Dynamic Analysis

CS252r Fall 2015
Reading

- *The Concept of Dynamic Analysis* by Thomas Ball, FSE 1999
- *Efficient Path Profiling* by Thomas Ball and James Larus, MICRO 1996
- *Adversarial Memory for Destructive Data Races* by Cormac Flanagan and Stephen Freund, PLDI 2010
Dynamic Analysis

• Analysis of the properties of a running program
• Static analysis typically finds properties that hold of all executions
• Dynamic analysis finds properties that hold of one or more executions
  • Can't prove a program satisfies a particular property
  • But can detect violations and provide useful information
• Usefulness derives from precision of information and dependence on inputs
Precision of Information

- Dynamic analysis typically instructs program to examine or record some of run-time state
- Instrumentation can be tuned to precisely data needed for a problem
Dependence on Program Inputs

- Easy to relate changes in program inputs to changes in program behavior and program output

**Dynamic Analyses** are input-centric

**Static Analyses** are program-centric
Complementary Techniques

• Completeness
  • Dynamic analyses can generate "dynamic program invariants", i.e., invariants of observed execution; static analyses can check them
  • Dynamic analyses consider only feasible paths (but may not consider all paths); static analyses consider all paths (but may include infeasible paths)

• Scope
  • Dynamic analyses examine one very long program path
    • Can discover semantic dependencies widely separated in path and in time
    • Static analyses typically and at discovering "dependence at a distance"

• Precision
Two (plus a bonus) Dynamic Analyses

- Frequency Spectrum Analysis
  - Efficient path profiling
- Dynamic race detection
Frequency Spectrum Analysis

- Understanding frequency of execution of program parts can help programmer:
  - partition program by levels of abstraction
  - find related computations
  - find computations related to specific attributes of input or output
#include <stdio.h>
main(t,_,a)
  char *a;
  {
    return!0<t?t<3?main(-79,-13,a+main(-87,1-_ main(-86,0,a+1)+a)):1,
t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?<_<13?
main(2,+1,"%s %d %d\n") :9:16:t<0?t<-72?main(_,t,
  "@n'+,#/*{w+w#cdnr/+,[}r/*/de]+,/*{+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#
   ;q#n+,/+k#;*+,/'r :d*3,}{w+K w'K:'+}e#';dq#'l \nq'#+d'K#!/+k#;q'#r}eKK#{nl]'/#;/q#n'}{#}w')}{nl]'/+#n';d}rw' i;#
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+ Krw iK{[nl]'/w#q#n'wK nw' \iwr{KK{nl]!/w{%l##w' i; :{nl]}/#{q#'ld;r'}{nlwb!/de}'c \
 ;{nl]-'{}rw'+'/+,'##*}'#nc,',#nw']/+kd'+e}+;#'rdq#w! nr'/' ') }+}{rl#{n' ')# 
}'+}##(!="
:t<-50?==*a?putchar(31[a]):main(-65,_,a+1):main(*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):a=='/''l!main(0,main(-61,*,a,
  "!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:
uwloca-0;m .vpbks,fxntdCeghiry"),a+1);"}
What it does...

$ gcc -w obfus.c
$ ./a.out
On the first day of Christmas my true love gave to me a partridge in a pear tree.

On the second day of Christmas my true love gave to me two turtle doves and a partridge in a pear tree.

On the third day of Christmas my true love gave to me three french hens, two turtle doves and a partridge in a pear tree.
Program understanding

• We know **what** the program does
• Our aim is to understand **how** it does it
• Before reverse engineering it, let's have a model in mind:
  • Gift $t$ mentioned $13-t$ times in the poem (e.g. "five gold rings" occurs $13-5=8$ times)
  • So $1+2+...+11+12 = 13\times6 = 78$ gift mentions (66 mentions of non-partridge gifts)
  • All verses except first have form
    
    \[
    \text{On the } <\text{ordinal}> \text{ day of Christmas my true love gave to me} \\
    <\text{list of gift phrases, from the ordinal day down to the second day}> \\
    \text{and a partridge in a pear tree.}
    \]

    and first verse is
    
    \[
    \text{On the first day of Christmas my true love gave to me} \\
    \text{a partridge in a pear tree.}
    \]

• Unique strings:
  • 3 strings for common structure ("On the", "day of Christmas...", "and a partridge ...")
  • 12 strings for the ordinals
  • 11 strings for the second through twelfth gifts.
  • $\Rightarrow$approx. $3+12+11 = 26$ unique strings in program, prints approx. $3\times12 + 12 + 66 = 114$ strings.
Model

- 12 days of Christmas (also 11, to catch "off-by-one" cases)
- 26 unique strings
- 66 occurrences of non-partridge-in-a-pear-tree presents
- 114 strings printed, and
- 2358 characters printed.
Program understanding

First let's make it readable:

```c
#include <stdio.h>
main(t,_,a) char *a;
{
    if (!0 < t) {
        if (t < 3)  main(-79,-13,a+main(-87,1-,main(-86,0,a+1)+a));
        if (t < _)  main(t+1,_,a);
        main(-94,-27+t,a);
        if (t==2 && _ < 13 ) main(2,_,1,""');
    } else if (t < 0) {
        if (t < -72) main(_,t,LARGE_STRING);
        else if (t < -50) {
            if (_. == *a) putchar(31[a]);
            else main(-65,_,a+1);
        } else main((*a=='/')+t,_,a+1);
    } else if (0 < t) main (2,2,"%s");
    else if (*a=='/') main(0,main(-61,*a,SMALL_STRING),a+1);
}
```
Path Profiling

- Count executions of paths of the function
  - E.g., path executed 2358 times likely involved in printing characters

<table>
<thead>
<tr>
<th>Path ID</th>
<th>Frequency</th>
<th>Condition</th>
<th>Call Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:0</td>
<td>1</td>
<td>( t == 1 )</td>
<td>[9]</td>
</tr>
<tr>
<td>main:19</td>
<td>1</td>
<td>( t == 2 ) &amp;&amp; ( t &gt;= _ )</td>
<td>[1,3,4]</td>
</tr>
<tr>
<td>main:22</td>
<td>1</td>
<td>( t == 2 ) &amp;&amp; ( t &lt; _ ) &amp;&amp; ( _ &gt;= 13 )</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td>main:23</td>
<td>10</td>
<td>( t == 2 ) &amp;&amp; ( t &lt; _ ) &amp;&amp; ( _ &lt; 13 )</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>main:9</td>
<td>11</td>
<td>( t &gt;= 3 ) &amp;&amp; ( t &gt;= _ )</td>
<td>[3]</td>
</tr>
<tr>
<td>main:13</td>
<td>55</td>
<td>( t &gt;= 3 ) &amp;&amp; ( t &lt; _ )</td>
<td>[2,3]</td>
</tr>
<tr>
<td>main:2</td>
<td>114</td>
<td>( t == 0 ) &amp;&amp; ( *a == '/' )</td>
<td>no call lines</td>
</tr>
<tr>
<td>main:3</td>
<td>114</td>
<td>( t &lt; -72 )</td>
<td>[5]</td>
</tr>
<tr>
<td>main:1</td>
<td>2358</td>
<td>( t == 0 ) &amp;&amp; ( *a != '/' )</td>
<td>[10]</td>
</tr>
<tr>
<td>main:7</td>
<td>2358</td>
<td>( t &gt; -72 ) &amp;&amp; ( t &lt; -50 ) &amp;&amp; ( _ == *a )</td>
<td>[6]</td>
</tr>
<tr>
<td>main:4</td>
<td>24931</td>
<td>( t &lt; 0 ) &amp;&amp; ( t &gt;= -50 )</td>
<td>[8]</td>
</tr>
<tr>
<td>main:5</td>
<td>39652</td>
<td>( t &gt; -72 ) &amp;&amp; ( t &lt; -50 ) &amp;&amp; ( _ != *a )</td>
<td>[7]</td>
</tr>
</tbody>
</table>

**Table 2.** Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.
Path Profiling

Efficient Path Profiling, Ball and Larus, MICRO 1996
Problem: path profiling

• Which **paths** through a procedure are most common?
  • e.g., perform aggressive optimization on hot paths, make sure all paths are tested.

• Naive approach: count edge transitions

```
Path     Prof1 | Prof2
ACDF     90    | 110
ACDEDF   60    | 40
ABCDF    0     | 0
ABCDEDF  100   | 100
ABDF     20    | 0
ABDEF    0     | 20
```

• Not enough information to determine paths!
Efficient Path Profiling

• (For DAGs)
  • Encode each path as a unique integer and record path as state
    • i.e., at end of DAG, value of a register identifies path through DAG

![DAG Diagram]

**Path** | **Encoding**
---|---
ACDF | 0
ACDEF | 1
ABCDF | 2
ABCDEF | 3
ABDF | 4
ABDEF | 5

`count[r]++`
Algorithm overview

• 1. Number paths uniquely
• 2. Use spanning tree to select edges to instrument (and compute appropriate increment for each instrumented edge)
• 3. Select appropriate instrumentation
• 4. After profiling, given path number, figure out which path it corresponds to
Compact path numbering

- Aim: assign non-negative constant value to each edge such that sum of values along any path from ENTRY to EXIT is unique. Moreover, path sums should be in range 0..(NumPaths - 1) (i.e., minimal encoding)
foreach vertex v in reverse topological order {
    if v is a leaf vertex {
        NumPaths(v) = 1;
    } else {
        NumPaths(v) = 0;
        for each edge e = v->w {
            Val(e) = NumPaths(v);
            NumPaths(v) = NumPaths(v) + NumPaths(w);
        }
    }
}
Efficiently compute sums

- Find min-cost spanning tree (= max cost chord edges)
  - Chord edges will be instrumented
- Move weights from non-chord edges to chord edges

![Diagram of DAG with weights and arrows](image)

### Figure 1: DAG with Weights and Arrows

- (a) Initial DAG
- (b) DAG after instrumenting chord edges
- (c) DAG after moving weights from non-chord edges to chord edges
• **Needed instrumentation:**
  - Initialize path register \( (r = 0) \) at ENTRY
  - Increment register on instrumented edges \( (r += \text{Inc}(e)) \)
  - Record path's counter at EXIT \( (\text{count}[r]++) \)

• **Can optimize:**
  - Chord edge \( e \) can initialize counter \( (r = \text{Inc}(e)) \) iff first chord edge on every path from ENTRY to EXIT containing \( e \)
  - Chord edge \( e \) may increment path register and memory counter \( (\text{count}[r+\text{Inc}(e)]++) \) iff last chord edge on every path from ENTRY to EXIT containing \( e \)
Instrumentation

```
// if requires regeneration
instrumentation
compute

The unexpected code
must be executed.
```

```
R
E
F

D
1
2
4
B

A
0

E

\text{count}[r+1]++
\text{count}[r]++

D

\text{count}[r]++

A
r=0

B
r=2

C
r=4

D

E

F

\begin{itemize}
\item Section 4.3
\item Example
\item Figures
\end{itemize}

\section*{Technical Report}

\begin{itemize}
\item the DAG of a loop
\item the worst case
\item the minimal paths
\item the computation
\item to the exact number
\item the remainder
\end{itemize}

\end{document}
Details, details, details

- Algorithm works for DAGs
- Need to transform programs to be DAG like (and profile on DAG sub-graphs of cyclic graph)
Path Profiling
Path Profiling

<table>
<thead>
<tr>
<th>Path ID</th>
<th>Frequency</th>
<th>Condition</th>
<th>Call Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:0</td>
<td>1</td>
<td>t == 1</td>
<td>[9]</td>
</tr>
<tr>
<td>main:19</td>
<td>1</td>
<td>t==2 &amp;&amp; t &gt;= _</td>
<td>[1,3,4]</td>
</tr>
<tr>
<td>main:22</td>
<td>1</td>
<td>t==2 &amp;&amp; t &lt; _ &amp;&amp; _ &gt;= 13</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td>main:23</td>
<td>10</td>
<td>t==2 &amp;&amp; t &lt; _ &amp;&amp; _ &lt; 13</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>main:9</td>
<td>11</td>
<td>t &gt;= 3 &amp;&amp; t &gt;= _</td>
<td>[3]</td>
</tr>
<tr>
<td>main:13</td>
<td>55</td>
<td>t &gt;= 3 &amp;&amp; t &lt; _</td>
<td>[2,3]</td>
</tr>
<tr>
<td>main:2</td>
<td>114</td>
<td>t == 0 &amp;&amp; *a == '/'</td>
<td>no call lines</td>
</tr>
<tr>
<td>main:3</td>
<td>114</td>
<td>t &lt; -72</td>
<td>[5]</td>
</tr>
<tr>
<td>main:1</td>
<td>2358</td>
<td>t == 0 &amp;&amp; *a != '/'</td>
<td>[10]</td>
</tr>
<tr>
<td>main:7</td>
<td>2358</td>
<td>t &gt; -72 &amp;&amp; t &lt; -50 &amp;&amp; _ == *a</td>
<td>[6]</td>
</tr>
<tr>
<td>main:4</td>
<td>24931</td>
<td>t &lt; 0 &amp;&amp; t &lt; -50 &amp;&amp; _ == *a</td>
<td>[8]</td>
</tr>
<tr>
<td>main:5</td>
<td>39652</td>
<td>t &gt; -72 &amp;&amp; t &lt; -50 &amp;&amp; _ != *a</td>
<td>[7]</td>
</tr>
</tbody>
</table>

12 days of Christmas (also 11, to catch "off-by-one" cases)
26 unique strings
66 occurrences of non-partridge-in-a-pear-tree presents
114 strings printed, and 2358 characters printed.

• With manual examination:
  • Path 0 (executed once) initializes the recursion with the call main(2,2,...).
  • Paths 19, 22, and 23 control printing of the 12 verses.
    • P19 first verse, P23 middle 10 verses, and P22 last verse.
  • Paths 9 and 13 control printing of non-partridge-gifts within verse. (Frequencies of P9 + P13 = 66)
  • Paths 2 and 3 responsible for printing out a string.
  • Paths 1 and 7 print out the characters in a string. (why two?)
  • Path 4 skips over n sub-strings in the large string, each sub-string terminated with '/'
  • Path 5 linearly scans the string that encodes the character translation
#include <stdio.h>
static char *strings = LARGE_STRING;  /* the original set of strings */
static char *translate = SMALL_STRING; /* the translation mapping */
#define FIRST_DAY 1
#define LAST_DAY 12

/* the original "indices" of the various strings */
enum {  ON_THE = 0, FIRST = -1, TWELFTH = -12, DAY_OF_CHRISTMAS = -13,
       TWELVE_DRUMMERS_DRUMMING = -14, PARTRIDGE_IN_A_PEAR_TREE = -25
       };

char* skip_n_strings(int n,char *s) { /* skip -n strings (separator is /), */
   if (n == 0) return s; /* where n is a negative value */
   if (*s=='/') return skip_n_strings(n+1,s+1);
   else return skip_n_strings(n,s+1);
}

/* find the character in the translation buffer
   matching c and output the translation */
void translate_and_put_char(char c, char *trans) {
   if (c == *trans) putchar(trans[31]);
   else translate_and_put_char(c,trans+1);
}
void output_chars(char *s) {
    if (*s == '/') return;
    translate_and_put_char(*s, translate);
    output_chars(s+1);
}

/* skip to the "n"th string and print it */
void print_string(int n) { output_chars(skip_n_strings(n, strings)); }

/* print the list of gifts */
void inner_loop(int count_day, int current_day) {
    if (count_day < current_day) inner_loop(count_day+1, current_day);
    print_string(PARTRIDGE_IN_A_PEAR_TREE+(count_day-1));
}

void outer_loop(int current_day) {
    print_string(ON_THE);       /* "On the " */
    print_string(-current_day);  /* ordinal, ranges from -1 to -12 */
    print_string(DAY_OF_CHristmas);   /* "day of Christmas ..." */
    inner_loop(FIRST_DAY, current_day); /* print the list of gifts */
    if (current_day < LAST_DAY)
        outer_loop(current_day+1);
}

void main() { outer_loop(FIRST_DAY); }
and now it's time for something completely different
Adversarial Memory for Detecting Destructive Data Races

- By Flanagan and Freund, PLDI 2010
- A dynamic analysis to find **data races** in concurrent programs
  - What's a data race?
Program Model

\[ \alpha \in Trace \quad ::= \quad Operation^* \]

\[ a, b \in Operation \quad ::= \quad rd(t, x, v) \mid wr(t, x, v) \]
\[ \quad \mid acq(t, m) \mid rel(t, m) \]
\[ \quad \mid fork(t, u) \mid join(t, u) \]

\[ s, t, u \in Tid \quad x, y \in Var \quad m \in Lock \quad v \in Value \]

- A multithreaded program has concurrently executing threads (each with thread identifier \( t \in Tid \))
- Each thread manipulates variables and locks
- A trace lists sequence of operations performed by threads
  - Ignores everything except read/writes to variables, lock operations, and fork/join.
Happens-before relation and races

The happens-before relation $<_\alpha$ for a trace $\alpha$ is the smallest transitive-closed relation over the operations\(^3\) in $\alpha$ such that the relation $a <_\alpha b$ holds whenever $a$ occurs before $b$ in $\alpha$ and one of the following holds:

- [Program order]: Both operations are by the same thread.
- [Locking order]: $a$ releases a lock that is later acquired by $b$.
- [Fork order]: $a$ is $fork(t, u)$ and $b$ is by thread $u$.
- [Join order]: $a$ is by thread $u$ and $b$ is $join(t, u)$.

- Two operations $a$ and $b$ are concurrent if neither $a <_\alpha b$ nor $b <_\alpha a$

- A trace has a race if there are two memory accesses to the same variable, at least one of them is a write operation, and the accesses are concurrent
Races are bad

- Often cause errors only on certain rare executions
  - Hard to reproduce and reason about
- Exacerbated by multi-core processors and relaxed memory models
- BUT many races are benign
  - E.g., approximate counters, optimistic protocols
- Lots of work on race detection
  - Static: can be difficult to reason about all possible interleaving
  - Dynamic: interleavings with races may be rare
- This work:
  - standard dynamic analysis to detect "racy" variables
  - Then try to produce an erroneous execution that exhibits the race and produces observable incorrect behavior (e.g., crash, uncaught exception, etc.)
Double-checked locking example

```java
1 class Point {
2     double x, y;
3     static Point p;
4
5     Point() { x = 1.0; y = 1.0; }
6
7     static Point get() {
8         Point t = p;
9         if (t != null) return t;
10        synchronized (Point.class) {
11            if (p==null) p = new Point();
12            return p;
13        }
14    }
15
16    static double slope() {
17        return get().y / get().x;
18    }
19
20    public static void main(String[] args) {
21        fork { System.out.println( slope() ); };
22        fork { System.out.println( slope() ); }
23    }
24}
```

- Relaxed memory model means that get().x could evaluate to zero.
  - (Thus, the race on p is destructive, i.e., non-benign)
- But most of the time, destructive behavior not exhibited
Adversarial Memory

- Exploits full flexibility of relaxed memory model to try and cause crashes
- Tool tracks memory and synchronization operations of execution, and keeps a **write buffer** recording history of writes to racy variables
- When a thread asks for a value, return older (but still legal) values whenever possible
Memory Models

- **Sequential Memory Model**: read operation $a = rd(t, x, v)$ in trace $\alpha$ may only return the value of the most recent write to that variable in $\alpha$
  - Intuitive but limits optimization by compiler, virtual machine, and hardware.

- **Happens-Before Memory Model**: read operation $a = rd(t, x, v)$ in trace $\alpha$ may return the value of any write operation $b = wr(u, x, v)$ provided:
  1. $b$ does not happen after $a$; and
  2. no intervening write $c$ to $x$ where $b <_\alpha c <_\alpha a$
• Consider
  \[ x := y \parallel y := x \]
• Assume \( x \) and \( y \) are initially zero.
• Under happens-before memory model, the following trace is possible:
  \[ rd(t_1, x, 42) \quad wr(t_1, y, 42) \quad rd(t_2, y, 42) \quad wr(t_2, x, 42) \]
• Where did 42 come from??!!?
• Java Memory Model extends the happens-before memory model with a causality requirement to preclude non-sensical traces as above
• This paper uses **Progressive Java Memory Model**: read operation \( a = rd(t, x, v) \) in trace \( \alpha \) may return the value of any write operation \( b = wr(u, x, v) \) provided:
  1. \( b \) is before \( a \) in trace \( \alpha \); and
  2. no intervening write \( c \) to \( x \) where \( b <_\alpha c <_\alpha a \)
Adversarial Memory Implementation

- Uses **vector clocks** to record time stamps of write operations
  - Vector clocks can be used to determine the happens-before relation
- Read operation for $x$ at time $C_t$ can return any value so long as it satisfies the Progressive Java Memory Model
  - i.e., a write in the write buffer for $x$ that happened at time $K_i$ such that there is no write at time $K_j$ where $K_i \subseteq K_j \subseteq C_t$
Adversarial Memory Heuristics

- **Sequentially consistent**: always return most recently written value
- **Oldest**: chose "most stale" value. (occasionally return most-recent value to satisfy fairness assumptions)
- **Oldest-but-different**: return oldest element that is different from the last value read
- **Random**: return a random value from the permitted values
- **Random-but-different**: return a random value from the permitted value that is different from the last value read
## Effectiveness

<table>
<thead>
<tr>
<th>Program</th>
<th>Field</th>
<th>Erroneous Behavior Observation Rate (%)</th>
<th>JUMBLE Configurations</th>
<th>Destructive Race?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No Jumble</td>
<td>JUMBLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequentially Consistent</td>
<td>Oldest</td>
<td>Oldest-But-Different</td>
</tr>
<tr>
<td>Figure 8</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Figure 2</td>
<td>p</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Figure 2</td>
<td>p.x</td>
<td>0</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>Figure 2</td>
<td>p.y</td>
<td>0</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>jbb</td>
<td>Company.elapsed_time</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>jbb</td>
<td>Company.mode</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>montecarlo</td>
<td>Universal.UNIVERSAL_DEBUG</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mtrt</td>
<td>RayTracer.threadCount</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>JGFRayTracerBench.checksum1</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>tsp</td>
<td>TspSolver.MinTourLen</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>sor</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>lufact</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>moldyn</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>
### Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (lines)</th>
<th>Num. Threads</th>
<th>Field</th>
<th>Base Time (s)</th>
<th>Slowdown</th>
<th>Num. Instances</th>
<th>Num. Writes</th>
<th>Max. Buffer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>jbb</td>
<td>30,491</td>
<td>5</td>
<td>Company.elapsed_time</td>
<td>74.4</td>
<td>1.3</td>
<td>1.3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>jbb</td>
<td>30,491</td>
<td>5</td>
<td>Company.mode</td>
<td>74.4</td>
<td>1.3</td>
<td>1.4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,669</td>
<td>4</td>
<td>Universal.UNIVERSAL_DEBUG</td>
<td>1.6</td>
<td>1.2</td>
<td>1.2</td>
<td>1</td>
<td>40,005</td>
</tr>
<tr>
<td>mtrt</td>
<td>11,317</td>
<td>5</td>
<td>RayTracer.threadCount</td>
<td>0.5</td>
<td>4.5</td>
<td>4.9</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,970</td>
<td>4</td>
<td>JGFRayTracerBench.checksum</td>
<td>5.6</td>
<td>1.1</td>
<td>1.1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>tsp</td>
<td>742</td>
<td>5</td>
<td>TspSolver.MinTourLen</td>
<td>0.7</td>
<td>2.3</td>
<td>4.0</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>sor</td>
<td>883</td>
<td>4</td>
<td>array index [0] and [1]</td>
<td>0.6</td>
<td>3.9</td>
<td>5.8</td>
<td>2,106</td>
<td>104,620</td>
</tr>
<tr>
<td>lufact</td>
<td>1,627</td>
<td>4</td>
<td>array index [0] and [1]</td>
<td>0.4</td>
<td>4.1</td>
<td>4.2</td>
<td>1,108</td>
<td>14,526</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,407</td>
<td>4</td>
<td>array index [0] and [1]</td>
<td>0.9</td>
<td>4.1</td>
<td>8.9</td>
<td>62</td>
<td>53,433</td>
</tr>
</tbody>
</table>

**Figure 10.** Performance of JUMBLE under the *Sequentially-Consistent* configuration.