Stack Trace Analysis for Large Scale Debugging [*]

Dorian C. Arnold[†], Dong H. Ahn[‡], Bronis R. de Supinski[‡], Gregory Lee[‡], Barton P. Miller[†], and Martin Schulz[‡]

[†] Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{darnold, bart}@cs.wisc.edu

[‡] Lawrence Livermore National Laboratory
Livermore, CA 94550
{ahn1, bronis, lee218, schulzm}@llnl.gov

Abstract

There are few runtime tools for modestly sized computing systems, with 10^3 processors, and above this scale, they work poorly. We present the Stack Trace Analysis Tool (STAT) to aid in debugging extreme-scale applications. STAT can reduce the problem exploration space from thousands of processes to a few by sampling application stack traces to form *process equivalence classes*, groups of processes exhibiting similar behavior. In typical parallel computations, large numbers of processes exhibit a small number of different behavior classes, manifested as common patterns in their stack traces. The problem space is reduced to representatives from these common behavior classes upon which we can use full-featured debuggers for root cause analysis.

STAT scalably collects stack traces over a sampling period to assemble a profile of the application's behavior. STAT routines process the trace samples to form a call graph prefix tree that depicts the program's behavior over the program's process space and over time. The prefix tree encodes common behaviors among the various stack samples, distinguishing classes of behavior from which representatives can be targeted for deeper analysis. STAT leverages MRNet, an infrastructure for tool control and data analyses, to overcome scalability barriers encountered by heavy-weight debuggers.

We present STAT's design and an evaluation that shows STAT gathers informative process traces from thousands of processes with sub-second latencies, a significant improvement over existing tools. Our case studies of production codes verify that STAT supports the quick identification of errors that were previously difficult to locate.

1 Introduction

In 2005, Lawrence Livermore National Laboratory's (LLNL's) BlueGene/L (BG/L) set the current benchmark for extremely large scale systems with 131,072 processors, while other supercomputer class systems like the Cray XT3 promise to operate with more than 60,000

[*]Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-ABS-225137). This work is also supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

processing cores. On the most recent Top 500 list [1], 228 systems (45.6%) had more than 1,024 processors and 18 systems had more than 8,192. With government initiatives for petaflop scale systems in the United States, Europe, Japan, and China, high-performance computing (HPC) systems with 10^4 and 10^5 processors will become commonplace, and 10^6 processor systems soon will be introduced. Yet, tools for debugging and analyzing the programs, even at existing scales, are non-existent. For example, on BG/L, TotalView [12], arguably the most advanced parallel debugger, takes more than one and two minutes to collect and compose stack traces from 2048 and 4096 processes (4,096 is just 3% of BG/L).

Our work targets the identification and diagnosis of application behavior. The set of questions we aim to help users answer include: what is the application currently doing? Is the application making useful progress or stuck in a deadlock or infinite loop? As noted by Roth, Arnold and Miller [24], developing a scalable diagnosis tool presents several challenges:

- *Overwhelming channels of control*: In most existing parallel debuggers, a single front-end process controls the interactions between back-end tool daemon processes and the processes of the application being debugged. The front-end spends unacceptably long times managing the connections to the back-end daemons at large process counts.
- *Large data volumes*: As the number of debugged processes increases, the volume of data becomes prohibitively expensive to gather.
- *Excessive data analysis overhead*: Even if the debug data can be gathered in acceptable time, the time to process and to present it becomes excessive, often causing users to resort to targeted print statements.
- *Scalable presentation of results*: Finally, presenting a standard source code trace for individual processes overwhelms the user and prevents quick anomaly detection; alternative presentation paradigms that synthesize across the set of processes are essential.

To address these challenges, we present the Stack Trace Analysis Tool (STAT), which manages the scalable collection, analysis and visualization of stack trace profiles used to depict application behavior. Specifically, we sample application stack traces to form *process equivalence classes*, groups of processes exhibiting similar behavior. Our experience shows that in typical parallel computations, very large numbers of processes will exhibit a small number of different behaviors even when exhibiting coding errors. Specifically, coding errors frequently manifest themselves as unexpected behavior in a small number of subsets, or even a single subset, of processes with similar runtime behavior. These errors can be detected through spatial differentiation of the process space; however, the unexpected behavior typically has a temporal aspect as well – the behavior is erroneous not because it occurs but because it persists. Our mechanisms detect these equivalence classes quickly, allowing the user to focus on a single representative of each.

Once we identify the equivalence classes, we present the user a call graph prefix tree that distinguishes them visually. We achieve scalability by leveraging the tree-based overlay network (TB$\bar{\text{O}}$N) model, which exploits the logarithmic scaling properties of trees for tool control and data analyses; specifically, STAT leverages the MRNet TB$\bar{\text{O}}$N implementation [24]. Our premise is that a scalable, lightweight diagnosis approach can effectively reduce the exploration space from thousands or even millions of processes to a handful of behavior classes (and class representatives). Once the problem space is effectively reduced, we can perform root cause analysis with a full-featured debugger, since now it is only necessary that this debugger attach to a small subset of the processes.

This rest of this paper details our three main contributions:

1. A simple but highly effective stack trace analysis technique that reduces debug exploration spaces from thousands to a handful of processes with sub-second latencies;
2. Process group visualizations that can effectively guide users to problem diagnoses by displaying both spatial and temporal relationships; and
3. A scalable tool prototype that implements our analysis and visualization techniques.

In Section 1, we present case studies of two production codes that motivate our stack trace analysis approach. In Section 2, we present the details of our techniques followed by the STAT's design, implementation, and performance in Section 3. In this section, we also apply STAT tool to the case studies and discuss future research. Related tools and research are discussed in Section 4. Finally, we summarize the expected impact of this work in Section 5. sectionMotivating Case Studies

To motivate the need for our technology, we consider two large-scale debugging case studies. These studies involve problems with elusive root causes in CCSM and ViSUS, two parallel applications. In both cases, the defects manifested themselves as hangs at relatively large scales on LLNL machines. These case studies capture a current manual approach for large-scale debugging, highlight its effectiveness, and present its scalability limitations. They demonstrate the need for a better tool that automatically applies similar techniques much more quickly and scalably.

1.1 Pthread Deadlock Exposed by CCSM

The Community Climate System Model (CCSM) [7] is a widely used parallel application that comprises multiple climatic simulators representing the atmosphere, ocean, sea ice and land surface connected by a coupling module. For several years, scientists have used CCSM and its components to improve climate predictions. Recently, an LLNL climate scientist reported intermittent hangs when running CCSM with 472 MPI tasks on the Thunder machine [17], an Itanium2 cluster with Quadrics Elan4 interconnect [23]. Several factors made the diagnostic process extremely challenging. For instance, CCSM consists of multiple programs and multiple data (MPMD). More importantly, the malfunction only occurred at large scale and non-deterministically manifested itself at apparently random program locations if at all.

During ten days of repeated attempts to reproduce the problem in a more controlled environment, the hang only occurred twice. Analysis of the debug data from the first reproduction of the hang suggested a pthread deadlock in an MPI task. Additional data from the second reproduction of the error confirmed this.

Figure 1 depicts the manual data gathering and postmortem analysis that we applied during the first reproduction of the error. We attached to all 472 MPI tasks using TotalView to gather several call graphs that capture the dynamic behavior of all tasks, an enhancement prototyped at LLNL. TotalView's call graph display provides a view into a program's spatial behavior (i.e., across job processes). The leftmost graph in Figure 1 shows the top of the agglomerated call graph from all MPI tasks' stack traces . This view reveals a characteristic known by the user: the job has five first-level process clusters: 8 coupler tasks; 16 land model

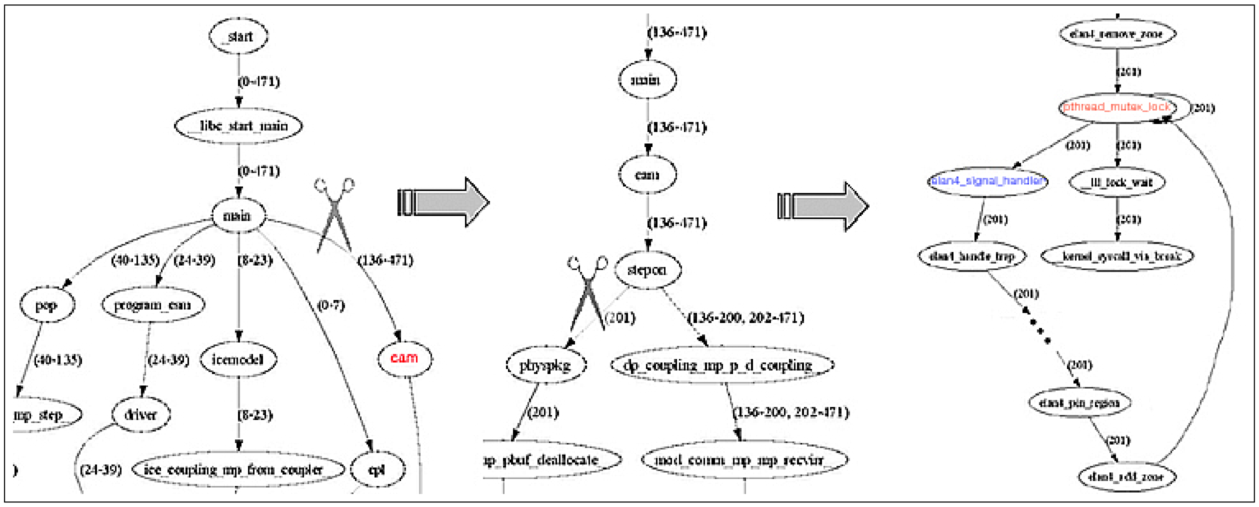


Figure 1: Manual debugging practice on the first CCSM hang at 472 MPI tasks

tasks; 16 ice model tasks; 96 ocean model tasks; and 336 atmosphere model tasks. These five application modules are depicted by the five sub-graphs rooted at `main`. The center graph of Figure 1 suggests an interesting behavior: the set of 336 atmosphere tasks form two clusters rooted at the `stepon` function node. One cluster consists of task 201; the other contains all other tasks of the atmospheric model. A comparison of the details of task 201 and task 200, an arbitrary representative of the second cluster, indicates a possible deadlock in task 201. Specifically, task 201 reentered the `pthread_mutex_lock` function in the ELAN4 library after an asynchronous interrupt due to a UNIX signal handler invocation, as indicated by the rightmost graph in Figure 1. The other tasks were stalled polling at `MPI_Waitall` as a consequence of the deadlock in task 201. Multiple stack trace snapshots taken over time for task 201 were almost identical, further supporting the deadlock theory.

This high-level analysis aided our second round of debugging during which we immediately noticed that two tasks formed a similar anomalous cluster. TotalView analysis revealed that the anomalous tasks were attempting to re-lock a mutex. Further, the mutex type was `PTHREAD_MUTEX_NORMAL`, which results in a deadlock under Linux. We theorize that Quadrics QsNet^{II} software's internal interconnect management raised a UNIX signal. Regardless, it is clear that the `elan4_remove_zone` function must be made signal safe.

1.2 Integer Overflow in ViSUS

Visualization Streams for Ultimate Scalability (ViSUS) is a research project that develops data streaming techniques for progressive processing and visualization of large scientific data sets [22]. About a year ago, an LLNL developer reported a hang during a scaling test on BG/L. The hang occurred deterministically at scales of at least 8192 tasks. Since the smallest scale exhibiting the defect was already far beyond the capability of any available debugger, the developer was forced to debug using print statements. Ad-hoc parsers were written to process the voluminous output; every change to the printed output demanded changes to the parsers. Ultimately, this strategy proved ineffective for analyzing the defect at this scale.

Eventually, a more scalable version of TotalView became available. Even so, the scale continued to pose a challenge. Debugging 8,192 tasks was twice TotalView's warranted upper limit on BG/L. However, careful provisioning and the avoidance of non-scalable operations allowed the tool to be used at the requisite scale. Once a debugging tool was available, diagnosis of the root cause of the hang was relatively simple. The debugger allowed the discovery of an unintentionally formed infinite loop due to 32-bit integer overflow.

These case studies demonstrate several facts. Some program errors only show up beyond certain scales. Further, the errors may be non-deterministic, and thus difficult to reproduce. Stack traces can provide very useful insight, but deficiencies exist in current tools for collecting and rendering this information: they either do not provide enough information or cannot run effectively at the requisite scale. STAT addresses these deficiencies by using a scalable, lightweight approach for expediently collecting, analyzing, and rendering the information necessary to reduce the problem to a manageable subset of the large-scale job.

2 Scalable Stack Trace Analysis

Motivated by manual debugging practices as demonstrated by the case studies of Section 1, we propose an automated, lightweight technique for scalably reducing problem exploration spaces. Our approach analyzes application process stack traces to discover process equivalence classes – processes exhibiting similar behavior based on the functions they are executing. This approach facilitates scalable data analysis as well as scalable visualizations that guide the user in the diagnosis process. In this section, we detail our stack trace analysis approach. We begin by discussing rudimentary techniques for processing stack traces over a distributed application space – some of which are implemented by existing tools. We then expand these techniques into our technique that uses stack traces collected over time and (process) space to identify application behavior.

For this discussion, we introduce a simple MPI program that we use as the target of our problem diagnosis. In this program, process ranks are organized into a virtual ring within which, each process performs an asynchronous receive operation from its predecessor in the ring and an asynchronous send to its successor. Each process then blocks for the I/O requests to complete (via `MPI_Waitall`). A whole program synchronization point (via `MPI_Barrier`) follows the ring communication. The code includes an intentional bug that causes one MPI process never to reach its send operation.

Singleton Stack Traces Figure 2 illustrates the fundamental data object in our analysis, a stack trace, which depicts the caller/callee relationships of the functions being executed by a process. In our model, we distinguish functions by invocation paths; in other words, if the same function is invoked multiple times via different call paths, it occurs multiple times in our stack trace. We believe that functions invoked via different call paths (including recursively) may demonstrate different application semantics to the user that would not be visible without this distinction. Also, distinction by invocation means that merged stack traces result in a tree, which is easier to analyze both algorithmically and manually than a more general directed graph. Such singleton stack traces are supported by most if not all debuggers, typically using a textual representation. Singleton traces do not allow a user to evaluate the behavior of a large application effectively: clearly, a thousand processes would generate a thousand stack traces – well beyond the threshold of human comprehensibility.

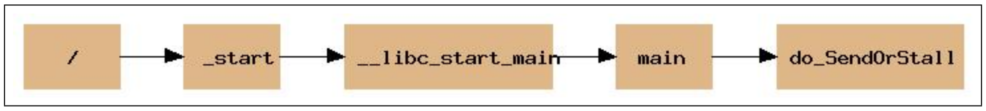
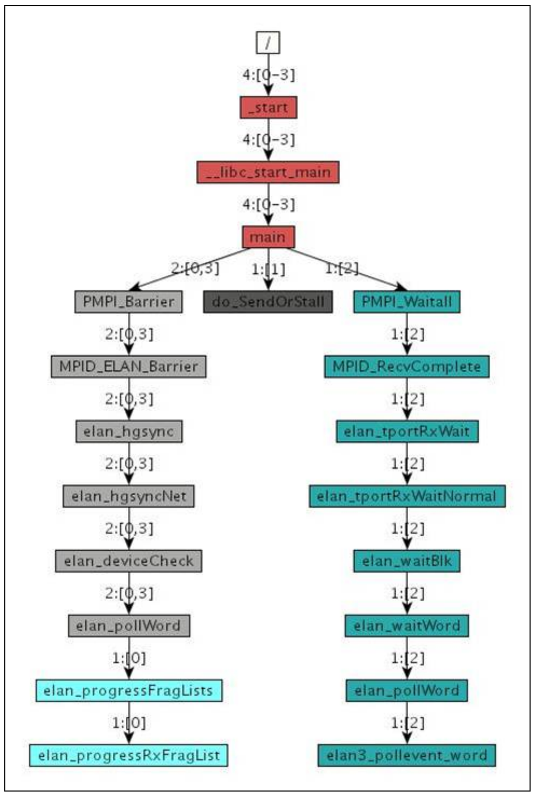
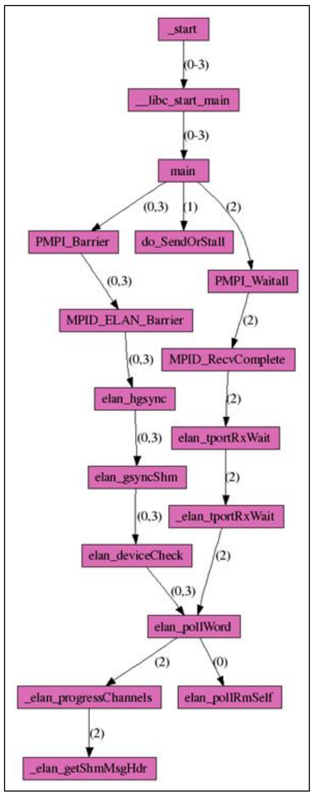


Figure 2: A stack trace showing caller/callee relationships of functions being executed.

2D-Trace/Space Analysis To address the deficiency of singleton traces, tools like TotalView and Prism support what we call a 2D-Trace/Space analysis, merging a single stack trace from each application process into a call graph prefix tree that maps stack traces into the application processes' space. The presumption, as well as the common reality, is that there will be significant overlap amongst the individual stack traces such that many processes will merge into a relatively small call graph prefix tree. This data object is illustrated in Figure 3, which compares our call graph prefix tree to TotalView's call graph. To help users quickly focus on a small number of individual processes, STAT analyzes the traces to depict process equivalence classes – processes in the same class are shown in the same color. In contrast, TotalView presents this information using a call graph that does not distinguish the invocation path for each function leading to a general directed graph that cannot be as easily partitioned into equivalence classes.



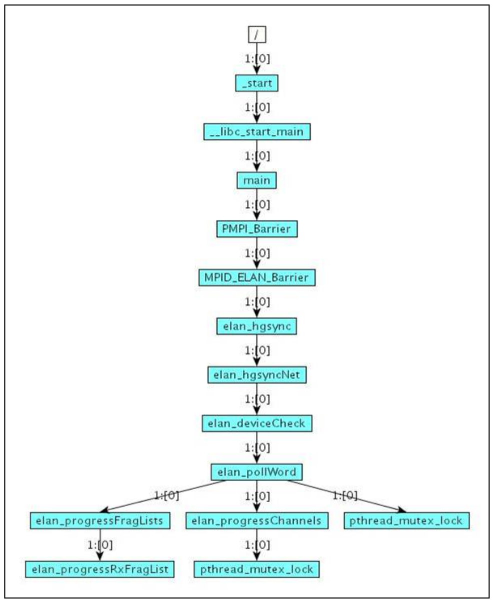
(a) STA Tool



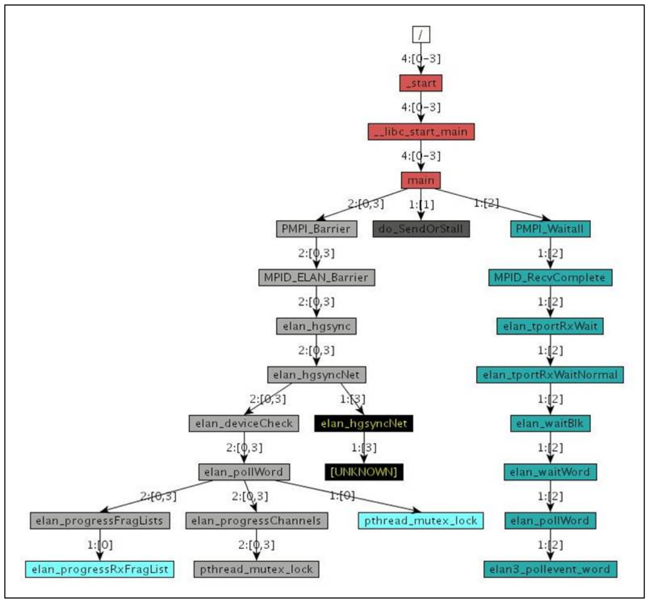
(b) TotalView

Figure 3: 2D-Trace/Space call graphs from (a) STAT, distinguishing nodes based on call paths, and (b) TotalView, representing nodes without call path history. STAT uses its tree representation to create equivalence classes of processes and displays them intuitevely.

2D-Trace/Time Analysis While our 2D-Trace/Space call graph prefix trees are well-suited for analyzing static scenarios like examining core dumps, they do not include the temporal information necessary to help answer questions about application progress, deadlock/livelock conditions, or performance. To address these issues, we introduce 2D-Trace/Time analyses: we merge and analyze a sequence of stack traces from a single process collected over a sampling interval. Figure 4(a) shows how this analyses renders a profile of a process's behavior over time. Allowing the user to specify the length of the sampling period in terms of the number of samples to collect and the interval between samples, STAT allows the user to control the granularity at which traces are sampled and thus tool overhead as well as the coverage of the collected profile.



(a) 2D-Trace/Time Call Graph



(b) 3D-Trace/Space/Time Call Graph

Figure 4: STAT Call Graph Prefix Trees. (a) A 2D-Trace/Time call graph prefix tree showing the call paths that a process has exhibited during a sampling period. (b) A 3D-Trace/Space/Time call graph prefix tree showing a global application profile and distinguishing unique behaviors as process equivalence classes.

3D-Trace/Space/Time Analysis 2D-Trace/Time analysis allows a user to profile an individual process quickly, just like singleton stack traces; however, it is an ineffective method for understanding the collective behavior of many thousands of processes. The solution is a 3D-Trace/Space/Time analysis that combines both previous analysis techniques: we merge and analyze sequences of stack traces from multiple processes to assemble a global application profile. As seen in Figure 4(b), the identified process equivalence classes distinguish the sets of process behaviors exhibited by the application.

Our visualizations are influenced by Miller's criteria for good parallel program visualization [19]. We briefly discuss how the visualization results of our 3D-Trace/Space/Time analysis observe some of these guiding principles:

- *Visualizations should guide, not rationalize*: By presenting a profile of the applications behavior for the sampling period and identifying equivalence classes of similar process behavior, the user quickly can extract new knowledge about program behavior.

- *Scalability is crucial*: For thousands of processes, it would not be scalable to display call graphs for each process. Instead, we merge these graphs into a single, compact tree. Further, as discussed below, the use of color to distinguish classes of process behavior helps the user to navigate complicated graphs.
- *Color should inform, not entertain*: In our visualizations, color is used to distinguish classes of process behavior. Nodes of identical color represent stack frames from groups of processes executing the same instructions in the program.
- *Visualizations should provide meaningful labels*: Node labels appropriately identify the function invoked at each level of the stack trace, and edge labels identify the number of processes (and their ranks) that have taken a call path starting at the root node and traversing that edge. Together the labels can be used to identify the behavior of a single process or assemble a global picture of dominant application behaviors.

3 STAT: The Stack Trace Analysis Tool

In this section, we describe STAT's scalable implementation of 3D-Trace/Space/Time analysis. We present the tool's implementation and performance as well as its application to the case studies of Section 1. We also discuss scalable visualization and future enhancements.

3.1 STAT Design and Implementation

STAT is composed of three main components: the tool front-end, the tool daemons, and the stack trace analysis routines. The front-end controls the collection of application stack trace samples by the tool daemons, and the collected stack traces are processed by our stack trace analysis routines. The front-end renders the result, a single call graph prefix tree. STAT utilizes MRNet [24], a TB$\bar{O}$N-based multicast/reduction network infrastructure developed at the University of Wisconsin. A TB$\bar{O}$N (tree-based overlay network) is a network of hierarchically-organized processes that exploits the logarithmic scaling properties of trees to provide scalable application control, data collection, and data analyses. The MRNet tree of processes forms the infrastructure that the STAT front-end and back-ends use to communicate. MRNet's data filter abstraction is used to implement the stack trace analysis algorithm, which executes at the tree's parent nodes on data streams from children nodes. Using this process organization, the analysis computation is distributed: tool daemons merge node-local samples, and the TB$\bar{O}$N combines local traces into a whole-program view.

The front-end is STAT's driver. The front-end's first responsibility is to instantiate the MRNet tree and tool daemon back-ends. Once the TB$\bar{O}$N is established, the front-end, via the MRNet API, dynamically loads the TB$\bar{O}$N processes with STAT's custom filter, which processes the collected stack trace samples. The front-end then instructs each daemon to attach to the application processes local to that daemon's compute node. After the attach phase, the front-end instructs each daemon to collect stack trace samples from the application processes for a sampling period defined by a sample count and an interval between samples. The collected samples are propagated through the TB$\bar{O}$N, and the result is 3D-Trace/Space/Time call graph prefix tree. Finally, the front-end color-codes the process equivalence classes and exports the graph as an AT&T dot format file.

Each STAT back-end is a simple daemon that implements the following capabilities: attach to application processes; sample process stack traces; merge/analyze locally-collected samples; and propagate analysis results up the tree. The Dyninst dynamic instrumentation library [8] allows us to debug unmodified application processes. Each tool daemon, including the back-end, processes locally collected stack trace samples with the core function that implements the stack trace merge and analysis routines before propagating results the through the TB$\bar{O}$N. This function is the filter that the internal TB$\bar{O}$N processes use.

The STAT filter inputs a vector of packets, one from each child of the executing internal node, and outputs a single packet. Each STAT packet encapsulates a call graph prefix tree. The core function of the filter inputs two call graph prefix trees and outputs the result of merging them into a single call graph prefix tree. This function is used to merge all the input trees into a single tree that is then further propagated through the TB$\bar{O}$N until a single global tree is obtained at the STAT front-end.

3.2 Performance Evaluation

We test STAT's performance on Thunder, a 1,024 node cluster at LLNL. Each node has four 1.4 GHz Intel Itanium2 CPUs and 8 GB of memory. The nodes are connected by a Quadrics QsNet^{II} interconnect using the Quadrics Elan 4 network processor.

For our experiments, we debug the MPI message ring program described in Section 2 at various scales. The application is run on an allocation of nodes with four MPI tasks per node. For debugging, STAT daemons must be co-located with the application processes. We place one tool daemon process on each node of the application's allocation; that is, one tool daemon debugs four application processes. The rest of the TB$\bar{O}$N, the front-end and internal nodes, are placed on a separate allocation.

We evaluate STAT's performance by measuring the latency from the front-end's broadcast to collect stack trace samples until the global call graph prefix tree is available at the front-end. We omit the sampling duration since this is determined by the number of samples and sampling interval as chosen by the user. For our experiments, we collected 10 samples from each process with a 1 second interval between samples. Micro-benchmark experiments show that on average it takes about 1.3 seconds for STAT to collect the first stack trace from an application, and about 0.06 seconds for subsequent samples. Dyninst's parsing of the application image for more accurate traces causes the high initial sampling latency. STAT caches this information for subsequent samples.

We compare the performance of two different types of tree topologies: 1-deep trees, the standard tool organization in which the front-end is directly connected to the tool-daemons, and a 2-deep tree with an intermediate level of internal nodes[1]. The 2-deep trees are completely balanced: all parent processes have the same number of children. The results of our experiments debugging up to 3844 application processes are shown in Figure 5. As expected, as we scale up the size of the application being debugged, the latency of the 1-deep tree grows linearly with the number of application processes being debugged. The latency of the 2-deep trees increase at a much slower rate due to the controlled TB$\bar{O}$N fan-out.

[1]As scale increases, it is natural to increase the depth of the tree to maintain scalable performance. For our experiments, trees of depth two were sufficient.

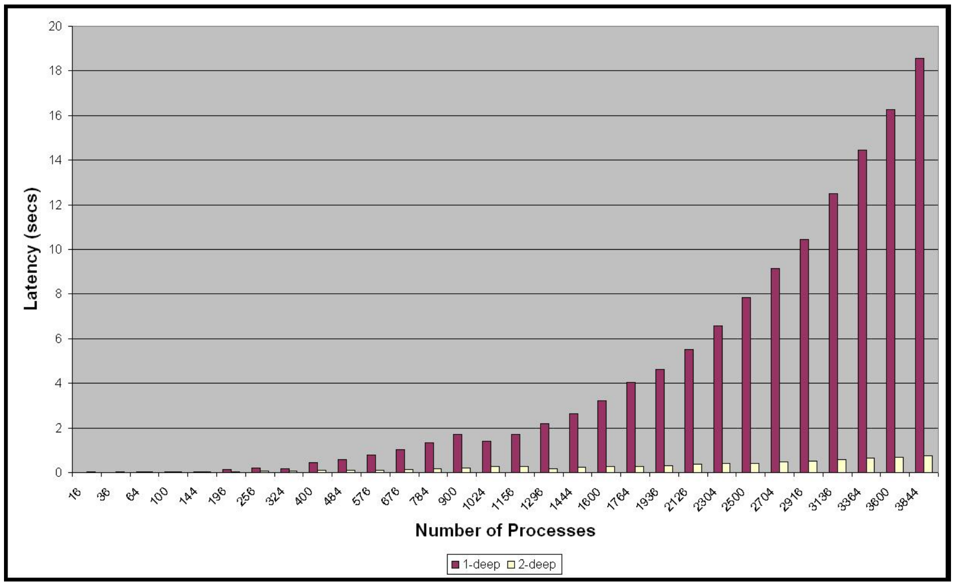


Figure 5: STAT Performance. The flat tree directly connects the tool front-end to the back-end daemons. The 2-deep tree has an intermediate level of TB$\bar{O}$N processes.

3.3 STAT on Real Applications

We apply STAT to two real large-scale debugging cases presented in anomalies. Section 1. The results of our empirical evaluation demonstrate that our tool provides insights into dynamic behavior of real anomalies.

3.3.1 Applying STAT to ViSUS

We apply the tool to ViSUS after reintroducing the bug into the code. We consider a reduced scale, 162 tasks, based on a subsequently found way to reproduce the hang at much lower scales. When the program hangs, we apply STAT to the running job to attach to all tasks, sampling ten stack traces with an interval of 100 milliseconds between samples.

Figure 6 shows a portion of the STAT graph capturing both spatial and temporal behavior. The figure indicates that all tasks behave homogeneously before entering the `composite` function. A node coloring scheme then reveals the first-level cluster refinement at this function, yielding three sub-clusters. The label of each link forked off of the `composite` node shows the rank membership for each sub-cluster: a majority of tasks executes the `handle_src` function, while only 14 tasks invoke `handle_cmp` and one task, rank 1, executes both `handle_cmp` and `handle_mux`. This cluster refinement correctly uncovers an important characteristic of ViSUS: all tasks form a virtual tree network to composite local data efficiently into global data. While all tasks participate in producing task-specific local data (`handle_src`), only a small subset composites them. Once all local data are composited, only the root task of the tree writes it to a file in `handle_mux`. The graph shows that the anomaly does not perturb this normal cluster structure.

The removed bottom of the STAT graph indicates that only those member tasks of the second cluster carry out MPI communication calls. Unlike the CCSM case, however,

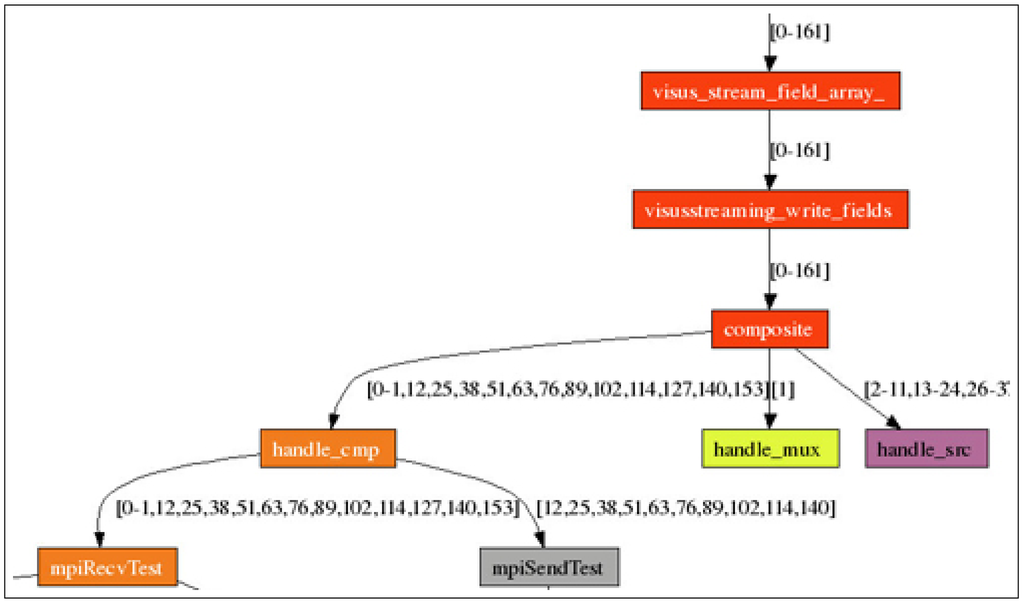


Figure 6: 3D-Trace/Space/Time STAT graph for ViSUS

examining further refinements of that cluster does not identify outlier tasks that would be a likely root cause of the hang. The graph naturally guides our attention rather to the `composite` function next. Examining a small code section of the `composite` function provides an insight into a possible infinite loop: `handle_cmp`, `handle_src` and `handle_mux` are enclosed in a while loop. For further root cause analysis, we pick three representative processes, respectively, rank 0, rank 1 and rank 2, one from each cluster. We then investigate those three tasks with a full-featured debugger. We then quickly locate a 32 bit integer overflow by applying debugging features such as lock-stepping and evaluation points. This 32 bit integer overflow prevented the `handle_src` function from ever returning a loop terminating code.

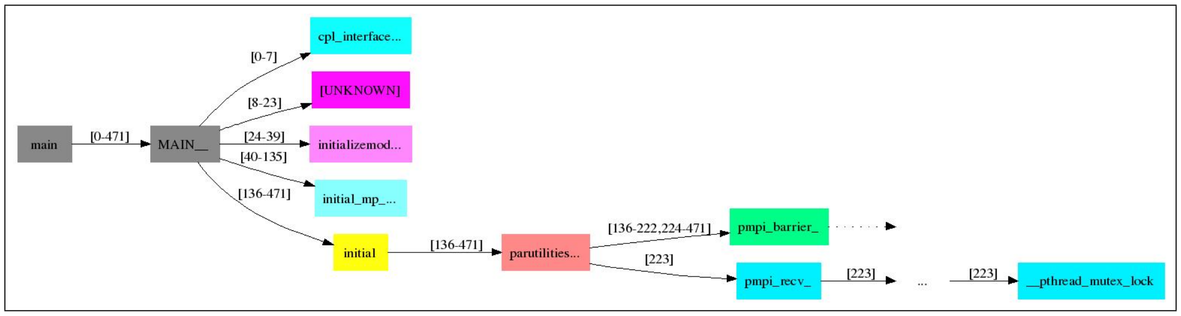


Figure 7: 3D-Trace/Space/Time STAT graph for CCSM

3.3.2 Applying STAT to CCSM

We apply STAT to CCSM when it hangs with the original configuration. STAT's temporal behavior analysis effectively guides detection of the deadlock condition in an MPI task. Figure 7 is the graph capturing both spatial and temporal aspects of the program. The graph depicts that the offending task does not make progress over the sample period, after separating itself from the other tasks at the `partutilitiesmodule_mp_parexchagevec` function. For example, its call path does contain any function node that has more than one immediate child node.

3.4 Scalable Visualization

Even though our stack trace analysis results in drastically reduced graphs compared to visualizing all nodes' stack traces individually, the previous examples have shown that even those reduced graphs can be large and complex. In most cases, however, the detection of equivalence classes and their coloring coding provides enough structure in the display. If this is insufficient, we can further reduce the complexity of the presented graphs without losing detailed information. One way to reduce the size of the displayed graphs is to provide optional pruning. With this method, we will default to displaying an easily comprehensible representation, the pruned graph. However, we will still allow users to expand pruned segments on demand to analyze relevant components more deeply.

The use of trees instead of directed graphs in STAT implicitly leads to the generation of hierarchical equivalence classes: each subtree of the complete stack trace information can be seen as one equivalence class, which can then be further subdivided into sub classes represented by lower branches in the tree. This property provides a natural abstraction for pruning larger trees since any pruning of a subtree results in aggregating finer-grain equivalence classes into a more coarse grain one, but will not destroy the abstraction produced by STAT. This mechanism therefore provides a true level of detail selection mechanism without the possibility of misguiding the user at higher levels of abstraction. A similar approach based on general directed graphs would require complex and expensive node coloring schemes.

An additional pruning criteria for call graphs is any transition from the user's application to library functions that are normally opaque to the user. For example, we could prune the subgraphs of any nodes that represent MPI routines. As the examples shown above demonstrate, this will significantly reduce the size of the graphs in many cases since call graph variations are likely to occur within the MPI library.

This idea of pruning trees can further be extended to let the user dynamically prune the tree at any level. At first the user will only see a very high-level view of a tree (for example, a fixed number of branches starting at the root node) and can then interactively select to expand any existing branch. Such an interactive selection of zoom detail can be supported within a traditional graph viewer or through a file browser like display or other hierarchical representations that support dynamic expansion and contraction of hierarchy levels.

A second major source for complexity in the graph visualization is the list of process ranks associated with each edge. Especially for large node counts, these lists can get long and fragmented and hence difficult to interpret. Since the concrete list is often not required to obtain a first overview, we suggest to replace the textual representation with a graphical one, for example, in the form of a bar code or digital fingerprint. This will allow users to identify and compare node sets quickly and visually without having to compare individual rank numbers. Only on request, for example, to further analyze leave nodes, these graphical representations will be expanded to the actual rank lists. Work on this feature is currently in progress and will be included in the final paper.

3.5 Future Enhancements

Our current tool provides a powerful abstraction of the state of a parallel application, but the actual diagnosis of a problem or the detection of an anomalous process is still up to the user of the tool. As a next step, we plan to integrate the tool with analysis techniques to automate the process of problem identification. For example, statistical clustering of call trees from all ranks can help identify processes with distinct and potentially anomalous behavior. Similarly, we can integrate techniques to detect other problems typically found in message passing codes like load imbalances or excessive waits.

To further support such automatic analysis, we plan to extend our tool to compare data from multiple runs. This will add an additional dimension to our stack traces and, combined with a manual tagging of runs, for instance, as good or erroneous, provide an easy way to identify the cause of anomalies.

4 Related Work

Our work focuses on scalable strategies for diagnosing problems in large scale parallel and distributed applications. A myriad of tools and research in this area exists. In this section, we present related work in three categories: parallel debuggers; problem diagnosis via data analysis; and automated debugging techniques.

Over the years, many parallel debuggers have targeted a variety of programming languages and hardware platforms including Fx2 [2], Ladebug [5], Mantis [18], mpC [4], p2d2 [13], pdbx [14], Prism [25], and TotalView [12]. All at least have the capability of viewing individual process stack traces. Both the Prism and TotalView debuggers aggregate individual process stack traces into a single call graph tree. However, neither tool accommodates the time-varying views of call graphs necessary to answer questions about the programs behavior over time. Both also use a non-scalable, single level hierarchy with the tool front-end directly connected to the back-end processes. While Ladebug does not support aggregated stack traces, it does use a TB$\bar{O}$N and data aggregation for responsive tool control and data collection. Finally, Mantis supports a colored process grid visualization in which node colors reflect process status, for example, running, stopped, error. The fully-functional nature of these debuggers is partially what renders them non-scalable. We view our lightweight STA approach as complementary to such tools: we scalably reduce the exploration space so users can apply these heavy-weight tools.

Several projects have investigated the use of statistical methods for automated application analysis. In the work of Dickenson et al [11], they collect call profiles from program runs and use several distance metrics to cluster similar profiles. Profiles from each cluster are analyzed to determine whether or not the cluster represents an anomaly. Yuan et al [27] apply a supervised classification algorithm to classify system call traces based on their similarity to previously analyzed traces with already diagnosed problems. Magpie [6] uses a string-edit distance metric, a measure of two string's difference, to cluster events. Events that do not belong to a sufficiently large cluster are considered anomalous. Mirgorodskiy et al [20] also use distance metrics to categorize data from control-flow traces, identifying traces that substantially differ from the others. They perform root cause diagnosis by, for instance, identifying which call path contributed most to the profile dissimilarity. Pinpoint [9] uses both clustering and decision trees on client-server traces to correlate request failures with any failures occurring in the components used to service the requests. Finally, Ahn and Vetter use multivariate statistics to cluster large performance data sets for scalable analysis and visualization [3]. Like our stack trace analysis, these approaches analyze collected run-time data to identify anomalies. These approaches are designed to run post mortem – after the application has exited. Our analysis is light-weight making it scalable and suitable for diagnosing program behavior as the application is running.

Several researchers have explored techniques to locate specific types of errors automatically. Umpire [26], Marmot [16], and the Intel Message Checker [10] trace MPI executions and and detect violations of the MPI standard, including resource leaks, deadlocks and type mismatches. Intel Thread Checker (formerly Assure) [15] simulates OpenMP directives and automatically identifies race conditions and other common OpenMP errors. Finally, several tools, including Valgrind [21] and TotalView [12] automatically detect memory usage errors including leaks and stray writes. All of these tools provide precise information about the locations of coding errors at the cost of significant overhead; they are not intended to be applied to production, large scale jobs but rather as final step in the development process. Rather than expensive techniques that locate certain classes of errors precisely, we explore fast, scalable, automated techniques to locate problem coding regions and tasks.

5 Conclusion

We have presented the design and implementation of STAT, the scalable Stack Trace Analysis Tool. This tool addresses an issue that is becoming increasingly important for large scale parallel platforms: how do we diagnosis program errors that emerge only under production runs at very high processor counts. Specifically, we provide a method for assembling stack traces across the processes of a parallel job into a 3D-Trace/Space/Time diagram. This diagram captures the hierarchical equivalence classes of execution paths across those processes, allowing programmers to focus on subsets of tasks and code regions quickly.

STAT builds upon the scalable MRNet tool infrastructure. STAT's tree of intermediate tool processes assembles 3D-Trace/Space/Time diagrams in a highly scalable manner. Our performance results demonstrate that STAT significantly improves our ability to examine stack traces across a parallel job: it achieves sub-second latencies at thousands of processes, compared to the multiple seconds with existing tools. This performance is critical since the current high latencies are sufficient to make most programmers give up for all but the most mission critical tasks. Even more importantly, we have presented several case studies of real world application debugging experiences and demonstrated that STAT can substantially improve our ability to locate the root causes of these errors quickly and accurately.

References

[1] Top 500 Supercomputer Sites. http://www.top500.org.

[2] Absoft. Fx2 Debugging Solution for Fortran, C/C++ Compilers. http://www.absoft.com/Products/Debuggers/fx2/fx2_debugger.html.

[3] D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. *SC '02*, pp. 1–16, Baltimore, Maryland, 2002.

[4] ATS Software. mpC Workshop. http://www.atssoft.com/products/mpcworkshop.htm.

[5] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a Traditional Debugger to Debug Massively Parallel Applications. *Journal of Parallel and Distributed Computing*, 64(5):617–628, 2004.

[6] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. *OSDI*, pp. 259–272, 2004.

[7] M. B. Blackmon, B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, and J. Hurrell. The Community Climate System Model. *Bulletin of the American Meteorological Society*, 82(11):2357–2376, 2001.

[8] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[9] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. *International Conference on Dependable Systems and Networks (DSN '02)*, pp. 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[10] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, Scalable Debugging of MPI Programs with Intel Message Checker. *Second International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS 2005)*, St. Louis, MO, May 2005.

[11] W. Dickinson, D. Leon, and A. Podgurski. Finding Failures by Cluster Analysis of Execution Profiles. *23rd International Conference on Software Engineering (ICSE '01)*, pp. 339–348, Washington, DC, USA, 2001. IEEE Computer Society.

[12] Etnus, LLC. TotalView. http://www.etnus.com/TotalView.

[13] R. Hood. The p2d2 Project: Building a Portable Distributed Debugger. *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pp. 127–136, Philadelphia, PA, 1996.

[14] IBM. Parallel Environment (PE). http://www-03.ibm.com/systems/p/software/pe.html.

[15] Intel. Intel(R) Thread Checker. http://www.intel.com/support/performancetools/threadchecker.

[16] B. Krammer, M. S. Muller, and M. M. Resch. MPI I/O Analysis and Error Detection with MARMOT. *EuroPVM/MPI 2004*, Budapest, Hungary, September 2004.

[17] Lawrence Livermore National Laboratory. Thunder. http://www.llnl.gov/linux/thunder.

[18] S. S. Lumetta and D. E. Culler. The Mantis Parallel Debugger. *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96)*, pp. 118–126, Philadelphia, PA, 1996.

[19] B. P. Miller. What to Draw? When to Draw?: An Essay on Parallel Program Visualization. *Journal of Parallel and Distributed Computing*, 18(2):265–269, 1993.

[20] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller. Problem Diagnosis in Large-Scale Computing Environments. *SC 06*, Tampa, Florida, November 2006. To appear.

[21] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[22] V. Pascucci. ViSUS: Visualization Streams for Ultimate Scalability. Technical Report UCRL-TR-209692, Lawrence Livermore National Laboratory, February 2005.

[23] Quadrics Supercomputers World, Ltd. Quadrics Documentation Collection. http://www.quadrics.com/onlinedocs/Linux/html/index.html.

[24] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. *SC '03*, Phoenix, AZ, 2003.

[25] Thinking Machines Corporation. Prism User's Guide, December 1991.

[26] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. *2000 ACM/IEEE Conference on Supercomputing (SC '00)*, Dallas, Texas, United States, November 2000.

[27] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. *EuroSys 2006*, Leuven, Belgium, April 2006.