably Determining File Properties

This work was done in collaboration with Lawrence Livermore National Labs.

Large-scale system sizes continue to grow exponentially. Systems with ten thousand or more compute cores are common and LLNL's recently delivered Sequoia system has over a million cores. This exponential growth in concurrency makes contention within the storage hierarchy common and efficient file access a challenge.

Avoiding contention requires an understanding of the performance and scalability of the entire storage hierarchy. Any software running on large-scale systems, including scientific applications, parallel libraries and tools, must determine dynamically how to adjust their strategies to improve performance. However, determining the properties of the storage hierarchy and its properties for all mounted file systems is nontrivial due to the increasing complexity of file system hierarchies. Further, existing parallel I/O software focuses on the I/O patterns for large data set accesses, and does not suit other I/O access patterns, such as uncoordinated, simultaneous accesses to small files — e.g., launching an executable that depends on many shared libraries triggers vast numbers of simultaneous accesses to the same library files when each process in the application loads the library dependencies. Further, in parallel environments, a file can reside in one or more local or remote file systems. Thus, different physical file systems may serve files with an identical file path to different processes of the same program.

To cope with these complexities, high performance computing (HPC) software requires a richer set of abstractions and scalable mechanisms by which to retrieve the performance properties of a file. To close this gap and enable efficient run time access to such information, we propose Fast Global File Status (FGFS), a scalable mechanism to retrieve file information including the degree of replication or distribution and consistency across local or remote file systems. FGFS builds on a simple node-local technique that raises the local namespace of a file to a global namespace using a memory-resident mount points table. FGFS extracts the global properties of a file path by comparing and grouping the global names seen by various processes.

FGFS status queries retrieve global information on both individual files and entire file systems. FGFS supports synchronous and asynchronous file status queries; File systems status queries serve as an inverse classifier that selects those mounted file systems that best match a given set of global properties required by an I/O operation. We design the FGFS Application Programming Interface (API) and its implementation to support the file access and information needs of a wide range of HPC programs, libraries and tools.

In this effort, we made the following contributions:

- A novel node-local technique to raise locally-defined file names to a global namespace;
- Scalable parallel algorithms based on string comparisons to compute global file properties;
- APIs and their implementations to provide global file information to existing HPC software at run time.

Our performance evaluation on a large multi-physics production application showed that most FGFS file status queries on its executable and its 848 shared libraries completed in 272 milliseconds or less at 32,768 MPI processes. Even the most expensive query that checks the global consistency of these files, takes under 7 seconds at this scale. Compared to the traditional technique in which remote daemons compute and compare checksums, FGFS provides several orders of magnitude improvements.

Additionally, we applied our techniques to three case studies and showed how FGFS enables a wide range of HPC software to improve the scalability of its file I/O patterns. The first case study applied FGFS to the Stack Trace Analysis Tool (STAT) and showed that FGFS aids this lightweight debugging tool in choosing between direct file I/O and file broadcasting. This capability resulted in a 52x speedup at 16,384 MPI processes. Second, we demonstrated that an efficient FGFS file status query is a crucial element for a highly scalable dynamic loading technique called Scalable Parallel Input Network for Dynamic Loading Environment (SPINDLE). The final study showed that FGFS file system status queries helped the Scalable Checkpoint/Restart (SCR) library to eliminate the need for arduous manual configuration efforts in discovering the best file system on which to store its multilevel checkpoints.

1.3 Scalability Infrastructure for Tools

Tools and middleware face a daunting challenge to operate effectively on the world's largest distributed systems containing tens of thousands of hosts and hundreds of thousands of processors. A large class of problems encountered at this scale result from system designs that force group operations to use serial interactions with operating systems and file systems. As the target group size grows, the resulting group operation latency grows linearly or worse.

In previous work, we introduced group file operations, a solution to the problem of applying the same file operations to a large group of files located across thousands of independent hosts. The keys to the group file operation idiom are explicit identification of file groups using directories as the grouping mechanism, and the ability to name a file group as the target for conventional file system operations such as read and write. Group file operations provide an interface that eliminates forced iteration, thus enabling scalable implementations. To support scalable group file operations, we developed the TBON-FS distributed file system, which employs a tree-based overlay network (TBON) to provide scalable communication of group file operation requests and distributed aggregation of response data. TBON-FS provides client tools with a single-system image (SSI) name space containing files from thousands of independent file servers. Single-system image name spaces enable applications to access and operate on distributed resources as if they were local, easing the development effort by allowing developers to focus on features rather than distributed access and communication.

Several classes of tools and middleware can benefit from group file operations, including systems for distributed system administration and monitoring, parallel application runtimes, and distributed debuggers. For instance, tools for distributed monitoring and debugging often need to access the synthetic files for process control or inspection as provided by `/proc` across a large set of independent hosts. Using group file operations, these tools can easily control or monitor groups of processes by defining file groups over the target files and performing group read or write operations.

Although our initial investigation clearly showed the scalability benefits of group read and write operations, it also revealed a significant piece was missing, the ability to define file groups in a scalable fashion. TBON-FS originally used a simple, static composition strategy for constructing its SSI name space -- each file server's name space was placed in an independent directory hierarchy of the global name space. This inflexible structure results in inefficient group definition for groups that contain files from many servers. For each new group, the TBON-FS client must create a directory and populate it with symbolic links to each member file in a non-scalable, iterative

manner that can take thousands of seconds for groups containing tens of thousands of distributed files. To avoid this centralized, iterative group definition, we began investigating scalable approaches for distributed construction of the name space that could be implemented using the TBON.

After considering a few straightforward techniques for addressing the problem of scalable group definition, including parallel path matching using regular expressions, it quickly became clear that no single approach to constructing the TBON-FS name space would meet the group definition requirements for a wide variety of tools and middleware. For instance, consider a strategy for creating groups from the synthetic files provided by `/proc` across a large set of independent hosts. A parallel debugger or job management system may wish to create a file group representing all the processes of a specific parallel application, while a distributed system load monitoring program may want groups consisting of all processes from every host or all processes running the same executable. We believe each TBON-FS client is best-suited to the task of constructing and organizing the global name space, and our goal is to develop a method for specifying global name space composition that is both flexible and scalable. Clients should be able to easily identify the files or directory hierarchies from each server's name space to include in the global name space, and to control how files from independent servers are correlated to achieve a name space tailored for use with group file operations. A key to achieving the latter property is an efficient and automated method for creating file groups as directories within the composite name space.

To address prior deficiencies and our name space composition goals for TBON-FS, we developed a language for describing compositions with three key qualities:

- *Scalability* - many name spaces can be combined using efficient distributed name space construction, avoiding centralized pair-wise operations.
- *Simplicity* - name space composition is easily described using a simple tree abstraction for name spaces and a set of tree composition operators with clear semantics.
- *Flexibility* - many interesting compositions can be specified by combining declarative tree operations with prescriptive programming constructs.

The language provides a semantic foundation that guides our approach for efficient large-scale name space composition within TBON-FS, and can be adopted by previous or future systems requiring flexible name space composition.

Our language is FINAL, for File Name space Aggregation Language. FINAL treats name space composition abstractly as operations on rooted trees of names, and provides five tree composition operations: subtree, prune, extend, graft, and merge. Specifications containing FINAL declarations are translated at runtime to produce a name space accessible via a library interface. We demonstrate FINAL's expressive power by using it to describe many interestingly diverse compositions.

1.4 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.

1.5 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 charac-

teristics 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

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] 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.

[3] 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.

[4] 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.

[5] 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

[6] 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.

[7] 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.

[8] Kevin A. Roundy and Barton P. Miller, "Binary-Code Obfuscations in Prevalent Packer Tools", *ACM Computing Surveys* **46**, 1, October 2013.

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

[10] Andrew R. Bernat, Kevin Roundy, and Barton. P Miller, "Efficient, Sensitivity Resistant Binary Instrumentation", *International Symposium on Software Testing and Analysis (ISSTA),* Toronto, Canada, July 2011.

[11] Michael J. Brim, Barton. P Miller, and Vic Zandy, "FINAL: Flexible and Scalable Composition of File System Name Spaces", *International Workshop on Runtime and Operating Systems for Supercomputers 2011 (ROSS'11),* Tucson, Arizona, May 2011.

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

[13] 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.

[14] 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.

[15] 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.

[16] Paradyn Project. **The Multicast/Reduction Network: A User's Guide**. University of Wisconsin, `http://www.paradyn.org/mrnet/release-2.1/UsersGuide.html`

4 Students Supported and Student Progress

Graduate Students Supported:

Emily Jacobson	**Kevin Roundy**
Daniel McNulty	**Benjamin Welton**
Xiaozhu Meng	

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).