LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-SR-499454

# Lightweight and Statistical Techniques for Petascale Debugging: Correctness on Petascale Systems (CoPS) Preliminry Report

B. R. de Supinski, B. P. Miller, B. Liblit

September 16, 2011

## Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

# Lightweight and Statistical Techniques for Petascale Debugging: Correctness on Petascale Systems (CoPS) Preliminry Report

Bronis R. de Supinski, Barton P. Miller and Ben Liblit

## 1 Project Summary

Petascale platforms with O($10^5$) and O($10^6$) processing cores are driving advancements in a wide range of scientific disciplines. These large systems create unprecedented application development challenges. Scalable correctness tools are critical to shorten the time-to-solution on these systems. Currently, many DOE application developers use primitive manual debugging based on *printf* or traditional debuggers such as TotalView or DDT. This paradigm breaks down beyond a few thousand cores, yet bugs often arise above that scale. Programmers must reproduce problems in smaller runs to analyze them with traditional tools, or else perform repeated runs at scale using only primitive techniques. Even when traditional tools run at scale, the approach wastes substantial effort and computation cycles. Continued scientific progress demands new paradigms for debugging large-scale applications.

The Correctness on Petascale Systems (CoPS) project is developing a revolutionary debugging scheme that will reduce the debugging problem to a scale that human developers can comprehend. The scheme can provide precise diagnoses of the root causes of failure, including suggestions of the location and the type of errors down to the level of code regions or even a single execution point. Our fundamentally new strategy combines and expands three relatively new complementary debugging approaches. The Stack Trace Analysis Tool (STAT), a 2011 R&D 100 Award Winner, identifies behavior equivalence classes in MPI jobs and highlights behavior when elements of the class demonstrate divergent behavior, often the first indicator of an error. The Cooperative Bug Isolation (CBI) project has developed statistical techniques for isolating programming errors in widely deployed code that we will adapt to large-scale parallel applications. Finally, we are developing a new approach to parallelizing expensive correctness analyses, such as analysis of memory usage in the Memgrind tool.

In the first two years of the project, we have successfully extended STAT to determine the relative progress of different MPI processes. We have shown that the STAT, which is now included in the debugging tools distributed by Cray with their large-scale systems, substantially reduces the scale at which traditional debugging techniques are applied. We have extended CBI to large-scale systems and developed new compiler-based analyses that reduce its instrumentation overhead. Our results demonstrate that CBI can identify the source of errors in large-scale applications. Finally, we have developed *MPIecho*, a new technique that will reduce the time required to perform key correctness analyses, such as the detection of writes to unallocated memory. Overall, our research results are the foundations for new debugging paradigms that will improve application scientist productivity by reducing the time to determine which package or module contains the root cause of a problem that arises at all scales of our high end systems.

While we have made substantial progress in the first two years of CoPS research, significant work remains. While STAT provides scalable debugging assistance for incorrect application runs, we could apply its techniques to assertions in order to observe deviations from expected behavior. Further, we must continue to refine STAT's techniques to represent behavioral equivalence classes efficiently as we expect systems with millions of threads in the next year. We are exploring new CBI techniques that can assess the likelihood that execution deviations from past behavior are the source of erroneous execution. Finally, we must develop usable correctness analyses that apply the *MPIecho* parallelization strategy in order to locate coding errors. We expect to make substantial progress on these directions in the next year but anticipate that significant work will remain to provide usable, scalable debugging paradigms.

## 2 STAT: The Stack Trace Analysis Tool

STAT, a scalable lightweight tool that won the R&D 100 Award this year, identifies process equivalence classes, groups of processes that exhibit similar behavior. It samples stack traces over time from each task of the parallel application, which it merges into a call graph prefix tree as shown in Figure 1. The call graph prefix tree intuitively represents the application's hierarchical behavior classes over space and time. Our graphical representation indicates these equivalence classes by node colors and the edge labels that identify which tasks are in the class. The developer can use these classes to reduce the number of tasks and to narrow down the code regions upon which to apply traditional techniques such as a detailed code tracing with a full-featured debugger. In this section, we describe the basic architecture of STAT and then detail an important advance made under CoPS.

### 2.1 STAT Background

Conceptually, STAT has three main components: the front end, the tool daemons, and the stack trace analysis routine. The front end controls the collection of stack trace samples by the tool daemons, and the stack trace analysis routine processes the collected traces. Each component of STAT leverages scalable tool infrastructures in order to achieve its primary design goal of scalability. It uses LaunchMON [1] to co-locate tool daemons scalably with the distributed target application processes by coordinating with the native resource manager or scheduler. The Stack Walker API is used to construct stack traces with very low overheads. Finally, the MRNet tree based overlay network (TBŌN) reduces the trace data and processing loads on STAT's front end through a custom MRNet filter that efficiently merges the stack traces.

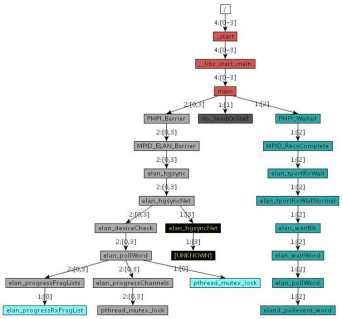

Figure 1: STAT stack prefix tree

The development and deployment of STAT has been very successful. The tool is effective in debugging real-world scientific applications such as the Community Climate System Model (CCSM) [15] by substantially reducing the search space to a handful of representative tasks for anomalies such as deadlock, livelock and infinite loops [5]. Our initial results demonstrated that the basic architecture and intelligent implementation of the filter routines support scalability to four thousand tasks.

Subsequently, we developed a STAT emulator that supported exploration of tool performance at two orders of magnitude more tasks than the number of processors available on the system [28]. These experiments led to design modifications that made STAT the first parallel debugging tool to scale to tens of thousands processors while achieving execution times that are appropriate for interactive use. More recently, we demonstrated that additional design research and tool performance analysis led to an understanding of new issues that arise for scalable debugging of over one hundred thousand tasks [26]. The critical lesson of these studies is that each order of magnitude increase in system size will present new hurdles that must be addressed with additional research. Our ability to emulate significantly large systems will allow us to understand those issues prior to deployment of the expected million processor systems, leading to solutions that are available when application developers most need them: when the system first becomes available for use.

While emulation is an essential aspect of our strategy to provide timely debugging solutions, we have conducted experiments on the world largest supercomputers as we incrementally scaled up STAT. These experiments stressed all components and guided our optimizations that enable it to function at those scales. Our lessons included an understanding of the importance of data structures that allow the tool to exploit TBŌNs effectively at extreme scale. Figure 2 shows the stack traces merge time results from BG/L with our latest bit vector optimization in comparison to the original implementation that used an $n$ bit vector at each node

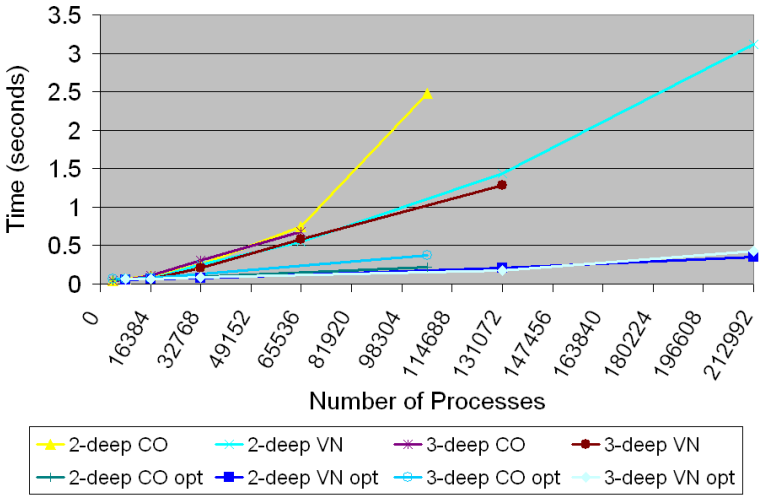


Figure 2: Optimized bit vector STAT merge time versus original bit vector STAT merge time

of the TBŌN to represent the global rank space of size $n$ in order. The original data structure led to linear tool network bandwidth requirements, which prevented the desired logarithmic scaling for tool performance. Thus, we realized that achieving interactive scalability beyond tens of thousands of tasks required utilizing not only a TBŌN but also distributed data structures that limit the total data volume sent through it. By redesigning the tool to use such data structures, STAT can now merge stack traces from over two hundred thousand tasks on BG/L in under half a second — performance sufficient for interactive use.

## 2.2 Temporal Ordering

STAT originally aggregated stack trace information across all nodes to form equivalence classes of processes, which identify a small subset of processes that can be debugged as representatives of the entire application. While often sufficient, stack traces can be too coarse grain for grouping processes and for understanding the relationship between their execution state. This coarseness may miss critical differences or dependencies. Thus, STAT did not always allow bug isolation and root cause analysis. Instead, we must identify additional data that capture the relative execution progress in each process and that supports accurate mapping of the debug state across all processes.

We have developed a novel, non-intrusive and highly scalable mechanism that refines process equivalence sets and captures the progress of each process. This technique creates a partial order across the process equivalence classes that corresponds to their relative logical execution progress. For sequential code regions, our approach analyzes the control structure of the targeted region and associates it with an observed program location. For loops or other complex control structures, we use static data flow analysis, implemented in the ROSE source-to-source translation infrastructure [11, 37], to determine which application variables capture relative progress. We then extract their runtime values in all processes to refine the process equivalence sets and to determine their relative execution progress. Similarly to previous work [19], our static techniques significantly reduce the amount of runtime data needed for analysis.

Our methodology requires no source code changes; it analyzes the existing code and then uses the results to locate the relevant dynamic application state through the standard debugger interface. Thus, our approach meets a critical debugging requirement: we can apply it to production runs, e.g., after the application aborts or hangs due to deadlock or livelock, thus manifesting a program bug.

```
     /* Exchange ghost points of A */
(5)  exchange_band( ... );
     /* Jacobi sweep, computing B from A */
(6)  sweep_band( ... );
(7)  get_norm( ... );
     /* Exchange ghost points of B */
(8)  exchange_band ( ... );
     /* Jacobi sweep, computing B from A */
(9)  sweep_band ( ... );
(10) get_norm ( ... );
(11) MPI_Allreduce ( ... );
(12) if ( diff <= tolerance )
(13)   converged = 1;

(14) if (converged)
(15)   handle_converged( ... );
(16) else
(17)   handle_not( ... );
```

(a) Solver program fragment

```
(1)  int poisson( ) {

(2)    it = 0;
(3)    converged = 0;

(4)    while ( it < MAX ) {

         ...
         fragment shown in (a)
         ...

(18)     if ( converged )
(19)       break;

(20)     it++;
(21)   }
(22)    return converged;
(23) }
```

(b) Iterative method that contains figure 3a

Figure 3: MPI program solving the Poisson problem iteratively

### 2.2.1 Comparing Process State through Relative Progress

We provide the required process equivalence class refinement and source code insight by distinguishing processes by their *relative progress*, i.e., how far their overall execution has advanced compared to the other processes. Conceptually, we compare the dynamic control flow graphs of the processes up to the current state of execution. Thus, we reduce the relevant state to those variables that capture progress through that control flow. Further, as we will demonstrate, we can identify the relevant variables in most cases automatically through static analysis.

The relative progress of processes in a parallel application is an intuitively simple concept: we want to order processes by how much of the dynamic execution they have completed. We can capture significant detail about the execution history of a process by considering stack traces. A single stack trace can provide simple but significant runtime data about a process's execution. However, it captures limited temporal information: the sequence of functions (i.e., the call path) immediately executed to reach the current state. We could track a sequence of stack traces to capture a much richer notion of progress through the source code, which we could use to compare processes in the same run. This approach is simple but infeasible: we cannot track the stack traces from the beginning of a run in general, particularly for production runs. Thus, we need some alternative representation from which we can deduce progress.

Stack traces alone are insufficient. Even if we tracked stack traces throughout execution, the same stack trace may appear multiple times in this sequence so additional information must distinguish them. Since we cannot practically capture the ordering of a sequence of stack traces, we instead look for variables that partially capture this sequencing information. We illustrate this concept with the fragment of a Poisson solver that figure 3 shows.

In any execution of the fragment in figure 3a, the call to `exchange_band` at (5) always occurs before the call to `sweep_band` at (6). Alternatively, the calls to `handle_converged` at (15) and `handle_not` at (17) cannot be ordered within the fragment since control flow ensures that only one will be executed; in multiple processes they are essentially concurrent. However, the context of a full execution can change these orderings. A process at (5) has progressed further than a process at (6) if it is at later iteration of the loop at (4) (i.e., `it` is larger in the first process). Similarly, the value of `it` can order processes at (15) and (17). table 1 summarizes the orderings based on the call stack information captured at those lines and the values of `it`, in which $it_1$ and

| Line Number | | | |
|---|---|---|---|
| Process 1 | Process 2 | Rel. iter. count | Logical execution order |
| (5) | (6) | $it_1 \leq it_2$ | Process 1 is behind Process 2 |
| (5) | (6) | $it_1 > it_2$ | Process 2 is behind Process 1 |
| (15) | (17) | $it_1 < it_2$ | Process 1 is behind Process 2 |
| (15) | (17) | $it_1 > it_2$ | Process 2 is behind Process 1 |
| (15) | (17) | $it_1 = it_2$ | No relative progress order |

Table 1: Two MPI processes executing the Poisson solver

$it_2$ are the values of `it` in processes 1 and 2.

Formally, we assume a parallel application with $N$ processes that execute the same program (extending our methodology to MIMD applications only requires us to merge the respective state sets). An *execution point* is a relative point of execution as defined by the current stack trace and the state (values) of variables relevant to control flow. We denote the set of all possible execution points as $\Sigma$. The current execution point of a process $i$ is $P_i \in \Sigma$.

**Definition 2.1** *Relative progress is a partial order $\preceq \subseteq \Sigma \times \Sigma$ between two processes, i and j, with $0 \leq i, j < N$, such that $P_i \preceq P_j$ if and only if process j has reached or passed $P_i$ during its execution before reaching $P_j$.*

Intuitively, if a process is executing code with a given control flow variable state that another process could have already executed with the same state then that first process is earlier in its execution than the second. Thus, it has made less progress in the logical execution space. Relative progress is a partial order since it is reflexive, antisymmetric and transitive. Relative progress is distinguished from previously explored partial orderings of parallel processes [9, 24] in that two processes may be ordered even when no chain of messages connects them.

Relative progress provides a theoretical foundation to compare the progress of different processes. However, in order to be practical, we must efficiently and scalably represent a process's progress at any point of its execution. Thus, we define an execution point representation that uniquely identifies any execution point by combining the static program locations of the current execution point with dynamic variable state information. Our representation hierarchically describes each program point relative to its enclosing statement block (e.g., a basic block, loop or function call). Thus, we can locally determine the information required to identify any program location. We augment these program points with dynamic execution information in the form of iteration counts, i.e., how often a particular program point has already been executed. We then combine this information into a tuple from which we can derive a lexicographic order that exactly corresponds to relative progress.

Our representation must treat loops very carefully because loop iteration counts should take precedence in ordering. For the code in figure 3b, we represent the program points (5) and (6) as:

$$(5) \rightarrow \langle (4-1), (iterCount), (5-4) \rangle \rightarrow \langle 3, (iterCount), 1 \rangle$$
$$(6) \rightarrow \langle (4-1), (iterCount), (6-4) \rangle \rightarrow \langle 3, (iterCount), 2 \rangle$$

Here, both program points are in the while loop beginning at (4) in the function starting at (1), denoted by its relative offset, $(4-1)$, which equals 3. Within that loop, (5) has relative offset $(5-4) = 1$ while (6) has offset $(6-4) = 2$. In order to represent the execution points, we also must determine the iteration count (`it`), which we place next to the loop statement's offset. The runtime value must take precedence over the offsets within the body of the loop. We must also ensure that we properly encode incomparable execution points, such as program points in distinct branches of a conditional statement. We represent the program points (15) and (17) in figure 3 as follows:

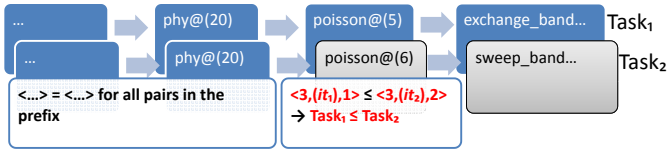


Figure 4: Annotated stack traces for Poisson solver processes; first divergence determines relative progress

$$
\begin{aligned}
(15) \rightarrow \langle (4-1), (iterCount), (14-14)^{\in(14-4)}, (15-14) \rangle \\
\rightarrow \langle 3, (iterCount), 0^{\in 10}, 1 \rangle \\
(17) \rightarrow \langle (4-1), (iterCount), (16-14)^{\in(14-4)}, (17-16) \rangle \\
\rightarrow \langle 3, (iterCount), 2^{\in 10}, 1 \rangle
\end{aligned}
$$

With these representations, (15) and (17) are incomparable when the values of *it* are equal since $0^{\in 10} \parallel 2^{\in 10}$. Figure 4 illustrates relative progress through the Poisson solver for a stack trace representation of two processes. This figure represents each active stack frame with a tuple of the function name and the line number of the callee invocation point. For illustration, we assume that the relative progress of the processes are equal up to the invocation of *poisson*. Thus, the lexicographical order of (5) and (6) determines the relative progress of processes 1 and 2, eliminating a need to evaluate later frames. We exploit this property to determine relative progress efficiently for large scale applications.

### 2.2.2 Automatic Extraction of Application Progress

We have implemented the necessary analysis techniques to determine the progress of a running process in this section. We begin by determining the components of our lexicographic order in general, which we split into two steps. The first step finds necessary offsets while the second determines the variables that we can use for iteration counts. We then conclude this section by showing how to limit the process dynamically only to relevant data, thus making it practical for use on large scale systems.

Our simple but efficient abstract syntax tree (AST) analysis technique translates a program location into the representation described in section 2.2.1. Our system represents a program location by the source file and line number of an instruction. We adopt this simplification since most application developers do not typically write multiple expression statements on a single line. Conceptually, our line number rewriting system is a syntax-directed definition that uses the offset, an iteration count token, and a conditional branch token as inherited attributes for each target high-level language statement. For every production of a statement, the definition associates the statement's offset within the containing compound statement's body and, if appropriate, either of the tokens, to the statement's attribute, prepended with the attribute of the compound statement, which has been produced similarly.

Our analysis uses ROSE [11, 37], a compiler infrastructure that parses high-level language source files and provides mechanisms to manipulate the resulting AST. We use a set of line numbers within a target function as input (we assume the binary is properly compiled with source and line number directives so that the debugging information is consistent with those seen by ROSE) and derive a stack object for each. After our ROSE translator parses the function's source file, it performs a postorder walk on the AST, during which it identifies all nodes that correspond to compound statements and function definitions and tests if any of their line number ranges span our target line numbers. We push any AST node with spanning line numbers onto the corresponding stack. Thus, each stack holds a set of compound statement and function definition nodes that span the associated line number after we walk the AST. These nodes appear on the stack in decreasing order of containment (e.g., the function definition node is on the top of the stack).

Next, we emit a lexicographical representation of each input line number. We first set the baseline to 0 and then pop each AST node off of the stack. For each of these nodes, we emit the displacement between its beginning line number and this baseline, which we then advance by that displacement. Also, we emit a special *non-terminal* token immediately after the offset if the node is a loop statement node. This token serves as a placeholder in which our subsequent analysis can capture the iteration count precedence for statements within the loop body. Similarly, we emit a special token if the node is a conditional branch so that we can identify incomparable execution points.

Static analysis alone cannot fully resolve our lexicographic order when program points are contained in a loop. The first part of our technique only produces a placeholder for the iteration count. Therefore, we devise a static analysis technique that identifies Loop Order Variables (LOVs), key program variables with runtime state from which we can resolve relative progress.

A LOV must satisfy certain properties. Its runtime sequence of values must increase (or decrease) during the execution of the target loop. Further, all processes must assign the same sequence of values. Our LOV analysis identifies program variables that satisfies these requirements.

**Definition 2.2** *Consider a variable x that is assigned a sequence of values during the execution of loop l. Let $x_i(p)$ be the function returning the $i^{th}$ value of x for the task p. Then x is a LOV with respect to l if:*

**(1)** *x is assigned a value at least once every iteration of l;*

**(2)** *the sequence of values assigned to x is either strictly increasing or strictly decreasing during the execution of l (i.e., either $\forall i : x_i(p) > x_{i+1}(p)$ or $\forall i : x_i(p) < x_{i+1}(p)$); and*

**(3)** *$x_i(p)$ is identical for all the tasks (i.e., $\forall p_1, p_2, x_i(p_1) = x_i(p_2)$).*

Our LOV analysis builds on two related branches of static analysis. First, it borrows from the extensive study on loop induction variables including loop monotonocity characterizations of these variables [17, 40, 42]. Unlike our dynamic testing scenario, strength reduction optimizations, loop dependence testing and runtime array bound and access anomaly checking primarily motivate these techniques. Second, our LOV analysis also uses the concept of the *single-valued* variable, a variable that maintains identical values across all MPI tasks through all possible control flows [3, 43]. Those analyses classify variables as *single-valued* or *multi-valued* in order to verify a program's synchronization pattern. Like other induction variable analysis techniques, LOV analysis requires the def-use chain of the function containing the target loop. LOV analysis characterizes uses and definitions of key variables and tests them for ambiguities through the def-use chain.

**Definition 2.3** *The use of the loop invariant variable c with respect to the loop l (i.e., no definition of c inside l reaches to the use) is ambiguous if:*

**(1)** *multiple definitions of c reach to this use (e.g., in* `if (cond1) a ← 1 else a ← 2 endif; do_work(a);`*, the use of* `a` *in* `do_work` *is ambiguous); or*

**(2)** *the only definition of c results from multiple data flows into l (e.g., in* `if (cond1) a ← 1 else a ← 2 endif; b ← a; do_work(b);`*, the use of* `b` *in* `do_work` *is ambiguous); or*

**(3)** *the value of c cannot statically be resolved into a compile-time constant within its containing function (e.g., in* `a ← random_func(); b ← a;` *the use of* `a` *in* `b ← a` *is ambiguous).*

**Definition 2.4** *The use of the loop variant variable v with respect to the loop l (i.e., one or more definitions of v inside l reach to the use) is ambiguous if either a definition of v reaching from outside l is ambiguous by the loop invariant ambiguity rules in definition 2.3 or multiple definitions of v inside l reach to the use.*

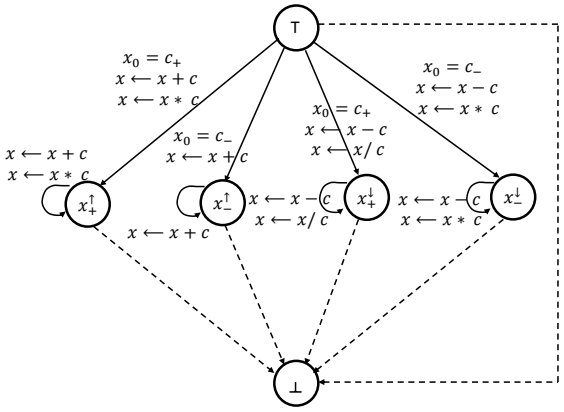


Figure 5: LOV candidate variable state changes; dotted lines represent transitions for unlisted conditions

Our LOV analysis first scans the target loop and constructs a list of expression statements in the loop that assign values to scalar variables of integer types, hence creating new definitions of them. For example, in the case of the C language, the list includes explicit assignment statements like $x = 3$ and $x = x + y * z$ and implicit statements like $x++$. Next, LOV analysis tests the basic LOV candidacy of these expression statements: does the expression have uses of the same variable, explicitly or implicitly, that it defines (e.g., the same variable appears on both the right-hand and left-hand sides of an explicit assignment expression).

The next phase of LOV analysis considers a statement only when its expression can be reduced, using the def-use chain, to a form of basic monotonic statements: $x \leftarrow x + c$, $x \leftarrow x - c$, $x \leftarrow x * c$ and $x \leftarrow x/c$ where $c$ is a positive compile-time integer literal or a positive loop invariant variable (when $c$ is negative, LOV analysis simply exchanges the rule between $x \leftarrow x - c$ and $x \leftarrow x + c$ while $x \leftarrow x * c$ and $x \leftarrow x/c$ are classified as non-monotonic).

We do not consider other more complex statements such as the dependent monotonic statement [40] where its defining variable inherits the loop monotonicity from other monotonic variables. We adopt this simplification because extracting one variable suffices for loop ordering, unlike other applications of monotonic variables, and thus the more complex monotonic variables are unnecessary.

As part of the expression reduction process, LOV analysis tests variable usage for ambiguity based on definition 2.3 and definition 2.4. If the $c$ term in a monotonic statement is ambiguous, LOV analysis assigns the chaos state ($\perp$) to its defining $x$. Similarly, LOV analysis assigns $\perp$ to its defining $x$ if its use is ambiguous.

Figure 5 illustrates the state changes of a loop order variable candidate. Each node represents a candidate loop order variable state and each edge represents a possible state transition that its labeled conditions trigger. The subscript of $x$ in a node encodes the initial value of $x$ on entry to the loop: $+$ means the initial value ($x_0$) is greater than or equal to zero and $-$ indicates it is less than zero. The superscript of $x$ in a node represents the loop monotonicity direction: $\uparrow$ and $\downarrow$ indicate monotonically increasing and monotonically decreasing respectively. Similarly, the subscript of the loop invariant $c$ denotes the sign of its value. As LOV analysis iterates over the assignment statements, it determines the state of the variable defined by that statement based on its previously classified state and the current classification. For example, the state of a candidate variable becomes $\perp$, regardless of its previous state, if LOV analysis classifies the current statement as $\perp$. Similarly, if the previous state is $x_+^{\uparrow}$ and the current statement is of the form $x \leftarrow x * c$, the state remains $x_+^{\uparrow}$. Any variable that is not $\perp$ or $\top$ (the initial state) when LOV analysis completes is a true LOV.

Our algorithm identifies `it` in figure 3 as a LOV with the $x_+^{\uparrow}$ attribute with respect to the loop that spans (4) and (21). The loop has only one corresponding monotonic statement, `it++` at (20), and it has an unambiguous, implicit use of `it` with an unambiguous initial value with the $+$ attribute. We can reduce this

expression to `it ← it + 1` that triggers the state transition from $\top$ to $x_{+}^{\uparrow}$. Thus, this variable satisfies all LOV properties. So, we can use its runtime value instead of the placeholder established in the first analysis.

### 2.2.3 Program Point Selection Strategy

Our combined static and dynamic analysis method to determine the relative progress of tasks could incur significant overhead. Transforming all stack trace frames into the lexicographical representation on compute nodes would allow a trace merging engine like STAT to resolve the lexicographic order at all branching points of the resulting prefix tree. However, dramatically increased file system access, data storage and transfer requirements at the fringes of an analysis tree [27] would quickly eclipse these benefits for large scale runs that use up to hundreds of thousands of MPI tasks. Thus, we designed an adaptive prefix tree refinement method that addresses these scalability challenges.

Our method begins with STAT's basic stack trace prefix tree and allows a user to refine the tree adaptively from the root to the leaves of the tree. Thus, the user can selectively focus on parts of the tree likely to exhibit errors. A simple menu action invokes the first step in which we analyze all frames leading up to the children of the first branching point, which transforms an unordered tree into an ordered one up to the branching point. We gather runtime information for a frame through a scalable communication infrastructure only when the lexicographical representation for the frame contains one or more LOV tokens. Otherwise, a single static analysis can evaluate a frame for all tasks. We stop the refinement if the runtime LOV resolution creates a new branching point; otherwise we continue up to the children of the branching point.

Our heuristic classifies the tasks into a set of temporal equivalence classes, thus presenting a user with a limited set of high-level choices in which to explore relative progress further. The user selects a class for further refinement and then a menu action invokes our method for the sub-prefix tree determined by that temporal equivalence class. Thus, we exploit the transitivity of the partial order: all frames in the sub-prefix tree maintain the same order, in relation to tasks of the other classes, as determined previously. In a sense, each branching point in the prefix tree is an idiom analogous to an individual frame of a singleton stack trace. As each frame allows a user to explore the temporal direction of the sequential execution space, each branching point in our relative progress tree allows the user to explore the temporal direction of the distributed execution space. Thus, our methodology transforms an execution chronology unaware prefix tree into a chronology aware one.

### 2.2.4 Demonstrating the Utility of Our Temporal Ordering

Our experiments demonstrate the utility of our temporal order analysis and that our prototype extension to STAT achieves scalable performance that more than supports interactive debugging. As discussed previously, we build upon ROSE's AST manipulation capabilities and def-use analysis [11, 37] to implement the analyses described in section 2.2.2. We have performed fault injection experiments that demonstrate the effectiveness of our techniques over a wide range of applications and faults. Our performance results demonstrate that the technique achieves sufficient speed to support interactive debugging. We also performed a case study that applyied the mto Algebraic MultiGrid (AMG) 200, a scalable iterative solver and preconditioner that is widely used in Office of Science applications. In preparation for extending it to unprecedented numbers of multicore compute nodes, the AMG team was testing performance-enhancing code modifications at increasingly large scales when a hang occurred at 4,096 tasks.

In order to diagnose this issue, we first examined the first level of detail with STAT, which merged stack traces based on the function names, which indicated that the hang occurred in the preconditioner setup phase during creation of the mesh hierarchy. However, no equivalence class was clearly the cause of the hang. Thus, we examined the line number-based, chronology-unaware tree and began the adaptive refinement process. As

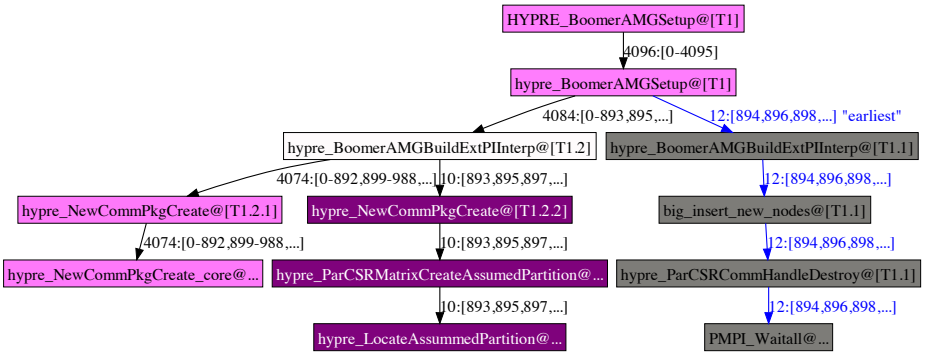


Figure 6: Chronology-aware prefix tree for a code hang exhibited by AMG2006 at 4,096 MPI tasks

figure 6 shows, this refinement quickly identified a group of twelve tasks that had ceased progressing due to a type coercion error at a function parameter in the `big_insert_new_nodes` function.

At the first refinement step, our method evaluated all frames leading up to the `hypre_BoomerAMGSetup` function and found that they were temporally equal. During this evaluation, we found one active loop, the `while` loop that tests the completion of the coarsening process within `hypre_BoomerAMGSetup`. LOV analysis identified the `level` variable, which keeps track of multigrid levels, as an LOV with the $x^{\uparrow}_{+}$ attribute. We found that its values were four in all 4,096 tasks.

The next refinement determined that a group of twelve tasks had made the least progress as indicated by edges in blue in figure 6. Because the next refinement step for these twelve tasks found all frames preceding the next branching point to be equal and the branching point was in the MPI layer, we manually inspected the code for relevant execution flows in and around the `hypre_BoomerAMGBuildExtPIInterp` $\rightarrow$ `big_insert_new_nodes` $\rightarrow$ `hypre_ParCSRCommHandleDestroy` path. We quickly found a type coercion problem for `int offset`, a function parameter of `big_insert_new_nodes`. The application team had recently widened key integer variables to 64 bit to support matrix indices that grow with scale. However, they overlooked the definition of this function, causing the type coercion. We theorize that at this particular scale and input, the 64-bit integers were truncated when coerced into 32-bits during parameter passing for the twelve tasks, which in turn caused the tasks to send corrupted MPI messages. Ultimately, this incorrect communication caused these tasks to hang in the `MPI_Waitall` call.

# 3 CBI: Cooperative Bug Isolation at Scale

Cooperative Bug Isolation (*CBI*) is a body of techniques and tools for diagnosing bugs in widely-deployed software systems. Prior to CoPS, CBI work focused on mainstream desktop software with large user communities. As its name suggests, CBI aggregates information from large numbers of runs in order to fix problems affecting many users. A cooperative approach also allows load sharing: each individual run incurs only a small instrumentation overhead to make its contribution to the much larger diagnostic picture. Yet from the developer's perspective, aggregation provides a critical window into the software's behavior (and misbehavior) to help to identify and to fix critical bugs rapidly. Under CoPS, we have extended CBI to consider the processes in a MPI application run as individual runs, which allows a single user to accumulate the data necessary to provide CBI's statistical methodology.

Debugging using CBI has two key phases: instrumentation and analysis. During the *instrumentation phase*, extra monitoring code is injected into an application before deployment. This monitoring code tracks run-time events of potential interest for debugging, such as branch directions, function return values

Table 2: Example CBI failure predictor list

| Thermometer | Predicate | Line |
|---|---|---|
|  | `files[filesindex].language > 16` | 5869 |
|  | `((*(fi + i)))->this.last_line == 1` | 5442 |
|  | `token_index > 500` | 4325 |
|  | `(p + passage_index)->last_token <= filesbase` | 5289 |
|  | `__result == 0 is TRUE` | 5789 |
|  | `config.match_comment is TRUE` | 1994 |
|  | `i == yy_last_accepting_state` | 5300 |
|  | `new value of f < old value of f` | 4497 |
|  | `files[fileid].size < token_index` | 4850 |
|  | `passage_index == 293` | 5313 |
|  | `((*(fi + i)))->other.last_line == yyleng` | 5444 |
|  | `min_index == 64` | 5302 |
|  | `((*(fi + i)))->this.last_line == yy_start` | 5442 |
|  | `(passages + i)->fileid == 52` | 4576 |
|  | `passage_index == 25` | 5313 |
|  | `strcmp > 0` | 4389 |
|  | `i > 500` | 4865 |
|  | `token_sequence[token_index].val >= 100` | 4322 |
|  | `i == 50` | 5252 |
|  | `passage_index == 19` | 5313 |
|  | `bytes <= filesbase` | 4481 |

and unusual floating-point numbers. CBI instrumentation is lightweight, using sparse random sampling to limit both overhead and feedback data size. During CBI's *analysis phase*, feedback data from deployed, instrumented applications is collected and mined for clues as to the root causes of failure. Each run is labeled as "successful" or "failed" in some generic or application-defined way. We then build statistical models that reveal program (mis)behaviors that are strongly predictive of failure, which is essentially a very high-dimensional supervised-learning problem, with monitored program behavior as inputs and subsequent success/failure as the output.

For example, CBI's analyses might determine that if `x == 0` and a specific branch goes left, then the program under analysis is much more likely to crash. We can present this information directly to a developer, or use it to direct the attention of heavyweight analyses that would have been impractical without such a focus. CBI data analysis combines ideas from both program analysis and machine learning; we use the term *statistical debugging* to describe CBI's use of statistical models to drive bug diagnosis and repair.

Table 2 shows an example of statistical debugging output. Each row presents one failure predictor, believed by CBI to represent one underlying bug. The first column uses a graphical representation, the *bug thermometer*, that we have developed to display summary data about program misbehavior. Failure predictors are ranked by severity and diagnosis confidence. Programmers should focus on wide thermometers that are mostly filled with red: these represent frequently recurring problems for which we have a very high-confidence explanation. The live, running system offers further interactivity not possible on paper, such as clickable links to source code and extensive drill-down detail for each failure predictor.

CBI's blend of static, dynamic and statistical methods has led to significant, transferable advances in all of these areas. Prior CBI research describes methods for lightweight, statistically fair instrumentation sampling

[30], shows that sparse data can be aggregated to isolate previously-unreported bugs [45], and develops a practical infrastructure for deploying and managing instrumented applications in real user communities [31]. Recent work focuses on CBI's second stage: data modeling to drive debugging. Various statistical debugging models have been developed, both by co-PI Liblit [29, 32, 33, 45, 46] and by others building on his work [14, 21, 22, 41, 44]. Some of these models constitute advances in pure machine learning [4], while others represent novel hybrids between static and dynamic/statistical sources of information. As an example of the later, work with Lal et al. [23] uses both statistical failure models and a static interprocedural dataflow analysis (based on weighted pushdown systems [38]) to identify failure-inducing program paths.

CBI has successfully isolated both fatal and non-fatal bugs relating to such varied problems as input validation, bad comment handling, unchecked return values, inconsistent data structure coordination, buffer overruns (both with and without memory writes), configuration-sensitive hash table mismanagement, memory exhaustion, premature returns, poor error-path handling, race conditions and dangling pointers [6, 23, 29, 30, 33]. CBI's statistical approach is robust in the face of uncertain, unrepeatable, and incomplete data, remaining effective at sampling rates of one event per hundred or even one per thousand [31].

Scientific computing represents a tantalizing new arena in which to apply statistical debugging techniques, but carries some unique challenges. Unlike desktop software, scientific applications are rarely, if ever, finished. Any given version of a scientific application is run very few times by very few users, rather than many times by many users as for widely-deployed desktop applications. Fortunately, scientific applications are often run at a large scale, with many individual processes participating. If we treat each process as a program run in the traditional sense, we can gather large quantities of feedback data for analysis in the few runs available. Unfortunately, because all of these processes are communicating, they are no longer independent; this violates one of the key assumptions underlying most of the statistical models used in feedback analysis.

Performance represents another challenge. The instrumentation used to observe program behavior at run time is generally lightweight and works very well for desktop applications. However, this instrumentation fares worse in the presence of the tight loops that are common to computationally-intensive code. However, if this problem can be surmounted, then there is good reason to expect that statistical debugging itself should scale well, as it requires no extra communication between compute processes.

## 3.1 Sampling Scientific Workloads

CBI's original instrumentation sampling procedure works well for typical interactive desktop applications, which spend most of their time waiting for user input. However, scientific workloads are CPU-bound and spend most of their time in loops performing numeric computations. The logic to choose between the fast or instrumented path is executed once per acyclic path, and therefore, once per loop iteration. This imposes a significant overhead, especially for loops with small bodies. We present an optimization to the sampling transformation to reduce sampling overhead substantially for most numeric loops.

### 3.1.1 Sampling Optimizations for Loops

Listing 1 shows a normal loop after the sampling transformation. Note the path check on line 2 that is based on the `countdown` and the two copies of the loop body: one instrumented and the other with only countdown decrements (the fast path). The `WEIGHT` constant referenced in the path check is, again, the maximum number of instrumentation sites in any path through one copy of this loop's body.

For small loops, the weight of the loop body is much less than the countdown and a significant fraction of these checks are wasted and result in the fast path being chosen. Our optimization amortizes the cost of the path check over as many iterations as possible. If the loop meets certain conditions, discussed in section 3.1.2, then we can precisely bound the number of loop iterations that can execute before we require another countdown check.

Listing 1: A simple sampled loop

```
1  for (i = 0; i < vec_len; i += STRIDE) {
2    if (countdown > WEIGHT) {
3      // Fast path
4    } else {
5      // Instrumented path
6    }
7  }
```

Listing 2: Optimized variant of listing 1

```
1  i = 0;
2  while (i < vec_len) {
3    int loop_start = i;
4    int bound = vec_len;
5    if (countdown <= (bound - i) / STRIDE * WEIGHT)
6      bound = i + (countdown - 1) / STRIDE * WEIGHT;
7    for (; i < bound; i += STRIDE) {
8      // Completely uninstrumented path
9    }
10   countdown -= (i - loop_start) / STRIDE * WEIGHT;
11   if (i < vec_len) {
12     // Instrumented path
13   }
14   i += STRIDE;
15 }
```

We rewrite the loop in three parts. First, it executes without any instrumentation, not even countdown decrements, up to the computed bound. Next, since the optimized loop body has no countdown decrements, we must update the countdown to reflect the number of instrumentation sites that were executed. Finally, execution enters a fully instrumented version of the loop body where we know a sample will be taken. We wrap these two steps inside of a driving loop to repeat the process as many times as is necessary to reach the total required number of iterations.

As a demonstration, the optimized form of the previous code example can be seen in listing 2. The constant `STRIDE` is the amount by which the induction variable changes each iteration. Line 2 shows the driving loop that ensures we execute the appropriate number of loop iterations. The bound is computed in lines 4 to 6, and determines the number of consecutive uninstrumented loop body iterations in line 7. The countdown is updated in line 10 to reflect the number of executed instrumentation sites. The additional check in line 11 ensures that we do not execute the loop body an extra time if the fully uninstrumented path included the last iteration of the loop.

The optimization generalizes and nested loops are fully supported, assuming each loop meets the transformation requirements. In practice we rarely find that the optimization can be applied to loops nested more than three deep. More complicated nested looping constructs are rare and typically contain other violations of the conditions in section 3.1.2. Further, the performance benefits of the optimization are typically maximized by doubly-nested loops due to the complexity of the loop bound calculation for more deeply-nested structures and limits imposed by the sampling rate.

### 3.1.2 Conditions on the Loop Body

Loop bodies must satisfy two high-level requirements in order to qualify for the loop optimization: the weight of the loop body must be finite and the number of iterations must be symbolically expressible. We further require that all paths through the loop body have the same weight and have no control-flow–altering constructs such as **`break`**. In principle, this restriction is not necessary; we can insert dummy instrumentation sites to balance out all of the paths through a loop. In practice, this type of loop typically fails to meet the finite-weight requirement and does not derive any benefit from the path balancing. The dummy instrumentation sites are undesirable, particularly in loop bodies, because they consume randomness without the chance to provide useful feedback data. Since they rarely offer performance benefits, we do not use them.

In order for the number of loop iterations to be symbolically expressible, the following conditions must hold:

- the loop body must not modify the induction variable,
- the loop body must not modify the bound on the iteration count,
- the stride must be constant,
- the loop condition must be idempotent, and
- the induction variable must be local.

The idempotence condition is required because the transformation duplicates the evaluation of the upper bound of the loop. When optimizing nested loops, the loop upper bounds and initial induction variable values must have no data dependencies on variables defined in enclosing loops.

### 3.1.3 Non-uniform Sampling Rates

The preceding optimization effectively amortizes path checks in numeric loops over many iterations. Unfortunately, realistic sampling rates tend to be about 1/100, limiting the scope of the amortization. Loops that are subject to this transformation are, by definition, computational *leaves* and are not permitted to call side-effecting functions. The instrumentation sites in these loops are typically floating-point operations and are less interesting from a debugging perspective than other operations. Moreover, they occur frequently, with event counts reaching the hundreds of millions.

We leverage the nature of these loops by dynamically reducing the sampling rate for their duration [18]. This, in turn, magnifies the amortization benefit of the loop-splitting optimization. Each optimized loop runs in a learning mode in which we discover how many iterations it executes during its first execution. Each time the loop completes a set of uninstrumented iterations, we reduce the sampling rate by a factor of 10, with a minimum sample rate determined by an environment variable. This mechanism exponentially decays the number of samples taken in each loop based on the size of its inputs. The new sampling rate for the loop is memoized and re-used in future executions of the loop.

### 3.1.4 Revisiting Numeric Loops

The loop-splitting transformation identifies a class of numerically- and computationally-intensive loops as optimization targets. The information that we obtain by instrumenting these loops has little diagnostic value for many classes of bugs. Consider a vector normalization function. The loop termination condition contributes two predicates since it can be observed to be either true or false on each iteration. However, these predicates are often redundant; assuming the loop executes at least once, predicates preceding the loop imply that the loop condition could be observed to be true. Likewise, predicates after the loop imply that the loop

condition could be observed to be false. If the loop never terminates, no predicates after the loop will be observed, yielding approximately equivalent results. When diagnosing these types of bugs, omitting the instrumentation from these loops entirely is an option.

## 3.2 Data Collection

Some prior work [31] relies on the instrumented application to report its own feedback data by writing to a file. This approach can be a barrier to scalability due to I/O pressure, and is not even possible in some computing environments (e.g. BlueGene/L). Additionally, reporting feedback data from a failing process requires handling POSIX signals to catch events like segmentation faults. Performing complex tasks in signal handlers is unwise at the best of times; when the process is failing and in an unsteady state, it is even more questionable. To address these problems, we (1) move the reporting infrastructure from the instrumented process to an external *watchdog*, and (2) propagate feedback data to a reporting node using MRNet [39].

### 3.2.1 Reporting Machinery

The watchdog process uses the Dyninst framework [7] to monitor instrumented processes for termination, abnormal or otherwise. To communicate feedback data efficiently from the instrumented process to the watchdog, we employ a shared memory segment visible to both processes. The instrumented process stores its feedback data within the shared memory segment. When the instrumented process terminates, the watchdog simply reads the feedback data out of the shared memory segment. In the event of an abnormal termination, the watchdog also captures a stack trace.

Besides efficiency, the shared memory segment offers significant robustness advantages over an in-process reporting approach. With in-process reporting, heap corruption could easily render the instrumented application incapable of producing a report at all, or worse, could cause it to produce a seemingly-valid report with hidden corruption. Standard library components such as I/O buffers or file descriptors are likewise vulnerable. By contrast, when using a shared memory segment, only the small area occupied by that segment is exposed to possible corruption. Relative to the entire address space, this is a much smaller surface of vulnerability. Even in the face of extreme termination measures such as `SIGKILL`, a shared memory segment still allows feedback to be captured, whereas in-process reporting does not.

After the watchdog collects all of the available feedback reports, it sends the data to a reporting node via MRNet, a scalable Multicast/Reduction Network. MRNet provides a tree-structured communication network in which our watchdog processes form leaves, or *backends* in MRNet parlance. Data is propagated up the tree to a *frontend* node, which we use as a reporting node to write the feedback data to disk.

Once feedback data enters the MRNet communication tree, it passes through *filter functions* at each level of the tree until reaching the frontend node. These filter functions allow arbitrary transformations of the data as it propagates; we use them to losslessly compress samples. Each watchdog process sends its feedback data into the communication tree uncompressed and the first layer compresses it with a standard compression algorithm. Further levels in the communication tree concatenate the data they receive. This approach allows the compression algorithm to exploit a large window of redundancy across more feedback reports than are available to a single watchdog.

### 3.2.2 Data Format

Feedback reports are ordered tuples of integers. Each tuple represents a single instrumentation site, while each entry in the tuple denotes the number of times that individual predicate was observed to be true at that instrumentation site. Many predicates are never observed to be true, or are observed to be true only a few times. On the other hand, predicates in nested loops can easily be observed hundreds of millions of times.

This range of data benefits from the implicitly variable-length encoding afforded by plain text; however, textual formats are wasteful in their use of delimiters and representation of very large numbers. An alternative is to use the standard Abstract Syntax Notation (ASN.1), which is a binary encoding with variable-length integer representations.

Furthermore, many instrumentation sites are never reached in a given run of a program. We use a *sparse* representation to reduce the space overhead of unreached code. Each feedback report has an associated bitmask and tuples containing all zeros are represented by a zero in the bitmask. All tuples containing data are represented by a one, and the full sequence of tuples is reconstructed at analysis-time. The sparse ASN.1 format has shown space savings of 35% to nearly 700% in report encodings.

We simulated data collections through feedback reports generated by IRS. The reports are encoded using the sparse ASN.1 representation discussed above and compressed using bzip2. Additionally, the reports are randomly perturbed to prevent the compression algorithm from achieving trivial best-case behavior. This simulation indicates that data format and communication infrastructure can collect reports from 500,000 processes in less than 50MB.

### 3.3 Implementation

We have implemented a new instrumenting compiler as a source-to-source translator using the ROSE compiler infrastructure [12]. To coordinate our watchdog processes with the applications that we debug, we start them simultaneously using LaunchMON [2]. These watchdogs monitor applications via the Dyninst StackwalkerAPI [7] and report results using MRNet [39], a scalable communication medium. We leverage the OpenMP, C++, and Fortran support in ROSE to handle a larger selection of applications than previous source-based instrumenting compilers. This support is particularly important for scientific applications, many of which use some features from (or components written in) these languages.

Three C++ language features complicate our instrumentation mechanism: (1) reference types, (2) objects with user-defined constructors or destructors (known as non-Plain Old Data, or *non-POD*, types), and (3) **`try`** blocks. These features are all troublesome for the same underlying reason: they inhibit jumping between fast and instrumented paths. We divide each function into two paths: the fast and instrumented. At the beginning of each acyclic region, execution can either stay on its current path or jump to the other, depending on the value of the countdown. After this jump, the same variables must all be in scope with the same values. This is easily facilitated in C by lifting all variable declarations, with proper renaming, to the top of a function body. In fact, we instrument C++ code that does not use any of the aforementioned three features in exactly this way.

Reference types are complicated because they must be initialized and cannot be made to refer to another object after initialization. If the declaration of the reference is lifted to the top of the function, its initializer might not yet be in scope. We handle this case by rewriting reference-typed variables as pointer-typed variables and making previously-implicit dereferences explicit.

Non-POD declarations also pose a scoping problem: if we move or duplicate these declarations then we also move or duplicate the side effects of their constructors or destructors. Additionally, execution cannot jump past the declaration of non-POD objects. We do not move non-POD declarations; instead we recursively treat the code that they dominate as a new function for the purposes of creating fast and instrumented paths. Effectively, we split the code around them. In principle, we could do this for all variable declarations. However, this technique negative impacts performance by making fast code paths shorter. Therefore, we apply this transformation only for non-POD variables. We continue to lift POD variables up to the top-level scope in each function.

C++ **`try`** blocks introduce a similar difficulty to non-POD declarations: execution cannot jump from the middle of one **`try`** block into the middle of another. Thus, we treat **`try`** blocks similarly to non-POD variable declarations: we never clone them, but instead recursively treat **`try`** block bodies as though they

Table 3: Failure predictors for ParaDiS

| Predicate | Function |
|---|---|
| `i < home->newNodeKeyPtr` | `SortNativeNodes` |
| `inode < home->newNodeKeyPtr` | `MonopoleCellCharge` |
| `tag.domID == home->myDom` | `GetNodeFromTag` |
| `cycleEnd == 0` | `DD3dStep` |
| `iNbr > nXcells` | `InitCellNeighbors` |
| `remDom == 0` | `GetNodeFromTag` |
| `node != 0` | `CommPackGhosts` |

were the entry points of new functions for purposes of fast versus instrumented path creation.

### 3.4 Evaluation of CBI for MPI Applications

We have applied our techniques to ParaDiS [8], a dislocation dynamics simulator. Version 2.0 of this code suffers from a bug that causes it to crash on most of its inputs. We instrumented the code to sample predicates on branches taken and function return values. After finding a working input, we applied our analysis to several crashing runs and a few runs on the successful input; the analysis identified the predicates in table 3 as significant failure predictors.

This code divides the problem space, and hence dislocations, into *domains* that are distributed among compute nodes. ParaDiS refers to dislocations as "nodes" internally, particularly in function names; except for direct references to ParaDiS functions with "node" in the name, we use the term to refer only to compute nodes in a cluster. Each domain is divided into cells and is responsible for a set of *native* dislocations; non-native dislocations are represented as *ghosts*. At each step in the computation, each compute node

1. *migrates* ownership of dislocations that cross the boundary of its domain to appropriate neighbors,
2. organizes its remaining native dislocations into cells,
3. sends updates of ghost information to neighboring nodes, and
4. computes the local effects of forces on dislocations.

The failure manifests as a segmentation fault in the `OrderNodes` function. This function is called by `MonopoleCellCharge` in the loop controlled by the predicate identified by our analysis. We can explain the bug by working backwards in the call graph from this point of failure. The nearest failure predictor is `remDom == 0` evaluating to true in `GetNodeFromTag`. This returns a `NULL` pointer, which eventually causes the segmentation fault.

Temporally, the next preceding predictors are in `CommPackGhosts` and `SortNativeNodes`, which update ghost dislocations in neighboring domains and divide local dislocations into cells, respectively. Both of these predictors arise because they are executed more frequently in failing runs, thus appearing in fewer successful runs. This suggests a correlation between failure and runs with many dislocations.

Temporally, the next nearest predictor is in `InitCellNeighbors`. This function is called before the first time step. Note that each step of the computation begins by migrating some dislocations to neighboring nodes. From here, we hypothesize that the crash arises because dislocation ownership is not tracked correctly. This leaves nodes with an inconsistent view of the dislocations owned by their neighbors after the first migration. `MonopoleCellCharge` causes the crash when it attempts to inspect non-existent dislocations on a neighbor node. The cases in which the application succeeds are those with few dislocations which happen to not fall

near cell boundaries. Inspection of the next ParaDiS release (2.2) shows that the code tracking dislocations eligible for migration and ghosts has been completely rewritten, suggesting that this was indeed a significant contributor to the underlying bug.

The predictor trace offers significant detail not available from backtraces. Backtraces can only show the state of the stack when a problem occurs; predictor traces can link together events from different branches of the run-time call graph. In this case, the backtrace would not include `GetNodeFromTag`, `InitCell-Neighbors`, `SortNativeNodes`, or `CommPackGhosts`, which are essential to our understanding of the actual root cause.

Our evaluations of our implementation of CBI for MPI applications demonstrated that we achieve reasonable overhead. We examined the overhead imposed by our instrumentation on AMG, IRS and ParaDiS against baseline versions compiled with GCC. Observed overheads are between 10% and 15% for IRS and ParaDiS, closely tracking the serial overheads reported previously. AMG also falls largely within this range, but with spikes up to 25% overhead at some problem sizes that have short running times, which provide little opportunity for instrumentation costs to be amortized.

# 4 MPIEcho: Parallelizing Dynamic Correctness Checking

Dynamic correctness checking or semantic debugging supports checking that application exectuion conforms to rules of correct usage of program constructs such as dynamic memory allocation. This important class of correctness tools works well on short, single node executions. Simple techniques have extended the basic tools to multinode systems but these techniques do not support application of the tools to large-scale executions, let alone millions of cores. Heavyweight debugging tools such as Valgrind [36] and Parallel Inspector [20] are indispensable when solving smaller problems, but their overhead precludes their use at scale except as a last resort: memory checking can reach 160x slowdown and thread checking can reach 1000x. In order to allow development of more scalable semantic debugging, we have developed *MPIecho*, a novel runtime platform that enables cloning of MPI ranks. Given identical execution on each clone, we can parallelize the heavyweight debugging approaches to reduce their overhead to a fraction of the serialized case. We have shown that this platform can be useful in isolating the source of hardware-based nondeterministic behavior and provide a case study based on a recent processor bug at LLNL.

In our work, we parallelize heavyweight tools in order to reduce the overhead incurred by existing tools and to allow the development of novel approaches. Per-instruction instrumentation such as that used by the Maid memory access checking tool can be rendered effectively embarrassingly parallel. The more interesting cases, such as parallelizing read-before-write detection, can still show substantial reduction in runtime overhead by duplicating write instrumentation and parallelizing read instrumentation. We have also shown this platform is flexible enough to be used in hardware debugging and performance analysis. By assuming cloned ranks should exhibit identical execution we can perform fast *hardware fault detection* by observing when this assumption is violated and correlating the fault to a particular node. We examined a case study of a recent processor bug at LLNL that has informed the design of *MPIecho*.

While total overhead will depend on the individual tool, we have shown that the platform itself contributes very little: 512x tool parallelization incurs at worst 2x overhead across the NAS Parallel benchmarks, hardware fault isolation contributes at worst an additional 44% overhead. Finally, we show how *MPIecho* can lead to near-linear reduction in overhead when combined with Maid, a heavyweight memory tracking tool provided with Intel's Pin platform. We demonstrate overhead reduction from 1,466% to 53% and from 740% to 14% for cg.D.64 and lu.D.64, respectively, using only an additional 64 cores.

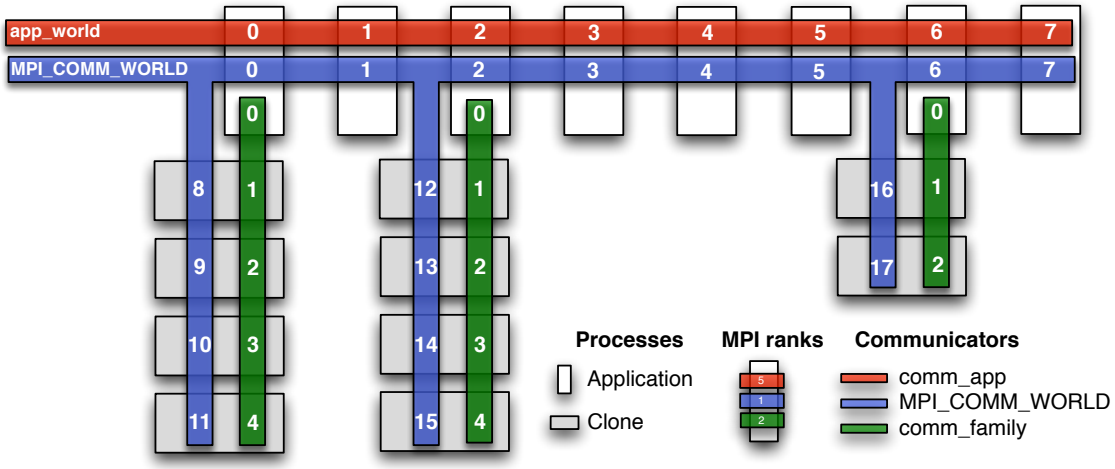


Figure 7: Architecture of *MPIecho*

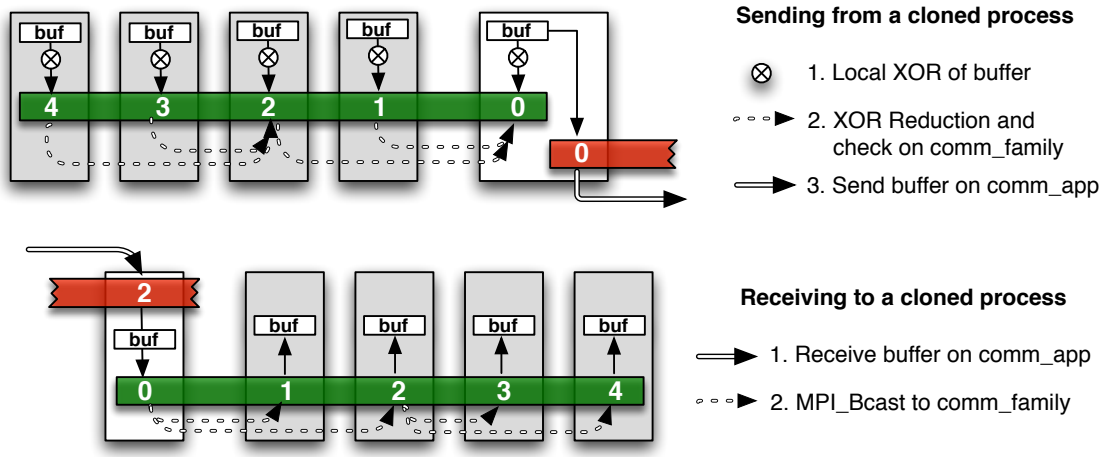


Figure 8: Send and Recv under *MPIecho*

## 4.1 *MPIecho* Overview and Implementation

The goal of the *MPIecho* platform is to provide duplicate execution of arbitrary MPI ranks. Overhead should be kept to a minimum and the behavior of the clones should not perturb the correctness of execution. In this section we describe the architecture of the platform as well as the experimental measurement of the tool's overhead.

At at high level, the design is simple. At startup, a cloned MPI process $r$ (the *parent*) will distribute all messages it receives to one or more clones $c$. Messages sent from the clones are routed to the parent if necessary but are not otherwise distributed to the rest of the system (see Figure 7 and Figure 8). As long as the state of a process depends only on the initial program state and inputs received via the MPI API, this general approach will guarantee identical execution on the clones. The overhead of this approach is dominated by the additional messages sent to the clones. If the cloned process is on the *critical path*, any additional overhead will accrue in overall execution time.

At a lower level, the function calls provided by the MPI API have IN, OUT, and INOUT parameters.

When any function call completes, the OUT and INOUT parameters should be identical across the parent and clones. For example, an `MPI_Recv` call has a buffer updated by the incoming message. This condition also applies to the parts of the API that are not involved with communication, for example querying the size of a communicator or constructing a derived datatype. A naive implementation could simply copy every OUT and INOUT parameter to the clones. This approach incurs unnecessary overhead and relies on non-portable knowledge of how opaque handles are implemented. Instead, we have minimized communication between the parent and clones using the following techniques:

1. *Broadcast.* The parent communicates with the clones via `MPI_Bcast` using a communicator dedicated to that purpose. In MPI implementations we are familiar with, this implies the parent sends out only $\log_2(|c|)$ messages per communication (where $|c|$ is the number of clones). We have found this sufficiently fast for our needs, but scaling this approach to thousands of clones may require the parent sending a single message to a single lead clone with the lead clone then (asynchronously)broadcasting to the remaining clones.

2. *Opaque handles.* MPI relies on several different opaque handles that cannot be queried except through the API. Copying these requires knowledge of their internal structure; this tends to be non-portable. Instead, we only communicate on a "need-to-know" basis. For example, a call to `MPI_Comm_split` will not be executed by the clones. Instead, the parent will send the associated index value of the new communicator to the clones, along with values for size and rank. Clones will not use this communicator for communicating and so no additional information is needed. Calls to `MPI_Comm_size` and `MPI_Comm_rank` can be resolved locally without any further communication with the parent, thus cutting out a potentially significant source of overhead.

3. *Translucent handles.* The `MPI_Status` datatype is only partially opaque: users may query several values directly without going through the API. Any call that potentially modifies a status results in copying just these values to the clones. For calls such as `MPI_Waitall` the visible status values are batched together and sent using a single broadcast.

4. *Vectors.* Several MPI calls such as `MPI_Alltoallv` support sending and receiving (possibly non-contiguous) vectors. Using derived datatypes, we construct a datatype that describes all updated buffers and uses this derived datatype to issue a single `MPI_Bcast` to the clones.

5. *Non-blocking communication.* Both the clone and parent record the count, type and buffer associated with each `MPI_Request`. In the case of the `MPI_Wait` family of calls both the parent and clone implicitly wait on an identical index and the broadcast occurs when the wait returns. In the case of `MPI_Test` the results are communicated in a separate broadcast, followed by the updated buffer if the test returned true.

6. *Barriers.* Some MPI calls, such as `MPI_Barrier`, have no OUT or INOUT parameters. The clone does not need to execute these calls at all, as no program state is changed. The clones resynchronize with the parent at the next communication call.

7. *Return values.* MPI calls return an error value, but there is no provision for recovery if the value is anything other than `MPI_SUCCESS`. We make the assumption that if an error condition occurs on either the parent or a clone, the only sensible choice is to indicate where the error occurred and halt. Future versions of MPI may make better use of the return codes; if so we will need to distribute them to the clones.

We use the PMPI profiling interface in our implementation. We intercept each MPI call and route it to our library. We use the `wrap` [16] PMPI header generator to create the necessary Fortran and C interface

| Bench-mark | Number of clones | | | | | | |
|---|---|---|---|---|---|---|---|
| | **8** | **16** | **32** | **64** | **128** | **256** | **512** |
| bt | 1.5 | -2.7 | -2.0 | 7.3 | 7.5 | 6.3 | 11.5 |
| cg | -2.0 | -0.3 | 2.6 | 3.6 | 9.0 | 8.5 | 15.7 |
| ft | 0.3 | 3.8 | 2.6 | 3.5 | 3.9 | 3.4 | 2.7 |
| is | 8.7 | 14.0 | 13.4 | 14.4 | 11.4 | 10.5 | 17.5 |
| lu | -1.4 | 4.4 | 2.2 | 2.1 | -1.5 | 1.5 | 9.2 |
| mg | 26.5 | 30.8 | 33.7 | 41.0 | 59.5 | 67.7 | 99.0 |
| sp | 2.3 | 0.0 | 3.6 | 1.4 | 6.5 | 10.2 | 15.5 |

Table 4: Percent overhead for cloning rank 0

code. These design choices allow us to use a single broadcast in the common case, and never more than two broadcasts per MPI call. Overhead is dominated by the size and number of messages. Effectively, the worst case cost is:

$$\begin{aligned}\text{Overhead} =& \text{Bcast}\left(\text{nranks} \times \text{typesize} \times \text{messagesize}\right)\\ &+ \text{Bcast}\left(\text{nranks} \times \text{statussize}\right)\end{aligned}$$

Because the clones do not execute any send functions (unless required by a tool implementation), they do not tend to remain on the critical path: the overhead should be limited to the direct cost of the barriers except in pathological cases.

## 4.2 Experimental Measurement of Overhead

We intend this platform to support parallel tools, but the time saved by committing more processors to the tools will eventually be offset by the additional time necessary to communicate with those processors. The overhead of the platform should contribute as little as possible to the overall overhead.

We executed the *MPIecho* experiments in this report on the Sierra cluster at Lawrence Livermore National Laboratory. We compiled the NAS Parallel Benchmark Suite [35], *MPIecho* and tools using GCC 4.1.2 fortran, c and c++ compilers and -O3 optimizations. We ran the experiments using MVAPICH2 version 1.5. All results are expressed in terms of percent time over the baseline case run without *MPIecho*. Process density has a significant affect on execution time: using 64 12-core nodes to run the baseline 64-process benchmarks can be usefully compared with 64 processes and $8 \times 64$ clones also running on 64 12-core nodes due to increased cache contention. We made a best-effort to keep process densities similar across all runs and increased node counts as needed.

In table 4 we show the percent measured overhead for several clone counts. Duplicating execution on node 0 scales to 512 clones with less that 18% execution time overhead for all benchmarks other than MG. In the case of MG, we observed an unusually high ratio of communication time to computation time which did not afford the opportunity to amortize the cost of the broadcasts to the same extent. However, even in this worst case we note that this approach still scales well: 512 clones of node 0 resulted in only doubling execution time. These results establish that the overhead incurred by the platform is low enough to be useful for parallelizing high-overhead tools.

## 4.3 Send Buffer Check

In this section we give a brief outline of the design, implementation and performance of *SendCheck*, a tool used to detect intermittent hardware faults. We illustrate how such a tool could have been useful in diagnosing

a recent CPU bug at Lawrence Livermore National Laboratory. With increasing processor and core counts, we expect similar tools to be increasingly important.

### 4.3.1 Problem Scenario

A user had reported seeing occasional strange results using the CAR [10] climate model. To the best of the user's knowledge the problem was isolated to a particular cluster and did not always manifest itself there. At this point, members of the Development Environment Group at LLNL were asked to assist with determining the source of the error. Since the problem appeared to be isolated to a particular cluster, the possibility of a hardware fault was raised early on in the process.

The first task was to determine if the problem was caused by a particular compute node. Node assignments on this particular cluster are nondeterministic and the user might only occasionally be assigned a particular bad node. After dozens of test runs the faulty node was eventually isolated. For a particular sequence of instructions a particular core failed to round the least significant bit reliably. Because the CAR climate model is highly iterative, repeated faults ultimately caused significant deviation from correct execution.

This bug was pathological in several ways: only a very specific instruction sequence triggered the fault, the fault is intermittent, and when the fault does occur it will only be noticeable in calculations that are sensitive to the values of the least significant bit. Indeed, one of the most curious manifestations of this fault was the observations that error could be introduced into partial results, disappear and the then reappear later. The design of the software allowed only partial results to be checked per timestep. These results did not provide sufficient granularity to isolate the fault to before or after a particular MPI message. For a complete description of the bug and the debugging process, see Lee [25].

### 4.3.2 *SendCheck* Design and implementation

We begin by noting that we are only interested in faults that affect the final state of the solution at the root node. The remaining nodes only affect the root node by sending messages to it. Rather than having to validate the entire machine state across multiple clones we have a far more tractable problem of validating only messages that would be sent by the clones.

The naive implementation is straightforward. At each MPI call where data is sent to another node, each clone copies its send buffer to the parent and the parent compares the buffers. If the buffers are not identical, the program is nondeterministic in a sense not accounted for by *MPIecho*. If all of the buffers are unique the cause of the nondeterminacy likely lies in the software. If only a single clone has a different buffer, the most likely explanation would be a fault isolated to that node. A small number of subsequent runs should distinguish between the two cases.

Copying potentially large messages is both expensive and unnecessary. Our implementation takes an md5 hash [13] of the message buffer and then executes a bitwise MPI XOR reduce across all clones of a given parent (see Figure 8). This optimization limits the amount of communication to a 16-byte hash per clone and allows the parent to perform an inexpensive check for differences. The saving in communication far outweighs the expense in creating the hash. This approach will not catch all errors: pathological conditions may be masked by the XOR. While unlikely, the user can fall back to the naive implementation if this msking is suspected.

### 4.3.3 *SendCheck* Experiments

For each of the benchmarks run we dynamically generated an md5 hash value for every send buffer before each MPI call executed. This was not restricted to the `MPI_Send` family, but included any call that transferred data to another MPI rank, including `MPI_Isend/MPI_Wait` and `MPI_Alltoallv`. The bitwise XOR of all hashes was send to the parent using an `MPI_Reduce`.

| Bench-mark | Number of clones | | | | | | |
|---|---|---|---|---|---|---|---|
| | **8** | **16** | **32** | **64** | **128** | **256** | **512** |
| bt | 2.1 | 6.7 | -0.2 | 0.0 | -0.8 | 2.5 | -0.3 |
| cg | 7.5 | 8.4 | 13.0 | 11.3 | 11.3 | 14.9 | 13.4 |
| ft | -0.5 | -1.8 | -0.1 | -0.3 | -0.7 | -0.7 | -0.4 |
| is | 46.8 | 46.9 | 55.0 | 48.1 | 50.8 | 50.8 | 42.4 |
| lu | 6.0 | 0.2 | 1.6 | 3.4 | 10.9 | 8.6 | -1.5 |
| mg | 1.2 | 2.8 | 4.1 | 5.1 | 4.5 | 1.2 | 3.7 |
| sp | -1.8 | 5.4 | -0.7 | 5.7 | 1.6 | -1.8 | -2.6 |

Table 5: Percent additional overhead for Sendcheck tool

| Bench-mark | Number of clones | | | | | | |
|---|---|---|---|---|---|---|---|
| | **8** | **16** | **32** | **64** | **128** | **256** | **512** |
| bt | 2.0 | 2.3 | -3.7 | 5.7 | 5.1 | 7.4 | 9.46 |
| cg | 11.7 | 14.5 | 22.9 | 22.3 | 28.5 | 32.1 | 39.16 |
| ft | 0.5 | 2.6 | 3.2 | 3.9 | 3.9 | 3.3 | 3.01 |
| is | 62.2 | 70.3 | 78.6 | 72.2 | 70.8 | 69.4 | 70.05 |
| lu | 2.8 | 2.9 | 2.1 | 3.8 | 7.5 | 8.4 | 5.76 |
| mg | 26.2 | 32.7 | 37.4 | 46.3 | 64.5 | 67.4 | 103.59 |
| sp | 0.3 | 5.2 | 2.7 | 7.0 | 7.9 | 8.0 | 12.08 |

Table 6: Percent total overhead for Sendcheck tool

Tables 5 and 6 show the additional and total overhead of the *SendCheck*. The overhead is marginal except in the case of IS. Here, the overhead introduced by the extra calls to md5sum could only be amortized over small amount of program execution. Validating messages up through 512 clones usually requires at most a 5% additional overhead with the worst case being less than 55%. In the worst case timing, numerous small messages prevented the usual amortization of the md5sum costs. We expect that we can lower this overhead significantly by selectively applying md5sum based on message size.

Had *MPIecho* been available when diagnosing the Opteron hardware fault we would have been able either to rule out hardware faults quickly (if the error appeared to be independent of node assignment) or to identify the hardware using a small number of runs. Thus, *SendCheck* would have saved months of debugging time. We plan to bring this tool into service debugging production code.

## 4.4 Maid

The Intel binary instrumentation platform Pin [34] provides several examples of tools that leverage its features. In this section we examine the Maid tool and explain not only why the serial case exhibits so much overhead but also how this kind of tool may be parallelized. We then present results using the combination of Pin, Maid, *MPIecho* and NAS Parallel Benchmark suite [35].

### 4.4.1 Overview of Pin and Maid

Pin can be thought of as a just-in-time compiler with two separate user interfaces: *instrumentation* and *analysis*. Instrumentation code determines where dynamic code will be inserted, and analysis code is what executes at the instrumentation points. Instrumentation code executes once when the binary is loaded into memory. Analysis code executes whenever the underlying code executes. The overhead due to instrumentation is paid only once and thus this code can be relatively complex. The overhead due to analysis will be paid every time the instruction is executed.

Maid identifies all instructions that access memory (both reads and writes) and inserts analysis code that checks if the effective address used by the instruction matches any of the addresses provided by the user. Similar functionality can be found in watchpoints in modern debuggers. However, a serial debugger must check every instruction to see if any memory references are of interest. A naive parallel implementation would divide memory regions among the clones and thus have each clone check only a small fraction of the potential address space. However, this implementation ends up being as slow as the serial case: every clone still checks every instruction.

Instead, given $n$ processes, each clone instruments every $n$th instruction, starting from the $i$th instruction (where $i$ is the process rank). This choice does not necessarily distribute the work evenly. In the pathological case that executes a single instruction in a tight loop, effectively nothing is parallelized. However, more realistic cases involve dozens to hundreds of instructions at the innermost loop level. Given sufficient clones each clone will instrument only a single instruction in the loop.

### 4.4.2 Maid Experiments

Only clones executed the dynamic binary instrumentation. We measure overhead against the no-clone, uninstrumented case: a single clone with all memory instructions are instrumented. For two clones, each clone instrumented half of the instructions, etc. The Maid tool is set up to check multiple memory locations with the overhead increasing as more locations are checked. For this set of experiments we check only a single memory location.

Table 7 lists our results. For the common case over a small number of clones we achieve near-linear scaling. In the most dramatic cases, bt went from an overhead of $1,466\%$ to only 53% and lu went from an

| Bench-mark | Number of clones | | | | | | |
|---|---|---|---|---|---|---|---|
| | **1** | **2** | **4** | **8** | **16** | **32** | **64** |
| bt | 1466 | 737 | 375 | 207 | 124 | 78 | 53 |
| cg | 317 | 138 | 131 | 144 | 187 | 185 | 194 |
| ft | 521 | 294 | 146 | 98 | 106 | 109 | 97 |
| is | 375 | 239 | 144 | 127 | 110 | 107 | 128 |
| lu | 740 | 369 | 183 | 100 | 58 | 24 | 14 |
| mg | 810 | 428 | 217 | 136 | 108 | 90 | 101 |
| sp | 376 | 180 | 84 | 37 | 23 | 14 | 12 |

Table 7: Percent total overhead for Maid tool

overhead of 740% to 14%, both using only 64 additional cores as clones. However, we note that in several cases the overhead due to additional nodes begins to dominate the savings gained by those nodes: in the cases of cg, is, and mg performance is worse at 64 clones than at 32 clones. However, the worst case, cg, still drops from 317% to 131% using only 4 clones.

### 4.5 *MPIecho* Discussion

Moving from a serial model of computation to a parallel model of computation not only led to existing problems being solved faster, it also allowed new kinds of problems to be solved. *MPIecho* allows a similar transformation to be brought to bear on debugging tools. Methods that may be left unexplored or unused to do prohibitive overheads may now be feasible, so long as they can be parallelized.

## 5 Conclusion

Given the expected processor counts in petascale systems, current strategies for debugging large-scale scientific applications will incur substantially increased costs. Since the costs of employing those strategies on existing systems are already far too high and represent a significant loss in application scientist productivity, new directions are essential. CoPS represents a paradigm shift for analyzing and understanding programming errors in large-scale scientific applications that will dramatically lower those costs. Overall, our results are creating new debugging paradigms that will provide debugging solutions for current and future large-scale systems. Our approach automates root cause analysis in many cases and substantially simplifies the use of traditional tools even when it fails to identify root causes automatically.

# 6 Literature Cited


[1] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming Scalability Challenges for Tool Daemon Launching. In *To appear in the Proceedings of the International Conference on Parallel Processing (ICPP)*, 2008.

[2] D. H. Ahn, D. C. Arnold, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Overcoming scalability challenges for tool daemon launching. In *ICPP*, pages 578–585. IEEE Computer Society, 2008.

[3] A. Aiken and D. Gay. Barrier inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–354, New York, NY, USA, 1998. ACM.

[4] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical Debugging Using Latent Topic Models. In J. N. Kok, J. Koronacki, R. L. de Mántaras, S. Matwin, D. Mladenic, and A. Skowron, editors, *ECML*, volume 4701 of *Lecture Notes in Computer Science*, pages 6–17. Springer, 2007.

[5] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *The International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007.

[6] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical Debugging Using Compound Boolean Predicates.

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

[8] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable line dynamics in ParaDiS. In *SC*, page 19. IEEE Computer Society, 2004.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[10] W. D. Collins and P. J. Rasch. Description of the NCAR community atmosphere model (CAM 3.0). Technical Report TN-464+STR, National Center for Atmospheric Research, 2004.

[11] K. Davis and D. Quinlan. ROSE: An optimizing code transformer for C++ object-oriented array class libraries. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98), at 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, volume 1543 of *Lecture Notes in Computer Science*. Springer Verlag, July 1998.

[12] K. Davis and D. J. Quinlan. ROSE: An optimizing transformation system for C++ array-class libraries. In S. Demeyer and J. Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pages 452–453. Springer, 1998.

[13] L. P. Deutsch. *Independent implementation of MD5 (RFC 1321)*. Aladdin Enterprises, 2002.

[14] L. C. Díaz. MUCODE: Código Crítico en Aplicaciones GNOME sobre Debian. Master's thesis, Universidad Rey Juan Carlos, Madrid, Spain, Sept. 2007.

[15] J. B. Drake, P. W. Jones, and J. George R. Carr. Overview of the Software Design of the Community Climate System Model. *Int. J. High Perform. Comput. Appl.*, 19(3):177–186, 2005.

[16] T. Gamblin. *The wrap MPI Wrapper Generator*. https://github.com/tgamblin/wrap, 2011.

[17] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.

[18] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In S. Mukherjee and K. S. McKinley, editors, *ASPLOS*, pages 156–164. ACM, 2004.

[19] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 74–81, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[20] Intel Corporation. *Parallel Inspector*. http://software.intel.com/en-us/articles/intel-parallel-inspector/, 2011.

[21] L. Jiang and Z. Su. Automatic Isolation of Cause-Effect Chains with Machine Learning. Technical report.

[22] L. Jiang and Z. Su. Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, Atlanta, Georgia, Nov. 2007.

[23] A. Lal, J. Lim, M. Polishchuk, and B. Liblit. Path Optimization in Programs and Its Application to Debugging. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2006.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[25] G. Lee. Comparative debugging: Identifying the source of rounding error on atlas587. Technical report, Lawrence Livermore National Laboratory, 2010. See also https://computing.llnl.gov/linux/corefputest.html.

[26] G. Lee, D. Ahn, D. Arnold, B. de Supinski, M. Legendre, B. Miller, M. Schulz, and B. Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *To appear in Supercomputing 2008*, 2008.

[27] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[28] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, B. P. Miller, and M. Schulz. Benchmarking the Stack Trace Analysis Tool for BlueGene/L. In *Parallel Computing: Architectures, Algorithms and Applications (Proceedings of the Internation Conference ParCo 2007)*, Julich/Aachen, Germany, 2007.

[29] B. Liblit. *Cooperative Bug Isolation: Winning Thesis of the 2005 ACM Doctoral Dissertation Competition*, volume 4440 of *Lecture Notes in Computer Science*. Springer, 2007.

[30] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Sampling User Executions for Bug Isolation. In A. Orso and A. Porter, editors, *RAMSS '03: 1st International Workshop on Remote Analysis and Measurement of Software Systems*, pages 5–8, Portland, Oregon, May 9 2003.

[31] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public Deployment of Cooperative Bug Isolation. In A. Orso and A. Porter, editors, *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 57–62, Edinburgh, Scotland, May 24 2004.

[32] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 15–26. ACM, 2005.

[33] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[34] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[35] NASA Advanced Supercomputing Division. *NAS Parallel Benchmark Suite*. `http://www.nas.nasa.gov/Resources/Software/npb.html`, 2006. Version 3.3.

[36] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dyanmic binary instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[37] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France*, volume 10 of *Parallel Processing Letters*. Springer Verlag, January 2000.

[38] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[39] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *SC*, page 21. ACM, 2003.

[40] M. Spezialetti and R. Gupta. Loop monotonic statements. *IEEE Trans. Softw. Eng.*, 21(6):497–505, 1995.

[41] H. M. G. H. Wassel. An Enhanced Bi-clustering Algorithm for Automatic Multiple Software Bug Isolation. Master's thesis, Alexandria University, Egypt, Sept. 2007.

[42] P. Wu, A. Cohen, J. Hoeflinger, and D. Padua. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 78–91, New York, NY, USA, 2001. ACM.

[43] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 194–204, New York, NY, USA, 2007. ACM.

[44] A. X. Zheng. *Statistical Software Debugging*. PhD thesis, University of California, Berkeley, Dec. 2005.

[45] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical Debugging of Sampled Programs. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *NIPS*. MIT Press, 2003.

[46] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical Debugging: Simultaneous Identification of Multiple Bugs. In W. W. Cohen and A. Moore, editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM, 2006.