Dynamic Bug Detection for Managed Languages

Kathryn S McKinley
The University of Texas at Austin
IBM Academic Visitor
Software Developer Dreams
Software Developer Dreams

• Easy to implement specifications
• Easy to get correct
• Robust to errors of all sorts
• Easy to maintain
• Runs fast
Reality
thanks to David Callahan
Software Developer Dreams
people-who-want-computers-to-do-things

• Easy to implement specifications
• Easy to get correct
• Robust to errors of all sorts

• Easy to maintain
• Runs fast
Software Developer Dreams

- Easy to implement specifications
- Easy to get correct learn & get results
- Robust to errors of all sorts

- Easy to maintain
- Runs fast
Software Developer Dreams
people-who-want-computers-to-do-things

• Easy to implement specifications
• Easy to get correct learn & get results
• Robust to errors of all sorts

Operating Environment Survives
No unrecoverable data loss
No observable implementation artifacts
No violations of security or privacy policies

• Easy to maintain
• Runs fast
Software Developer Dreams
people-who-want-computers-to-do-things

• Easy to implement specifications
• Easy to get correct learn & get results
• Robust to errors of all sorts

Operating Environment Survives
No unrecoverable data loss
No observable implementation artifacts
No violations of security or privacy policies

• Easy to maintain share
• Runs fast enough and consistently
Impediments to Reality & the Dream

Applications are more complicated
- In 10 years, Windows grew from 10M to 50M lines
- Application knowledge spread thin
- Testing is not enough

Architectures are more complicated
- Transistors on a chip [Moore’s law]:
  - 180,000 transistors - First CMOS Vax 1987
  - 291,000,000 transistors - Core2 Duo 2007
- Many core

The glue is more complicated
- Dynamic optimization & runtime systems
- Coordination of multiple languages and runtime systems
- Impossible to test all scenarios

Difficult to understand programs or runtime behavior
Multi-pronged Approach to Correct High Performance Software

- **Better languages**
  - Java and C# are not the last programming languages

- **Validation when possible**
  - We probably will not be able to validate substantial parallel applications any time soon
  - Is application growth outpacing validation advances?

- **Analysis and development tools**
  - Static bug finding tools
  - Dynamic bug finding tools

- **Self healing & dynamically updatable systems**
  - Don’t crash

- **Run Fast**
  - New optimization opportunities
    - move code & data around
    - use JIT to specialize to input data

- **Evaluation**
Detect, Report, & Tolerate Bugs

- **Memory errors**
  - find leaks [Bond ASPLOS 2006, Jump POPL 2007]
  - tolerate leaks [Bond in progress]

- **Null pointer exceptions**
  - store a PC canary instead of zero for null [Bond et al. OOPSLA 2007]
  - keep running [Kent/Bond, in progress]

- **Untested control paths & security holes**
  - low cost probabilistic calling context [Bond OOPSLA 2007]

- **Random semantic errors**
  - data structure repair [Elkahriablieh et al. OOPLSA 2007]
  - heap analysis [Jump in progress]
  - Keep running [Kent/Bond in progress]

- **Dynamic software updating**
  - update software while it is running [Subramanian in progress]
A memory leak in a garbage-collected language occurs when a program inadvertently maintains references to objects that it no longer needs, preventing the collector from reclaiming space.

- **Best case**: increases GC workload
- **Worst case**: systematic heap growth causes crash after days of execution

**Cork** accurately pinpoints systematic heap growth during production runs
Cork’s Solution: Find growing data structures

1. Summarize heap objects by type & reference relationships
   • Calculate *type points-from graph*
   • Piggyback on full-heap object scan
2. Difference the graphs
3. Generate debugging reports
   • Candidate Report
   • Data Structure Slice Report
   • Allocation Site Report
Type Points-From Graph

- Dark blue = HashTable
- Light green = Queue
- Orange = Queue
- Light blue = Company
- Yellow = People

Heap

TPFG

1 - 3
3 - 4
4 - 1
1 - 2

= instance
= type
Difference TPFGs

$TPFG_i$
Difference TPFGs

\[ TPFG_i \]

\[ TPFG_{i+1} \]
Difference TPFGs

$TPFG_i$

$TPFG_{i+1}$
Difference TPFGs

\[ TPFG_i \]

\[ TPFG_{i+1} \]

\[ TPFG_{i+2} \]
Interpreting TPFG

- Find nodes of types $t$ that grow
  \[ V_{t(i)} > (1 - f) \times V_{t(i-1)} \]
  - $i$ is the phase
  - $f$ is the decay factor e.g., .05
- Rank nodes and edges
  \[ r_i = r_{i-1} \pm p_i \times (Q - 1) \]
  - $p$ is the number of phases that the type grows or shrinks,
    add to the rank if it grows, subtract if it shrinks
  - $Q$ is a ratio $> 1$ of $V_i$ to $V_{i-1}$
- Designate node as a candidate if $r_{t(i)} > R_{threshold}$
- Calculate data structure slice from each candidate
  - Set of all paths $n_0...n_n$ such that $r_{n(k+1)} \cup r_{n(k)} > 0$
  - Identifies the growing data structure
Implementation and Methodology

- Jikes RVM with MMTk
- Benchmarks:
  - SPECjvm98, SPECjbb2000, DaCapo
  - Eclipse 3.1.2
- Garbage collector
  - Generational with 4MB bounded nursery
- For performance, report application & GC time (not JIT)
  - Replay compilation
  - 2nd run methodology
Efficiency and Scalability

• Node/type data stored in type information block (TIB) adding 5 words
  - 1 word for type volume and edge list pointer for each of the previous 4 collections
  - 1 word for # of phases (ρ)

• Edge data stored in lists
  - Prune parts of TPFG that are non-growing
  - # of edges are linear in practice
# Space Overhead

<table>
<thead>
<tr>
<th></th>
<th>jess</th>
<th>Eclipse</th>
<th>Geomean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># of types</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bm+VM</td>
<td>1744</td>
<td>3365</td>
<td>1747</td>
</tr>
<tr>
<td>TPFG avg</td>
<td>318</td>
<td>667</td>
<td>334</td>
</tr>
<tr>
<td>TPFG max</td>
<td>319</td>
<td>775</td>
<td>346</td>
</tr>
<tr>
<td><strong># of edges</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPFG avg</td>
<td>844</td>
<td>4090</td>
<td>904</td>
</tr>
<tr>
<td>TPFG max</td>
<td>861</td>
<td>7585</td>
<td>1142</td>
</tr>
<tr>
<td>% pruned</td>
<td>66%</td>
<td>42%</td>
<td>59.8%</td>
</tr>
<tr>
<td>Increased Heap</td>
<td>0.094%</td>
<td>0.167%</td>
<td>0.233%</td>
</tr>
</tbody>
</table>
Time Overhead

![Graph showing normalized total time over heap size relative to minimum. The graph has a y-axis labeled 'Normalized Total Time' ranging from 0.9 to 1.3 and an x-axis labeled 'Heap size relative to minimum' ranging from 3 to 6. There are data points marked with an 'x' and a trend line labeled 'GenMS'.]
Benchmarks with Cork

- Identified systematic heap growth in popular benchmarks
  - jess, SPECjbb2000, fop, eclipse
- Accurately reported types growing and the data structure containing them
- We quickly found and fixed memory leaks in 2 SPECjvm benchmarks
Eclipse 3.1.2 with Cork

- IDE
- Big, complex, and open-source
- Bug repository details memory leaks and how to reproduce them
  - #115789: Memory Leak
  - Comparing 2 source trees or jar files
  - Manually repeat while running Cork
Eclipse Slice:
out of 775 nodes in the TPFG
Eclipse 115789

- Before (74.9 K/M)
- After (4.75 K/M)
- Fixed (no growth)
Related Work

• Leakbot [ECOOP 2003]
  - Summarization by data structure
    • Heuristics for pruning the graph
  - Uses heap snap shots and analyzes them in a separate thread

• C tools [Hauswirth & Chilimibi]
  - high memory overheads
Tolerating Memory Leaks

[Mike Bond: work in progress]

• Problem:
  - Identified leaks take time to fix
  - In the mean time, they degrade application and GC performance, or crash the program

• Solution:
  - Managed languages provide an opportunity to reorganize memory
Tolerating Memory Leaks

Roots

In-Use Space

Stale Space

A → B → C
E → F → G → D
Tolerating Memory Leaks

In-Use Space

Roots

A → B → C

E → F → G → D

Stale Space

(1) Identify stale memory
Tolerating Memory Leaks

- Roots
- In-Use Space
  - A
  - B
  - C
  - E
- Stale Space
  - B_{scion}
  - B_{stub}
  - D
  - G
  - F

(2) Move stale memory out of application & collector working set
Tolerating Memory Leaks

In-Use Space

- A
- B
- C
- E

Stale Space

- D
- G
- F
- B_{stub}
- B_{scion}

(3) OS pages stale space to disk & compresses it
Tolerating Memory Leaks

In-Use Space

E

A → B → C

B_{scion}

Stale Space

D

B_{stub}

G

F

(4) Activate object if touched

Roots
Tolerating Memory Leaks

(4) Activate object if touched
Tolerating Memory Leaks

In-Use Space

Roots

E
B
C

F

Illusion of fixed leak!

Stale Space

D
G

B_{scion}
B_{stub}
Throughput of Eclipse Leak

![Graph showing the time (ms) on the y-axis and iteration on the x-axis, with three lines representing Jikes RVM, Sun JVM, and Melt.]
Multi-pronged Approach to Correct High Performance Software

• Better languages
  - Java and C# are not the last programming languages
• Validation when possible
  - We probably will not be able to validate substantial parallel applications any time soon
  - Is application growth outpacing validation advances?
• Analysis and development tools
  - Static bug finding tools
  - Dynamic bug finding tools
• Self healing & dynamically updatable systems
  - Don’t crash
• Run Fast
  - New optimization opportunities
    • move code & data around
    • use JIT to specialize to input data
• Evaluation
Thank you!

www.dacapogroup.org
<table>
<thead>
<tr>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution efficiency</td>
<td>Developer productivity</td>
</tr>
<tr>
<td>Trusts the programmer</td>
<td>Protects the programmer</td>
</tr>
<tr>
<td>Arbitrary memory access possible</td>
<td>Memory access only through objects</td>
</tr>
<tr>
<td>Can arbitrarily override types</td>
<td>Type safety</td>
</tr>
<tr>
<td>Procedural or object-oriented</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>Meaning of operators immutable</td>
</tr>
<tr>
<td>Powerful capabilities of language</td>
<td>Feature-rich, easy-to-use standard library</td>
</tr>
<tr>
<td>Explicit memory control</td>
<td>Automatic memory management</td>
</tr>
</tbody>
</table>

[Wikipedia: Comparison of Java and C++, Dec 2006]
Panacea for Bugs?

• PMD, FindBugs, JLint, …
• ESC/Java, Bandera, …
• HPROF, JProbe, HAT, Leakbot, …

Microsoft reports that in C#, 75% of development time is spent in debugging

• Provide a good start
• Programs still ship with memory and semantic errors