# Scalable Temporal Order Analysis for Large Scale Debugging

Dong H. Ahn[1], Bronis R. de Supinski[1], Ignacio Laguna[3], Gregory L. Lee[1],
Ben Liblit[2], Barton P. Miller[2], Martin Schulz[1]

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

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

[3]Purdue University, School of Electrical and Computer Engineering,
West Lafayette, IN 47907, ilaguna@purdue.edu

## ABSTRACT

We present a scalable temporal order analysis technique that supports debugging of large scale applications by classifying MPI tasks based on their logical program execution order. Our approach combines static analysis techniques with dynamic analysis to determine this temporal order scalably. It uses scalable stack trace analysis techniques to guide selection of critical program execution points in anomalous application runs. Our novel temporal ordering engine then leverages this information along with the application's static control structure to apply data flow analysis techniques to determine key application data such as loop control variables. We then use lightweight techniques to gather the dynamic data that determines the temporal order of the MPI tasks. Our evaluation, which extends the Stack Trace Analysis Tool (STAT), demonstrates that this temporal order analysis technique can isolate bugs in benchmark codes with injected faults as well as a real world hang case with AMG2006.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-PROC-412227). It is supported in part by LLNL contracts B579934 and B580360; Department of Energy grants 93ER25176, 07ER25800, and 08ER25842; AFOSR grant FA9550-07-1-0210; and NSF grants CCF-0621487, CCF-0701957, and CNS-0720565. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.



## 1. INTRODUCTION

High Performance Computing (HPC) applications continue to grow in complexity, often combining large numbers of independent modules or libraries. Further, they are using unprecedented processor counts. These trends significantly complicate application design, development, and testing. In particular, interactive debugging techniques, as implemented in tools like gdb [1], DDT [2], or TotalView [3], no longer work well since they must gather data across all modules or libraries and provide little assistance in identifying tasks in which errors arise.

Novel techniques for bug detection and isolation, like Cooperative Bug Isolation (CBI) [4, 5] or DIDUCE [6], target this problem and provide users with (semi) automatic mechanisms to aid in discovering root causes in sequential programs. However, extending these approaches to parallel codes is not straightforward and requires a deeper understanding of the cross-task relationships and synchronizations. In our previous work, we developed STAT, the Stack Trace Analysis Tool [7, 8, 9], which provides a first step into this direction by aggregating stack trace information across all nodes. STAT uses this information to form equivalence classes of tasks, 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 tasks and for understanding the relationship between their execution state. This coarseness may miss critical differences or dependencies. Thus, STAT does not always allow bug isolation and root cause analysis. Instead, we must identify additional data that captures the relative execution progress in each task and that supports accurate mapping of the debug state across all tasks.

This work introduces a novel, non-intrusive and highly scalable mechanism that refines task equivalence sets and captures the progress of each task. This technique creates a partial order across the task 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 [10, 11],

to determine which application variables capture relative progress. We then extract their runtime values in all tasks to refine the task equivalence sets and to determine their relative execution progress. Similarly to previous work [12], 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. This paper makes these contributions:

- We define a **Metric and Notation for Application Progress** that captures the global application state and refines the notion of task equivalence;
- We introduce a **Task Progress Partial Order** that relates application state across processes;
- We develop a novel **Application Progress Extraction Technique** that combines static analysis with automated runtime debugging techniques;
- We detail a **Prototype Debugging Tool Set** that implements these techniques to detect fine grain equivalence classes and their relationships at large scales.

Our study shows that relative progress significantly reduces the effort to locate tasks that manifest errors for a wide array of faults. In particular, we identify randomly injected errors in the Block Tridiagonal (BT) benchmark of the NAS Parallel Benchmarks [13] and a real deadlock that AMG2006 [14] exhibited at 4,096 tasks. Our evaluation demonstrates that our technique extends STAT to provide these benefits with moderate additional overhead.

The remainder of this paper is structured as follows. Section 2 details the current state of the art in debugging methodologies. We then define the partial order that determines the relative progress of tasks in a parallel application in Section 3. We then detail our methodology, combining static and dynamic analysis, to gather that order in large scale application runs in Section 4. Section 5 then presents detailed experiments that evaluate the effectiveness and performance of our temporal ordering methodology.

## 2. STATE OF THE ART IN SEQUENTIAL AND PARALLEL DEBUGGING

Debugging is one of the fundamental steps in the software development cycle. However, current techniques, particularly for parallel applications, have reached their limits due to rising application and architectural complexity. Traditional debugging techniques, including sequential debuggers such as gdb and "printf debugging," in which users print key variables during repeated executions, require the user to identify coding errors and to trace their origins manually. Traditional parallel debuggers, such as TotalView [3] and DDT [2], are similar, although these tools must control multiple processes concurrently and aggregate the distributed state. They provide convenient interfaces to that state but the process of identifying errors remains manual. Overall, traditional debugging techniques require a significant amount of user experience, intuition, and time, and thus are not practical for large, complex applications.

Differential debugging provides a semi-automated approach to the analysis and understanding of programming errors by comparing executions [15, 16]. Recent research has focused on developing statistical techniques to pinpoint the root cause of correctness problems automatically [4, 5, 17, 6]. While both approaches hold significant promise, they require extensive runtime execution data often from multiple runs or extensions that guide the analyses to significant differences between the processes in a single run. As a result, most techniques have not yet been applied to large scale runs of parallel applications. Complementing these techniques, we need mechanisms that relate the state across the individual processes and group ones with similar behavior into task equivalence classes. These techniques will allow users not only to reason about the state of large sets of processes instead of having to look at each process individually, but also to isolate errors in outliers.

Our previous work developed the Stack Trace Analysis Tool (STAT) [7, 8], which provides scalable task equivalence class detection based on the functions that the processes execute. Specifically, STAT gathers stack traces across tasks and over time and merges the traces into a call graph prefix tree, from which it identifies task equivalence classes. Users can then attach a traditional parallel debugger to a single representative of a class with suspicious behavior or to representatives from several classes. In either case, the technique presents the user with much less data that is targeted at the underlying problem.

STAT builds on scalable and portable tool infrastructure such as the MRNet tree-based overlay network [18]. As a result, it has been ported and deployed on many high-end computing systems including Linux clusters, BlueGene/L and P, and the Cray XT4 and 5. The tool has also proven effective even at very large scales; it has demonstrated sub-second merging latencies on 212,992 tasks, larger than any previously published tool experiment [9].

STAT's stack trace analysis is useful for diagnosing certain classes of errors. It can quickly identify when a small subset of tasks has diverged from the rest of the application, which occurs, e.g., when most of the tasks are blocked in a barrier, waiting for one or a few other tasks. STAT's time-based sampling can also help to determine when processes are making progress or when processes are stuck in a single call path. Additionally, STAT's intuitive representation can quickly identify anomalous call paths, such as an unexpected application call sequence. In general, STAT is most effective when process behavior can be differentiated at the granularity of function calls, that is, when two processes can be classified as equivalent based on the function names in their stack trace.

STAT's function-based equivalence classes are often a useful starting point, but can group processes when their behaviors are distinct. For example, a function *foo* may contain multiple call sites to a function *bar*, which are all equivalent at the function level. However, an error may manifest itself due to the context in which *bar* is invoked, which we could detect by distinguishing the call sites. Similarly, two processes can exhibit the same stack trace despite being in different simulation time steps, which would often indicate that one process is more likely manifesting an error. Thus, function-based equivalence classes often do not differentiate tasks sufficiently so we must consider refinements. We must also understand the relationship between classes and identify the classes that are likely to provide insight into the root causes of errors. The latter task requires techniques that not only work at large scales but also relate the equivalence classes to details in the source code.

## 3. COMPARING PROCESS STATE THROUGH RELATIVE PROGRESS

We provide the required task 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

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

Table 1: Two MPI tasks executing the Poisson solver

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 tasks in a parallel application is an intuitively simple concept: we want to order tasks by how much of the dynamic execution they have completed. We can capture significant detail about the execution history of a task by considering stack traces. A single stack trace can provide simple but significant runtime data about a task'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 tasks 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 1 shows.

In any execution of the fragment in Figure 1a, 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 tasks they are essentially concurrent. However, the context of a full execution can change these orderings. A task at (5) has progressed further than a task at (6) if it is at later iteration of the loop at (4) (i.e., `it` is larger in the first task). Similarly, the value of `it` can order tasks 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 $it_2$ are the values of `it` in tasks 1 and 2.

Formally, we assume a parallel application with $N$ tasks 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 3.1. *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 task is executing code with a given control flow variable state that another task could have already executed with the same state then that first task 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 ($\forall i : P_i \preceq P_i$), antisymmetric ($\forall i, j : P_i \preceq P_j \wedge P_j \preceq P_i \Leftrightarrow P_i = P_j$) and transitive ($\forall i, j, k : P_i \preceq P_j \wedge P_j \preceq P_k \Rightarrow P_i \preceq P_k$). Relative progress is distinguished from previously explored partial orderings of parallel processes [19, 20] 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 tasks. However, in order to be practical, we must efficiently and scalably represent a task'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 1b, 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. The following defines our lexicographic order formally.

DEFINITION 3.2. *Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two program points within a function represented as a sequence of iteration counts and offsets, where each $a_i$ or $b_i$ represents a displacement between a program statement and the closest compound statement that contains it. Then $A \preceq B$ if and only if there is an index $j \geq 1$ such that $(\forall i < j) : (a_i = b_i) \wedge (a_j < b_j)$ or $(\forall i > 1) : (a_i = b_i)$.*

We must also ensure that we properly encode incomparable execution points, such as program points in distinct branches of a conditional statement. Therefore, we extend this definition to encode the information that enforces their incomparability. Thus, we modify Definition 3.2 as follows:

DEFINITION 3.3. *Let $x^{\in S}$ and $y^{\in S}$ denote offsets from distinct branches of the conditional statement S. Then we write $x \parallel y$ to indicate that x and y are incomparable.*

DEFINITION 3.4. *Now let A and B be as given in Definition 3.2. Then in our revised definition, $A \preceq B$ if and only if there is an index $j \geq 1$ such that $(\forall i < j) : (a_i = b_i) \wedge ((a_j < b_j) \wedge (a_j \nparallel b_j))$ or $(\forall i > 1) : (a_i = b_i)$.*

```
      /* 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 1a

Figure 1: MPI program solving the Poisson problem iteratively

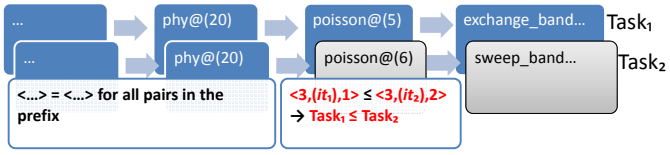



Figure 2: Annotated stack traces for two Poisson solver tasks; the first divergence determines relative progress

Under Definition 3.3 and Definition 3.4, we represent the program points (15) and (17) in Figure 1 as follows:

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

With these representations, (15) $\not\preceq$ (17) and (17) $\not\preceq$ (15) when the values of *it* are equal since $0^{\in 10} \parallel 2^{\in 10}$ and the execution points are incomparable in our partial ordering system as desired.

Figure 2 illustrates relative progress through the Poisson solver for a stack trace representation of two tasks. 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 tasks are equal up to the invocation of *poisson*. Thus, the lexicographical order of (5) and (6) determines the relative progress of tasks 1 and 2, eliminating a need to evaluate later frames. We exploit this property to determine relative progress efficiently for large scale applications.

# 4. AUTOMATIC EXTRACTION OF APPLICATION PROGRESS

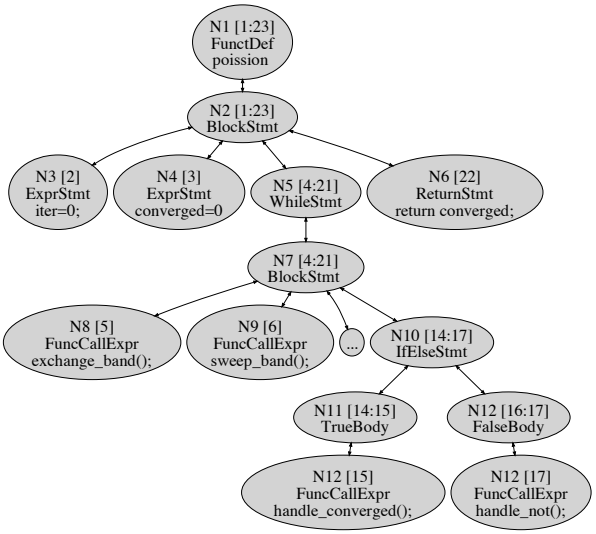
We introduce the necessary analysis techniques to determine the progress of a running process in this section. We begin by showing how to determine the components of our lexicographic order in general, which we split into two steps. The first step finds necessary offsets while the second determines the 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.

## 4.1 Program Point Rewriting System

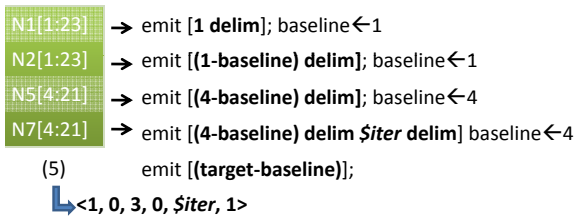
We first present a simple but efficient abstract syntax tree (AST) analysis technique that translates a program location into the representation described in Section 3. Our system represents a program location by the source file and line number of an instruction. We adopt this simplification since most 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.

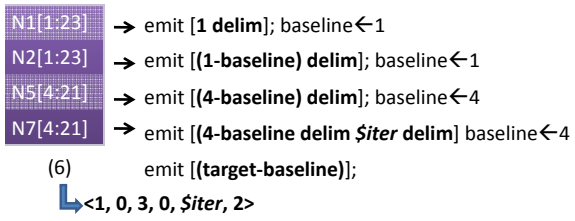
Our analysis uses ROSE [10, 11], a compiler infrastructure that parses high-level language source files and provides mechanisms to manipulate the resulting AST. We use a set of line numbers within a target function as input (we assume the binary is properly compiled with source and line number directives 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.

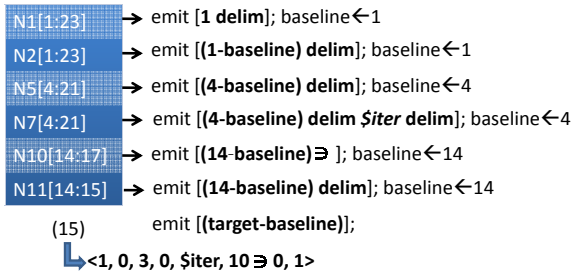



(a) AST built from Figure 1

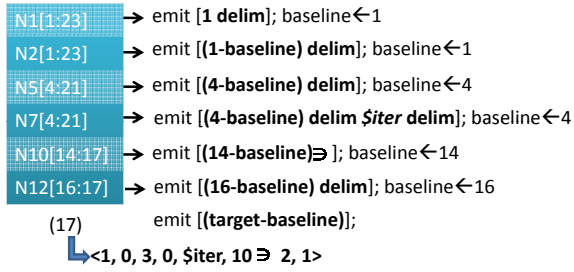



(b) Rewriting of (5) and (6)




(c) Rewriting of (15)




(d) Rewriting of (17)

Figure 3: Program point rewriting system

Figure 3a shows the AST for the code shown in Figure 1. Figure 3b through Figure 3d depict how our system emits a series of tokens to generate the lexicographical representations of the program points (5), (6), (15) and (17) within the `poisson` function. These tokens represent $x^{\in S}$ as $S \ni x$.

## 4.2 Loop Order Variable Analysis

Clearly, static analysis alone cannot fully resolve our lexicographic order when program points are contained in a loop. The technique described in Section 4.1 only produces a placeholder for the iteration count. We could instrument each loop of the program statically with an iteration counter, which would impede many compiler optimizations, and fetch its runtime value if needed for ordering. However, scientific applications have many loops, most of which are not *interesting* for a particular debugging problem, so the costs are usually unnecessary. 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 4.1. *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 [21, 22, 23]. 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 [24, 25]. 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 4.2. *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*

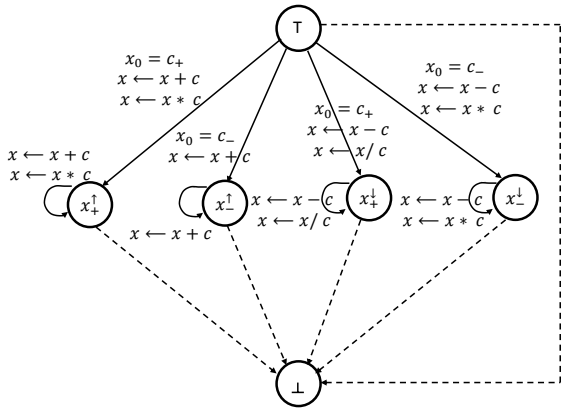



Figure 4: LOV candidate variable state changes; dotted lines represent transitions for unlisted conditions

**(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 4.3. *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 4.2 or multiple definitions of v inside l reach to the use.*

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 [22] 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 4.2 and Definition 4.3. 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 4 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.

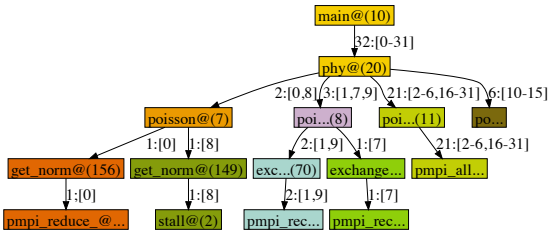
Our algorithm identifies `it` in Figure 1 as a LOV with the $x_+^{\uparrow}$ attribute with respect to the loop that spans (4) and (21). The loop has only one corresponding monotonic statement, `it++` at (20), and it has an unambiguous, implicit use of `it` with an unambiguous initial value with the $+$ 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 in Definition 4.1. So, we can use its runtime value instead of the placeholder established in Section 4.1.

## 4.3 Program Point Selection Strategy

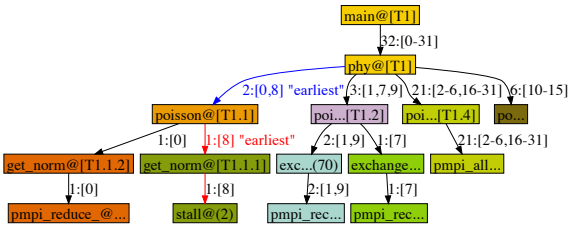
Our combined static and dynamic analysis method to determine the relative progress of tasks could incur significant overhead. Transforming all stack trace frames into the lexicographical representation on compute nodes would allow a trace merging engine like STAT to resolve the lexicographic order at 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 [9] 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.




(a) Chronology-unaware prefix tree




(b) Adaptive refinement of Figure 5a

Figure 5: Execution chronology-unaware vs. aware call graph prefix tree for the program in Figure 1 at 32 tasks

To illustrate this, we introduce an artificial bug in Figure 1 that stalls one MPI task in the `get_norm` function. This Poisson solver divides the domain up into parallel horizontal strips, and has MPI tasks cooperatively perform the Jacobi iteration until it detects convergence. Due to communication dependencies, stalling a task quickly leads to a hang in which most tasks reach a global synchronization point (e.g., `MPI_Allreduce`) waiting for the other tasks that cannot make progress due to dependence on the stalled task.

On an application run that hung at 32 MPI tasks, Figure 5a shows the execution chronology-unaware tree. While it describes stack traces succinctly, representing each process behavior equivalence class with a distinct node color and participating MPI rank tasks information with caller-callee edge labels (`task count:[rank or rank range]`), the unordered tree does not provide any hints as to which classes the user should examine first. Such hints would clearly be useful since many classes are formed even at a small scale. On the other hand, Figure 5b captures the relative progress order by re-drawing node labels from `func@lineNumber` to `func@[`$T_{TemporalRank}$`]`. We see that two refinement steps reveal the least progressed task (rank 8), the one in which the stall occurred. While ours is not the only method to detect this deadlock condition [26], this example demonstrates that the execution chronology-aware tree quickly locates the least progressed tasks, which is very useful information for many types of errors.

# 5. EXPERIMENTS

This section presents experiments that 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 [10, 11] to implement the analyses described in Section 4.1 and Section 4.2.

To start, STAT gathers stack traces, each of their frames containing the function name as well as source file and line number information based on the program counter. We keep the application halted to preserve its state, in case we need to extract runtime state from the processes. We later use the source file and line number information as the program point input to our static analysis engine. We use STAT's existing scalable merging algorithm to handle the fine-grain stack traces (changes to support precise source file and line number information are straightforward) and the resulting merged call graph prefix tree forms an execution chronology-unaware representation from which we can begin the refinement process. During each adaptive refinement step, we input a set of execution points in the same routine into our static analysis engine. If the refinement requires the values of a loop order variable from the live application processes, we multicast a request from the STAT front end to extract the runtime values to its backend daemons. The daemons subsequently communicate the result through STAT's scalable merging routine. At the end of each refinement step, STAT re-draws the selected region of the prefix tree, transforming that region into a temporally ordered tree by encoding the temporal rank into each node label.

## 5.1 Fault Injection

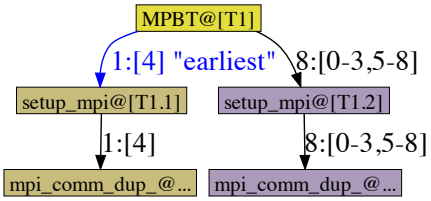
Our first experiments empirically evaluate the effectiveness of our prototype through injecting faults into BT from the NAS parallel benchmarks, which solves multiple independent systems of non-diagonally dominant, block tridiagonal equations with a $5 \times 5$ block size [13]. Our fault injector can introduce a wide array of software faults dynamically into MPI application runs. It monitors MPI function calls using P$^N$MPI [27] and statistically introduces a fault into the address space of a task. We designed the faults to emulate common application bugs. Currently, it supports three main classes of faults: data corruption; local livelock/deadlock; and communication errors. It emulates data corruption faults by three different methods: increasing or decreasing the value of an element in an MPI message by a set amount; replacing it with a random value; or overrunning a configurable number of bytes beyond the message boundary. We emulate local livelock/deadlock conditions by injecting an infinite loop. Finally, we simulate a communication error by sending a configurable number of extra MPI messages. Our injector ensures fair observations of an application's behavior against a fault by randomly selecting injection parameters such as target MPI rank, routine, data type and fault type, based on the actual communication profile of the application and on the fault catalogue.

The impact of fault injection varies widely. Often, the application runs to completion with no apparent errors. Other times, it completes but with some corrupted output or it hangs or aborts. Table 2 summarizes the results of our experiments with BT with the C class problem at nine MPI tasks. As its fourth column indicates, the overall activation rate, the percentage of runs in which an injected fault results in a manifest symptom, is 56%. Of these detected erroneous runs, execution hangs and aborts occur most frequently, accounting for nearly 58% of the manifest errors. We focus on this class since users would investigate the stack trace information of these hard errors to gain insight into the root cause. More specifically, we randomly select one case out of each of the six fault types among the hang cases and apply our prototype. We consider an abort as a special hang case since we can easily map it to a hang by installing a signal handler for all anomalous UNIX signals such as `SIGSEGV`, in which the execution simply stalls.

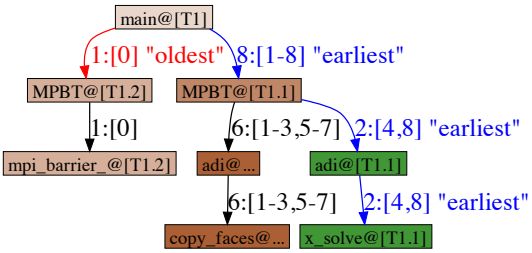
Figure 6 depicts the resulting temporally ordered STAT graphs on these six cases. They show that the infinite loop (Figure 6a) and buffer overflow (Figure 6d) faults affect BT's setup phase while the extra message (Figure 6c) and value change (Figure 6b) faults interfere with the main iteration. Also, it shows that both the value increase (Figure 6e) and decrease (Figure 6f) faults manifest themselves at `MPI_Barrier` sites.

| Fault Type | No. Injections | No. Errors | Activation (%) | Error type distribution (%) | | |
|---|---|---|---|---|---|---|
| | | | | Soft Error | Hang | Abort |
| Value Change | 288 | 262 | 90.97 | 80.53 | 3.05 | 16.41 |
| Value Increase | 289 | 173 | 59.86 | 81.50 | 16.18 | 2.31 |
| Value Decrease | 290 | 196 | 67.59 | 83.57 | 11.22 | 5.10 |
| Buffer overflow | 868 | 392 | 45.16 | 15.56 | 17.35 | 67.09 |
| Infinite loop | 273 | 273 | 100.00 | 0.00 | 100.00 | 0.00 |
| Extra messages | 419 | 63 | 15.04 | 3.65 | 93.65 | 0.00 |

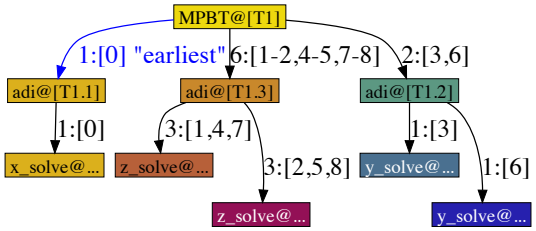
Table 2: Fault activation rates and distribution of observed errors in the NPB BT



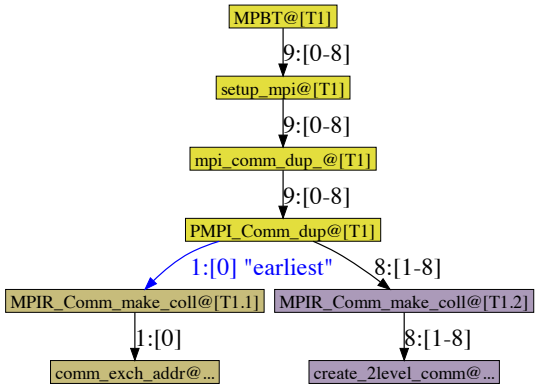

(a) Infinite loop: fault injected into rank 4



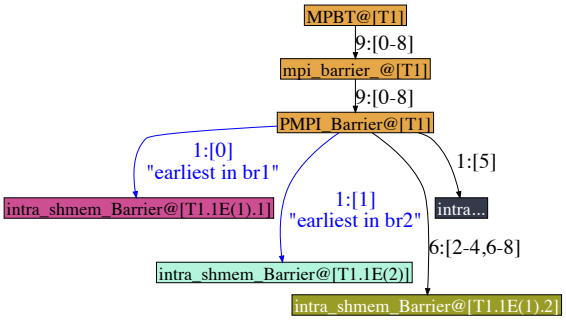

(b) Value change: fault injected into rank 0



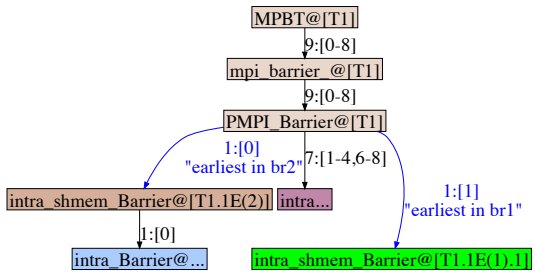

(c) Extra messages: fault injected into rank 0




(d) Buffer overflow: fault injected into rank 0




(e) Value increase: fault injected into rank 1




(f) Value decrease: fault injected into rank 0

Figure 6: Temporal analysis on BT for each of the six injected fault types

While the fault types, and the affected ranks and execution points vary, our experiments verify that the prototyped technique supports walking the prefix tree with relative task order information, and more importantly the quick isolation of the task from which the error originated. That is, for all but the value change case, it locates the offending task in the least progressed task as edges in blue in Figure 6 indicate. In the case of the value increase and decrease faults, our prototype identifies several incomparable children nodes, indicating the tasks are in distinct branches of a conditional statement at that level. In both fault types, however, the offending task takes a conditional branch distinct from that taken by the other tasks. Thus, the offender represents the only, and therefore, least progressed task executing that branch by our convention. For the value change fault, our prototype attributes the error to the most progressed task (rank 0) and we find that problem size assertion failed due to the value change at which point the task jumped to the MPI barrier right before the `MPI_Finalize` call.

In summary, this empirical study suggests that finding either the least or most progressed tasks simplifies finding the execution point closer to the root cause. The study also verifies that our temporal order technique effectively enables finding these tasks in the distributed execution space.

## 5.2 Case Study on AMG2006

This case study describes our experience applying the prototype to Algebraic MultiGrid (AMG) 2006. AMG2006 is a scalable iterative solver and preconditioner for solving large unstructured sparse linear systems that arise in a wide range of science and engineering applications. In preparation for extending it to unprecedented numbers of multicore compute nodes, the user 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 7 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 7. 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` → `big_insert_new_nodes` → `hypre_ParCSRCommHandleDestroy` path. We quickly found a type coercion problem for `int offset`, a function parameter of `big_insert_new_nodes`. The application team 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. Once the team corrected this error, they confirmed correct execution at this scale.

| Trace Type | Sample Time | Merge Time |
|---|---|---|
| Function Name Only | 0.74 | .004 |
| Function Name and Line Number | 6.57 | .004 |

Table 3: STAT sample and merge times for a 256 task job in seconds

## 5.3 Performance

We tested the performance of STAT with temporal analysis on Hera, an 864 node Linux cluster with an Infiniband interconnect at Lawrence Livermore National Laboratory. Each node of Hera has four quad core AMD Opteron chips running at 2.3 GHz. We performed tests on an MPI message ring topology program, the same test driver used in our previous work [7, 9] for performance evaluations. In this simple program, each task performs an `MPI_Irecv` from the previous task in the ring and an `MPI_Isend` to the next task, followed by an `MPI_Waitall` and an `MPI_Barrier`. A bug is introduced into MPI rank task 1 that causes it to hang before its send.

We first compare the sampling and merging times for STAT gathering function names only versus STAT gathering function names as well as the source file and line number tuples. These comparisons were run on 16 nodes, 256 MPI tasks, and employed a flat 1-to-N topology in STAT. The results in Table 3 show a significant, but still tolerable, increase in the time required to gather the source file and line number of each frame in the stack traces. The additional time arises from parsing additional debug information in order to obtain the more detailed information. In contrast, the merge time showed no increase, despite the additional information, and took only four milliseconds.

Next we measure the time to perform the temporal analysis on the resulting call graph prefix tree. With this application, three equivalence classes are formed in main, with one task hung in its send, another task in an `MPI_Waitall`, and the remaining 254 tasks in an `MPI_Barrier`. Our measurements found that the first adaptive refinement of this prefix tree, which identifies the relative progress of these three equivalence classes, took 1.58 seconds. In the absence of active loops, an adaptive refinement step requires STAT to perform a one-time static analysis on the front-end node. Thus this performance represents a constant overhead independent of scale at which the program runs that is acceptable for interactive debugging sessions.

# 6. CONCLUSION

Correct MPI code proceeds forward in a coordinated manner. When execution fails, disruption of the orderly temporal plan reveals important clues as to the root cause. At supercomputer scales, though, one can no longer rely on programmer intuition to recognize which parts of a large application have fallen behind. We have codified this intuition as a formal partial order among program states. The partial order uses a blend of static and dynamic measures: static ordering based on the control-flow structure of task code, and dynamic ordering based on Loop Order Variables (LOVs), a novel variation of loop induction variable analysis.

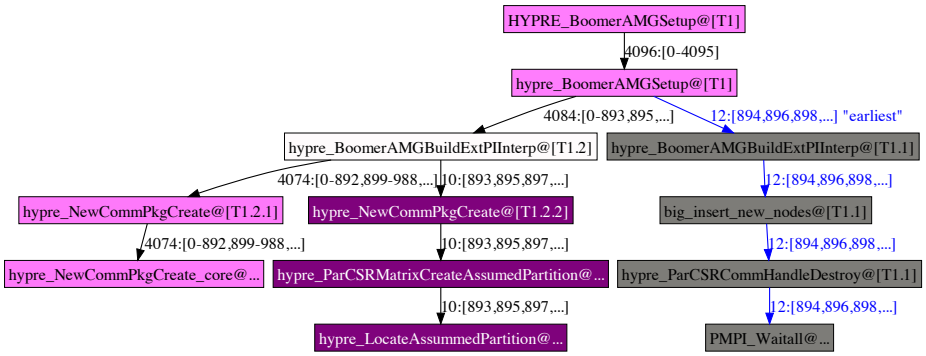



Figure 7: Chronology-aware prefix tree for a code hang exhibited by AMG2006 at 4,096 MPI tasks

Taken together, these static and dynamic views offer hierarchical progress markers that reflect relative logical progress through an execution.

We have implemented our temporal ordering analysis as an extension of the Stack Trace Analysis Tool (STAT). Our implementation achieves scalability by starting with a chronology-unaware call graph prefix tree that we then refine to add temporal ordering under user guidance. We expressly designed our implementation for use in production environments: it requires absolutely no changes to application source code; it imposes zero runtime overhead on running programs; and retrieves all needed dynamic information postmortem, using the standard debugger interface. Experiments with our implementation show that it is a highly effective debugging tool. In a controlled study using randomized fault injection, we quickly identified root causes across six typical classes of MPI bugs. Temporal order analysis has proven itself in the field as well, allowing us to diagnose and repair a real-world bug in a complex application that does not manifest itself with under 4,096 tasks. Finally, our scalability experiments indicate that our extensions add primarily constant overheads to STAT, which has run successfully with 212,992 tasks.

Thus, our formal temporal partial order is not merely interesting theoretically but is also an excellent basis for real, useful debugging tools. For future work, we will explore several extensions, including more techniques to handle a loop with no true LOV. We will allow the user to identify LOVs, providing a list of possible LOVs with some properties that our analysis cannot resolve. To overcome aggressive compiler optimizations that can lead to significant differences between observed code behavior and its source code, we will also explore ways to perform our static analyses directly on the target binary.

## 7. REFERENCES


[1] GDB Steering Committee, "GDB: The GNU Project Debugger," http://www.gnu.org/software/gdb/documentation/.

[2] Allinea Software, "Allinea DDT the Distributed Debugging Tool," http://www.allinea.com/index.php?page=48.

[3] TotalView Technologies, "TotalView Debugger," http://www.totalviewtech.com/productsTV.htm.

[4] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu, "Statistical debugging using latent topic models," in *18th European Conference on Machine Learning*, S. Matwin and D. Mladenic, Eds., Warsaw, Poland, Sep. 17–21 2007.

[5] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *31st International Conference on Software Engineering (ICSE 2009)*, J. Atlee and P. Inverardi, Eds. Vancouver, Canada: ACM SIGSOFT and IEEE, May 2009.

[6] S. Hangal and M. S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," in *ICSE '02: Proceedings of the 24th International Conference on Sof tware Engineering*. New York, NY, USA: ACM, 2002, pp. 291–301.

[7] 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.

[8] 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 International Conference ParCo 2007)*, Julich/Aachen, Germany, 2007.

[9] 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*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–9.

[10] 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*, ser. Lecture Notes in Computer Science, vol. 1543. Springer Verlag, July 1998.

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

[12] 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*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 74–81.

[13] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS parallel benchmarks," NASA Ames Research Center, RNR-91-002, Aug. 1991.

[14] V. E. Henson and U. M. Yang, "Boomeramg: a parallel algebraic multigrid solver and preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, pp. 155–177, 2002.

[15] D. Abramson, I. Foster, J. Michalakes, and R. Socič "Relative Debugging: A New Methodology for Debugging Scientific Appl ications," *Communications of the ACM*, vol. 39, no. 11, pp. 69–77, 1996. [Online]. Available: citeseer.ist.psu.edu/article/abramson96relative.html

[16] G. Watson and D. Abramson, "Relative Debugging for Data-Parallel Programs: A ZPL Case Study," *IEEE Concurrency*, vol. 8, no. 4, pp. 42–52, 2000.

[17] Q. Gao, F. Qin, and D. K. Panda, "Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

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

[19] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[21] 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.*, vol. 17, no. 1, pp. 85–122, 1995.

[22] M. Spezialetti and R. Gupta, "Loop monotonic statements," *IEEE Trans. Softw. Eng.*, vol. 21, no. 6, pp. 497–505, 1995.

[23] 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*. New York, NY, USA: ACM, 2001, pp. 78–91.

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

[25] 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*. New York, NY, USA: ACM, 2007, pp. 194–204.

[26] D. Kranzlmüller, S. Grabner, and J. Volkert, "Event graph visualization for debugging large applications," in *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. New York, NY, USA: ACM, 1996, pp. 108–117.

[27] M. Schulz and B. R. de Supinski, "PNMPI tools: a whole lot greater than the sum of their parts," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.