Providing Order to
Extreme Scale Debugging Chaos

LLNL-PRES-412482  Paradyn/Dyninst Week, April 28, 2009

Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Martin Schulz

Dorian C. Arnold, Matt LeGendre, Ben Liblit, Barton P. Miller

Lawrence Livermore National Laboratory
University of Wisconsin

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

- We have 100,000's of cores, with more on the way.
- Architecture trends put enormous pressure to applications.
- Applications continue to grow in complexity.
- We struggle to deal with all these factors… yet tools are even further behind…

University of Wisconsin

Lawrence Livermore National Laboratory
LLNL Petascale Debugging Strategy

Feature Set

Root cause analysis at manageable subset

Scalable, lightweight infrastructure and tools

Lightweight debug engines

Advanced communication infrastructure

Automatic debug into analyses

Problematic neighbors detector

Scalable, lightweight infrastructure and tools

Scalar

Petascale
STAT: the Stack Trace Analysis Tool

- STAT gathers stack traces from:
  - Multiple processes
  - Multiple samples per process
Equivalence Class Reduces Search Space
Scalable Architecture for 3 Core Operations

1) Startup

2) Sample

3) Merge
Scalability milestones

- **[2006]** prototype demonstrated sub-second latencies at 3,844 tasks
  

- **[2007]** STATBench estimated latency to be under three seconds at 131,072 tasks on BG/L with an edge label optimization
  

- **[2008]** Further optimizations led to sub-second latencies on 208K(212,992) cores of BG/L
  
Production use indicates…

- Most effective when behavior is differentiated at granularity of function calls
- Does not provide metrics to detect “crucial” equivalence classes in which the error originated
  - Relies on simple heuristics to guide users (e.g., Longest/shortest path and persistent classes)
- Generally lacks a systematic framework to provide increasingly fine analysis granularity with an ability to enable the identification of crucial classes
STAT on JetCold Deadlock at 512 MPI Tasks

Analysis is too coarse to provide a next step
STAT calls for levels of analysis detail

- General direction
  - Combine best from static and dynamic analysis techniques to provide increasingly fine granularity
  - Support detection of crucial equivalence classes
  - Maintain extreme scalability

- The rest of the talk: a preview of temporal order analysis technique, a fine-grain domain analysis using static-dynamic techniques to determine the relative progress order of MPI tasks
What is the temporal order analysis and why?

- Order tasks by how much of the dynamic execution they have completed
- Reasoning about the chronology of execution space is essential for debugging
  - Runtime stack of sequential execution…
  - Manual `printf` debugging…
- How can we do this non-intrusively for large scale MPI applications?
Logical program execution provides a means to evaluate relative progress order among tasks.

```c
int poisson() {
  it = 0;
  converged = 0;
  while (!converged) {
    if (!converged)
      break;
    it++;
  }
  return converged;
}
```

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Relative iteration count</th>
<th>Logical program execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5)</td>
<td>(6)</td>
<td></td>
<td>(it_1 \leq it_2)</td>
<td>Task 1 is behind Task 2</td>
</tr>
<tr>
<td>(5)</td>
<td>(6)</td>
<td></td>
<td>(it_1 &gt; it_2)</td>
<td>Task 2 is behind Task 1</td>
</tr>
<tr>
<td>(15)</td>
<td>(17)</td>
<td></td>
<td>(it_1 &lt; it_2)</td>
<td>Task 1 is behind Task 2</td>
</tr>
<tr>
<td>(15)</td>
<td>(17)</td>
<td></td>
<td>(it_1 &gt; it_2)</td>
<td>Task 2 is behind Task 1</td>
</tr>
<tr>
<td>(15)</td>
<td>(17)</td>
<td></td>
<td>(it_1 = it_2)</td>
<td>No relative progress order between tasks</td>
</tr>
</tbody>
</table>

Table 1: Two MPI tasks executing the Poisson solver.

```c
if (converged)
  handle_converged(...);
else
  handle_not(...);
```
We transform the temporal ordering of program points within a function into a lexicographical ordering.

```
int poisson()
{  
  it = 0;
  converged = 0;
  while ( it < MAX )
    ...  
  if ( converged )
    break;
  it++;
  return converged;
}
```

\[ (5) \Rightarrow (4-1), (it), (14-14) \in (14-4), (15-14) \Rightarrow (3), (it), 0^{10}, 1 \]

\[ (15) \Rightarrow (4-1), (it), (14-14) \in (14-4), (15-14) \Rightarrow (3), (it), 0^{10}, 1 \]

\[ (17) \Rightarrow (4-1), (it), (16-14) \in (14-4), (17-16) \Rightarrow (3), (it), 2^{10}, 1 \]
This stack trace representation extends the partial order to MPI tasks.

\[ \langle \ldots \rangle = \langle \ldots \rangle \text{ for all pairs in the prefix} \]

\[ \langle 3, (it_1), 1 \rangle \leq \langle 3, (it_2), 2 \rangle \rightarrow \text{Task}_1 \leq \text{Task}_2 \]
AST-based program point rewriting system

emit [1 delim]; bline ← 1
emit [(1-baseline) delim]; bline ← 1
emit [(4-baseline) delim]; baseline ← 4
emit [(4-baseline) delim $iter delim]; bline ← 4
emit [(target-baseline)];

<1, 0, 3, 0, $iter, 1>
Loop Order Variable (LOV)

- Key program variables that the runtime state of which can resolve relative progress for loops
- Variables that satisfy following properties

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., \( \forall p_1, p_2, x_i(p_1) = x_i(p_2) \)).
Loop Order Variable (LOV) Analysis

(1) Assume that the def-use chain is available, and thus loop invariants are also available.

(2) Fetch all the expression statements that define a variable.

(3) Iterate over each stmt, and try to reduce it into a basic monotonic form (i.e., \( x \leftarrow x + c \)) ; if reduced, test the use of variables for ambiguity.

(4) Use the FSM to keep track of the variable each stmt defines.

(5) Once the iteration is finished, variables with non-chaos states become true LOVs of the target loop.
Ambiguity

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

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).
Adaptive prefix tree refinement adds scalable "chronological walking" direction over tasks.
STAT isolated a real deadlock in AMG 2006 at 4K tasks.
Bug injection experiment on BT indicates...

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>No. Injections</th>
<th>No. Errors</th>
<th>Activation (%)</th>
<th>Error type distribution (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Change</td>
<td>288</td>
<td>262</td>
<td>90.97</td>
<td>Verification: 51.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 3.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 16.41</td>
</tr>
<tr>
<td>Value Increase</td>
<td>289</td>
<td>173</td>
<td>59.86</td>
<td>Verification: 56.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 16.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 2.31</td>
</tr>
<tr>
<td>Value Decrease</td>
<td>290</td>
<td>196</td>
<td>67.59</td>
<td>Verification: 54.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 11.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 5.10</td>
</tr>
<tr>
<td>Buffer overflow</td>
<td>868</td>
<td>392</td>
<td>45.16</td>
<td>Verification: 15.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 17.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 67.09</td>
</tr>
<tr>
<td>Infinite loop</td>
<td>273</td>
<td>273</td>
<td>100.00</td>
<td>Verification: 0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 100.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 0.00</td>
</tr>
<tr>
<td>Extra messages</td>
<td>419</td>
<td>63</td>
<td>15.04</td>
<td>Verification: 6.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hang: 93.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Abort: 0.00</td>
</tr>
</tbody>
</table>

Table 2: Fault activation rates and distribution of observed errors in the NPB BT.
Walking towards the least or most progressed class quickly isolates the culprit task.
Conclusion

- Our petascale debugging strategy has guided us to take tactical initiatives to rescue ourselves from extreme debugging chaos.
- STAT has demonstrated its scalability to the world’s largest core count.
- Through production use, STAT has proven effective in diagnosing a common class of bugs that emerge at large scale.
- Combining best from static and dynamic analysis promises to isolate even larger classes of bugs still scalably, and temporal order analysis is the first response to deliver on this promise.
Drowning? We don’t have to!