a way to track data structure evolution, cheaply and automatically

Nick Mitchell
nickm@us.ibm.com

IBM TJ Watson Research Center
Hawthorne, New York, USA

December 4, 2003
our three constraints
(framework-intensive applications are fun!)

• as automated as possible
  – no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  – application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  – sandwiching effects
    (the framework is the driver, your application is just along for the ride)

• scale to gigantic heaps
  – e.g. 40 million objects on a laptop, analyzed in a few minutes

• impose minimal perturbation
  – time and space perturbation on the server must be in the noise
The common datatypes don't help diagnose problems with structure evolution.

<table>
<thead>
<tr>
<th>Class Path</th>
<th>Live Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>java/lang/String</td>
<td>230025</td>
</tr>
<tr>
<td>com/ibm/servlet/util/HashtableEntry</td>
<td>92825</td>
</tr>
<tr>
<td>java/util/Hashtable$Entry</td>
<td>59727</td>
</tr>
<tr>
<td>org/apache/xerces/dom/TextImpl</td>
<td>15627</td>
</tr>
<tr>
<td>org/apache/xerces/dom/AttrImpl</td>
<td>11278</td>
</tr>
<tr>
<td>org/apache/xalan/xpath/xml/StringToStdStringTable</td>
<td>11204</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>org/apache/xerces/dom/DocumentImpl</td>
<td>52</td>
</tr>
</tbody>
</table>
the big data structures are... big, not bugs

<table>
<thead>
<tr>
<th>Class Path</th>
<th># constituents</th>
</tr>
</thead>
<tbody>
<tr>
<td>com/ibm/servlet/DynamicClassLoader</td>
<td>82882</td>
</tr>
<tr>
<td>com/ibm/servlet/DynamicClassLoader</td>
<td>73537</td>
</tr>
<tr>
<td>com/.../XSLTransform</td>
<td>71628</td>
</tr>
<tr>
<td>com/.../PropertiesFactory</td>
<td>66957</td>
</tr>
<tr>
<td>elements of Finalizer queue</td>
<td>39886</td>
</tr>
<tr>
<td>org/apache/xalan/xslt/TemplateList</td>
<td>28969</td>
</tr>
<tr>
<td>owned by native code</td>
<td>18829</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

if we wait long enough, then the leaking data structure will float to the top; otherwise, noise effects dominate
our three constraints
(framework-intensive applications are fun!)

• as automated as possible
  - no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  - application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  - sandwiching effects
    (the framework is the driver, your application is just along for the ride)

• scale to gigantic heaps
  - e.g. 40 million objects on a laptop, analyzed in a few minutes

• impose minimal perturbation
  - time and space perturbation on the server must be in the noise
object reference graphs are getting very large

250 thousand in 1/2002

14 million in 11/2003
our three constraints
(framework-intensive applications are fun!)

- as automated as possible
  - no shallow patterns identify the bug
    (not dominant type or allocation site or biggest data structure, not even diff'd over time)
  - application-level memory management
    (caches, pools, lazy pools, lazy or asynchronous deallocation policies)
  - sandwiching effects
    (the framework is the driver, your application is just along for the ride)

- scale to gigantic heaps
  - e.g. 40 million objects on a laptop, analyzed in a few minutes

- impose minimal perturbation
  - time and space perturbation on the server must be in the noise
categories of evolution

the whole reference graph
categories of region evolution

a region is a subset of objects equivalent in the way they evolve
categories of region evolution

XML documents
categories of region evolution

growing without bound (a leak)
categories of region evolution

- constituency doesn't change (a pool)
- constituency is in flux (hot cache)
- growing without bound (a leak)
- transient (transaction-local)
- shrinking (cold cache)
regions as equivalence classes

A region is a subset of objects equivalent in the way they evolve.
when a region grows...
when a region grows...

doh! leaked an XML document
when a region grows...

doh! leaked an XML document

currently on the fringe
(i.e. a newbie pointed to by an oldie)
when a region grows...
we can observe a **fringe**

currently *on the fringe*
for each region, we can also infer a **historic fringe**

currently on the fringe

at some point in the past, on the fringe

XML

XML

XML
finally, verify that the region evolves as expected currently on the fringe

at some point in the past, on the fringe

imetype) in the future, on the fringe
three tasks of evolution analysis

observe the present

infer the past

validate using future info
a way to diagnose heap evolution, in production

- **collect** a few heap snapshots
- **observe** what *is* on the fringe
  - yields a set of “seed” region keys
- **infer** what *was* on the fringe
  - yields regions populated based on region key equality
- **validate by adaptive tracing**
  - generate a set of *change detectors* that monitor violations or confirmations of a region's category of evolution
  - periodically execute a detector to refine category, set of change detectors, and quantification of how evolution is progressing
how do we implement all that?

• what's a region key?
  – a tuple of features that summarize that region's evolution
  – each object gets a key, set of canonical keys is the set of regions

• can we avoid presenting, tracking every region?
  – yup! a mixture model reduces from millions of regions to a handful

• what's a region change detector?
  – a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

• can we avoid modeling every region?
  – use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
how do we implement all that?

• what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions

• can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful

• what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

• can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
how do we implement all that?

• what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions

• can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful

• what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

• can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
how do we implement all that?

• what's a region key?
  – a tuple of features that summarize that region's evolution
  – each object gets a key, set of canonical keys is the set of regions

• can we avoid presenting, tracking every region?
  – yup! a mixture model reduces from millions of regions to a handful

• what's a region change detector?
  – a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

• can we avoid modeling every region?
  – use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
when are two objects in the same region?

recall that we're leaking XML documents
(and so these two dudes are on the historic fringe)
when are two objects in the same region?

all objects “below” the historic fringe have equal region keys
(each DocumentImpl is a proxy for its dominated evolution)

\[ k_1 = k_2 = k_3 = k_4 = \tilde{k} \]
these two objects have different keys

\[ \text{region} = (\text{leak}, \tilde{k}) \]

\[ k_5 \neq \tilde{k} \]
feature #1:
 historic fringe datatype
these two objects also have different keys

same fringe type, but different data structure

$k_5 \neq \tilde{k}$
feature #2: root data structure
these two objects also have different keys

same owner, same fringe type, but different path between them

\[ k_5 \neq \tilde{k} \]
feature #3:
owning container

Diagram showing relationships between `XSLTransform`, `XSLTEngineImpl`, `Vector`, `Object`, `Stack`, and `DocumentImpl`.
to each object, a region key tuple

\[ \tilde{k} = [L, O, \text{typeof}(C)] \]
how do we implement all that?

- what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions

- can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful

- what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

- can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
how do we avoid presenting and tracking every region?

Define leak root metric, $\text{LRM}=B \cdot M \cdot G$, such that each leaking region has one $o$ with $\text{LRM}(o) > 0$, and few $o$'s have $\text{LRM}(o) > \Theta$.

- **B**: eight binary rules to rule out impossible
  - (be Sherleak Holmes!)
  - narrow from a million to a hundred

- **M**: mixture model to rank the remaining
  - narrow from a hundred to tens

- **G**: global fixpoint to ensure uniqueness
  - narrow from tens to a handful of highly-ranked leak roots
### B: ruling out the impossible
(using structural information)

#### A. objects pointing to nothing aren't very interesting

#### B. arrays themselves don't leak
(but their dominating containers might)

#### C. ibid for objects not at the head of a single-entry region

#### D. objects which don't uniquely own anything also aren't interesting

<table>
<thead>
<tr>
<th># objects</th>
<th>fraction of objects remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>phone company</td>
<td>267,956</td>
</tr>
<tr>
<td>IDE</td>
<td>350,136</td>
</tr>
<tr>
<td>brokerage1</td>
<td>838,912</td>
</tr>
<tr>
<td>brokerage2</td>
<td>1,015,112</td>
</tr>
<tr>
<td>credit bureau</td>
<td>1,320,953</td>
</tr>
</tbody>
</table>
B: ruling out the impossible
(using temporal information)

<table>
<thead>
<tr>
<th></th>
<th># objects</th>
<th>-structural</th>
<th>-E</th>
<th>-E-F</th>
<th>-E-F-G</th>
<th>all told</th>
</tr>
</thead>
<tbody>
<tr>
<td>phone company</td>
<td>267,956</td>
<td>16,346</td>
<td>73</td>
<td>73</td>
<td>72</td>
<td>29</td>
</tr>
<tr>
<td>IDE</td>
<td>350,136</td>
<td>25,653</td>
<td>99</td>
<td>99</td>
<td>29</td>
<td>10</td>
</tr>
<tr>
<td>brokerage1</td>
<td>838,912</td>
<td>26,291</td>
<td>97</td>
<td>82</td>
<td>64</td>
<td>67</td>
</tr>
<tr>
<td>brokerage2</td>
<td>1,015,112</td>
<td>12,020</td>
<td>102</td>
<td>102</td>
<td>64</td>
<td>17</td>
</tr>
<tr>
<td>credit bureau</td>
<td>1,320,953</td>
<td>160,900</td>
<td>579</td>
<td>519</td>
<td>518</td>
<td>242</td>
</tr>
</tbody>
</table>

**E.** ignore structures that contain only old or only new objects
(e.g. an already-primed pool)

**G.** ignore structures with no overlap in datatypes over time

**F.** structures that contain only new arrays are boring
(there's nothing new in those arrays)

**H.** structures that contain no objects on the fringe are safe to ignore
LRM=B◦M◦G, for example
(before applying the bug fixes)

<table>
<thead>
<tr>
<th></th>
<th># constituents</th>
<th>size rank</th>
<th>LRM(o)</th>
</tr>
</thead>
<tbody>
<tr>
<td>com/.../EventNotifier</td>
<td>377276</td>
<td>1</td>
<td>0.895</td>
</tr>
<tr>
<td>com/.../FormProperties</td>
<td>270</td>
<td>157</td>
<td>0.658</td>
</tr>
<tr>
<td>com/.../XslTemplateCollection</td>
<td>32</td>
<td>841</td>
<td>0.463</td>
</tr>
<tr>
<td>com/.../VerifySignonScenario</td>
<td>18</td>
<td>1050</td>
<td>0.420</td>
</tr>
</tbody>
</table>

of the highest-ranked candidate roots,
the top two indeed leak
(from 1,015,112 live objects)
LRM = B ⊙ M ⊙ G, for example (after applying the bug fixes)

<table>
<thead>
<tr>
<th></th>
<th># constituents</th>
<th>size rank</th>
<th>LRM(o)</th>
</tr>
</thead>
<tbody>
<tr>
<td>com/websphere/AlarmThread</td>
<td>399</td>
<td>130</td>
<td>0.322</td>
</tr>
<tr>
<td>com/.../ContextModel</td>
<td>837</td>
<td>86</td>
<td>0.266</td>
</tr>
<tr>
<td>com/websphere/PoolManager</td>
<td>391</td>
<td>134</td>
<td>0.260</td>
</tr>
<tr>
<td>com/websphere/PoolEpm</td>
<td>385</td>
<td>137</td>
<td>0.254</td>
</tr>
</tbody>
</table>

after fixing the leak, there are no stand-out candidates (from 779,540 live objects)
how do we implement all that?

• what's a region key?
  - a tuple of features that summarize that region's evolution
  - each object gets a key, set of canonical keys is the set of regions

• can we avoid presenting, tracking every region?
  - yup! a mixture model reduces from millions of regions to a handful

• what's a region change detector?
  - a short path traversal of the program's running heap that sees if additions, removals, or internal relinking of a region has occurred

• can we avoid modeling every region?
  - use the historic fringe to identify and model only the subsets of the reference graph likely to evolve in ways the analysis cares about
detecting evolution cheaply

• a region evolves when elements are
  - added to
  - removed from
  - relinked within

• track evolution with region change detectors
detecting evolution cheaply

• a region evolves when elements are
  - added to
  - removed from
  - relinked within

• track evolution with **region change detectors**

• a detector is a tuple \([R, H, T, B, P, M]\)
  - \(R\): region to detect changes in
  - \(H, T\): the head and tail of a short, bounded-size traversal
  - \(B\): a sample bias
  - \(P\): a match precondition
  - \(M\): a mutator, updates the set of existing detectors
leakbot in action

just after initial analysis

about one minute later

and another few minutes...

over time...

a non-leaking region

is downgraded
final stuff

- analysis handles 40 million objects with 600M
- adaptive, online tracing slows app down only 2%
- can identify very slow leaks in a few minutes
- implemented as a JVMPI agent (written in C++) and an analyzer (written in Java)
- going into WebSphere and Rational Studio
final stuff

- analysis handles 40 million objects with 600M
- adaptive, online tracing slows app down only 2%
- can identify very slow leaks in a few minutes
- implemented as a JVMPI agent (written in C++) and an analyzer (written in Java)
- going into WebSphere and Rational Studio

- thanks to the team! Bowen Alpern, Glenn Ammons, Vas Bala, Herb Derby, Todd Mummert, Darrell Reimer, Gary Sevitsky, Edith Schonberg, Harini Srinivasan, Kavitha Srinivas
  - JIT/BCI interface for efficient bytecode-level probing (going into J9)
  - rules-based validation system (going into Rational Studio)
  - automated performance analysis (ongoing)
factoring out objects via heap differencing is insufficient

<table>
<thead>
<tr>
<th>Class</th>
<th>&quot;new&quot; live instances</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java/lang/String</code></td>
<td>9444</td>
</tr>
<tr>
<td><code>org/apache/xerces/dom/TextImpl</code></td>
<td>6810</td>
</tr>
<tr>
<td><code>org/apache/xerces/dom/AttrImpl</code></td>
<td>5290</td>
</tr>
<tr>
<td><code>java/util/Hashtable$Entry</code></td>
<td>3244</td>
</tr>
<tr>
<td><code>org/apache/xerces/dom/NamedNodeMapImpl</code></td>
<td>2713</td>
</tr>
<tr>
<td><code>org/apache/xerces/dom/ElementImpl</code></td>
<td>2123</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>org/apache/xerces/dom/DocumentImpl</code></td>
<td>27</td>
</tr>
</tbody>
</table>
an atom of a leak
(every leaking operation leaks lots of these objects)

you're leaking Strings
a bowl leaks
(every leaking operation leaks one of these data structures)

(a bowl of spaghetti)

(a whole leaking bowl of spaghetti)

(DocumentImpl)

(AttrImpl)

(Vector)

(ElementImpl)

(TextImpl)

(NamedNodeMapImpl)

(AttrImpl)

(AttrImpl)

(arrays)

(String)
leakbot and its loops

snapshot the heap

reconstruct regions

rank regions

generate region change detectors

dotimes(N) {
    d = next detector
    if(d.B > rand()) {
        traverse from d.H to d.T
        if(d.P(d.T)) {
            d.M()
        }
    }
}

sleep
Strategies for Dissecting Leaks
(and some problems with each)

- **histogram by datatype**
  - Strings are in every data structure

- **histogram by allocation site**
  - Strings are allocated everywhere
  - expensive (c.f. HPROF's 5-10x slowdown)

- **visualize reference graph**
  - an application doesn't just leak objects, it leaks entire (and entirely ugly) data structures
  - c.f. Jinsight, JProbe, Purify
Summary of the LeakBot Technique

- **structure live objects into Co-evolving Regions**
  - portions of data structures which change in similar ways

- **rank regions** according to likelihood of problem
  - only present to user those regions likely to leak, the suspects
  - e.g. of Schwab's 1M live objects, leakbot identifies three suspects

- **track evolution** of regions as program runs
  - treat structuring and ranking as initial estimates
    - e.g. we might have caught a pool being populated – it'll eventually plateau
  - from them, derive a scheme for very lightweight probing
  - verify whether initial estimates correct, and update ranking
M: the mixture model

- no single property is entirely indicative
- instead, use gated mixture of them all

<table>
<thead>
<tr>
<th>EventNotifier</th>
<th>377,276</th>
<th>34%</th>
<th>0</th>
<th>44</th>
<th>33%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadDiscriminator</td>
<td>274,433</td>
<td>2%</td>
<td>455</td>
<td>52</td>
<td>13%</td>
</tr>
<tr>
<td>FormProperties</td>
<td>270</td>
<td>97%</td>
<td>50</td>
<td>3</td>
<td>100%</td>
</tr>
<tr>
<td>XslTemplateCache</td>
<td>32</td>
<td>90%</td>
<td>0</td>
<td>1</td>
<td>40%</td>
</tr>
<tr>
<td>VerifySignonScenario</td>
<td>18</td>
<td>11%</td>
<td>1</td>
<td>1</td>
<td>50%</td>
</tr>
</tbody>
</table>

this application had two leaks