

Go with the Flow: Profiling Copies to Find Run-time Bloat

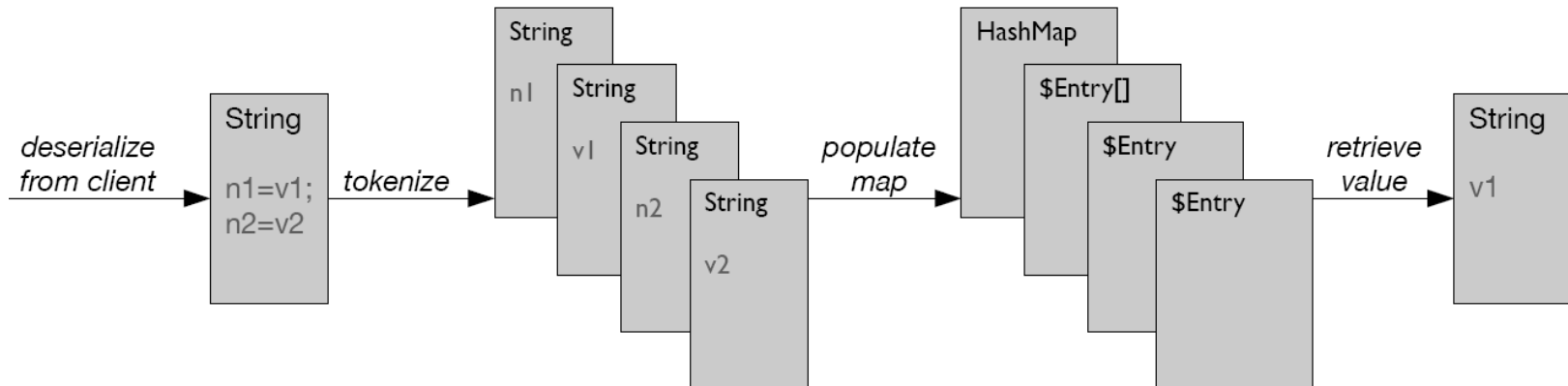
**Guoqing Xu, Matthew Arnold, Nick Mitchell,
Atanas Rountev, Gary Sevitsky
Ohio State University
IBM T. J. Watson Research Center**

Motivation

- Higher-level languages such as Java
 - Large libraries
 - Framework-intensive applications
- Consequences
 - Massive amounts of wasteful computation
 - Performance problems that current JIT compilers cannot solve (**grand challenge**)
- Runtime bloat
 - Performance problems caused by inefficiency
 - time (slow) and space (memory hog)

Bloat Example

- A commercial document server
 - Run on top of IBM WebSphere application server
 - **25000** method invocations and **3000** temporary objects to insert a simple document
- Example of bloat
 - Decodes the whole cookie in an **8-element** hash map, but extracts only **1 or 2** elements
 - **1000** method calls



Optimization Challenges

- Many large applications are free of hot-spots
 - For the document server, no single method contributes more than **3.19%** of total application time
 - Applications are suffering from *systemic* runtime bloat
- The current JITs do not optimize bloat away
 - The JIT cannot perform expensive analysis
 - The JIT does not understand the semantics
 - The JIT cannot identify “useless” operations
- Our goal: *assistance with hand-tuning*

Copying as Indicator of Bloat

- **Data copying** is an excellent indicator of bloat
 - Chains of data flow that carry values from one storage location to another, often via temporary objects
 - No values are changed
 - The same data is contained in many locations: both wasted space and wasteful operations
- Identifying data copying
 - Profile data copying for postmortem diagnosis
 - Data-based metric

Copy Profiling

- A data copy
 - Load-store pair without computation in between
 - In particular a heap load - heap store pair
 - Example: ***int a = b.f; int c = a; A.g = c;***
- Data copy profiles
 - Percent of instructions that are **copies**, compared to other activities (e.g., **computation** or **comparison**)
 - Observation: copy activity is often concentrated in a small number of methods
 - For the document server, the top 50 methods explain 82% of total copies, but only 24% of total running time

Copy Chain Profiling

- Copy chain profiling
 - Understanding the chains as a whole
- A copy chain is
 - A sequence of copies that carries a value through two or more heap locations
 - **Node**: a heap location (instance field and static field)
 - **Edge**: a sequence of copies that transfers a value from one heap location to another
 - An edge abstracts away intermediate copies via stack locations

An Example

```
class List{
    Object[] data; int count = 0;
    List(){ data = new Object[1000];
        // O3}
    void add(Object o)
    { data[count++] = o; }
    Object get(int i){ return data[i]; }

    List clone(){
        List newL = new List(); // O2
        for(int j = 0; j < count; j++)
            newL.add(get(j));
    }
}
```

```
static void main(){
    List l = new List(); // O1
    for(int i = 0; i < 1000; i++)
        l.add(new Integer(i)); // O4
    List l2 = l.clone();
    for(int i = 0; i < 1000; i++) {
        System.out.println
            (l2.get(i));
    }
}
```

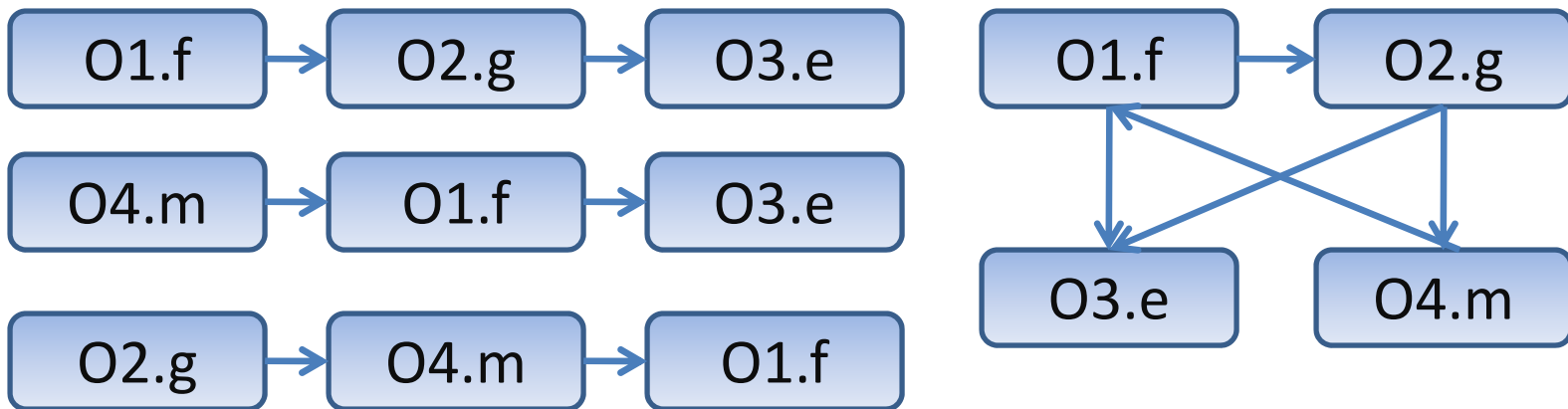
O3.ELEM → O3.ELEM

Chain Augmentation

- We augment a chain with
 - A producer node (source of data)
 - A constant value
 - A new expression
 - A computation instruction creating new data
 - A consumer node **C** (sink of data)
 - Has only one instance **C**
 - Shows that the data goes to a computation instruction or a native method
- Example
 - O4 → O3.ELEM → O3.ELEM → C

Copy Graph

- It can be prohibitively expensive to profile copy chains
 - Both time and space expensive
- Abstractions
 - Static abstractions of objects
 - Profile copy chain **edges only**: copy graph



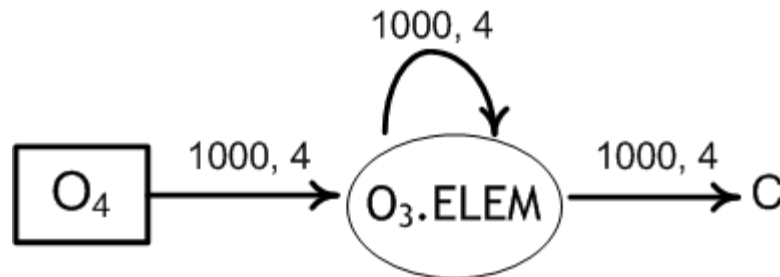
Copy Graph Profiling

- Nodes
 - Allocation site nodes “ O_i ”
 - Instance field nodes “ $O_i.f$ ”
 - Static field nodes “ $A.f$ ”
 - Consumer node “ C ”
- Edges annotated with two integers
 - Frequency and the number of copied bytes (1, 2, 4, 8)

- Example

- Copy chain: $O_4 \rightarrow O_3.ELEM \rightarrow O_3.ELEM \rightarrow C$

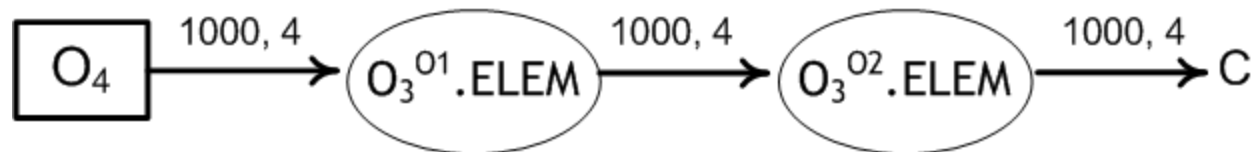
- Copy graph:



Context Sensitive Copy Graph

- Imprecision from context insensitive copy graph
 - Invalid copy chains may be derived due to nodes merging
- Context sensitivity
 - *k-Object sensitivity* (Milanova-ISSTA 02)
- 1-object-sensitive naming scheme
 - An object is named as *its allocation site* + the *allocation site of the receiver object* of the method containing the allocation site

- Example:



Runtime Flow Tracking

- JVM flow tracking
 - Implementation in an IBM production JVM: J9
 - Tag all application data with dataflow metadata information, referred to as *tracking data*
- Shadow locations
 - Stack location: an extra local variable
 - Object field: a *shadow heap* that has the same size of the Java heap
 - The shadow location of a runtime instance *a* can be calculated through *addr(a) + distance*

Client Analyses to Find Bloat

- Hot chain detector
 - Recover hot copy chains from copy graph based on heuristics
- Clone detector
 - Find pairs of objects (**O1, O2**) with a large volume of copies occurring between the *two object sub-graphs* reachable from them
- Not assigned to heap (NATH) analysis
 - Find allocation site nodes that do not have outgoing edges

Detect Real World Bloat

- DaCapo bloat
 - **Data copy profile**: 28% instructions executed are copies
 - 50% of all data copies came from a variety of *toString* and *append* methods
- What we found from **hot copy chains**
 - Most of these calls centered around code of the form *Assert.isTrue(cond, "bug: " + node)*
 - The second argument is printed only when *cond* evaluates to true
- Elimination of these strings resulted in
 - **65%** reduction in objects created
 - **29% - 35%** reduction in running time

Detect Real World Bloat (Cond.)

- Eclipse 3.1
 - A large framework-intensive application
 - Performance problems result from the pile-up of wasteful operations in its plugins
- What we found from **NATH analysis** report
 - Millions of objects were created solely for the purpose of carrying data across one-level method invocations
- **9.3%** running time reduction

Copy Graph Characteristics

- Memory overhead
 - Shadow heap size: the same as the size of the Java heap
 - Space for storing copy graph: less than **27M** for DaCapo, **150M** for IBM document server for 1-object-sensitive analysis
- Time overhead
 - On average **37X** slowdown for 1-object-sensitive analysis
 - Optimization may be achieved by employing sampling-based profiling

Conclusions

- Copy activity is a good indicator of bloat
- Profiling copies
 - Data copy profiles show performance problems
 - Copy graph profiling helps pinpoint certain performance bottlenecks in an application
- Three client analyses based on copy graph
- Experimental results
 - Problems were found in real world large applications
 - Although incurring significant overhead, the tool works for large scale long-running programs

Thank you

Data Flow Tracking Example

program

```
C c = new C();  
//alloc1
```

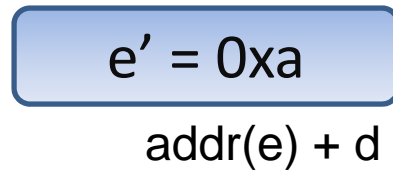
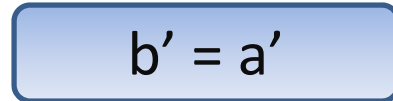
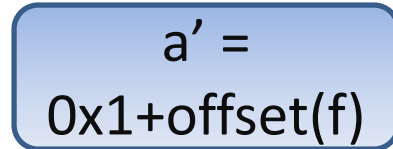
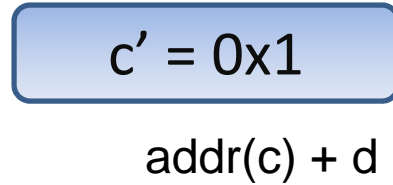
```
...  
int a = c.f;
```

```
...  
int b = a;
```

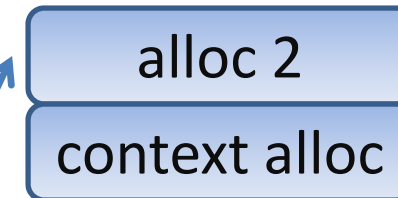
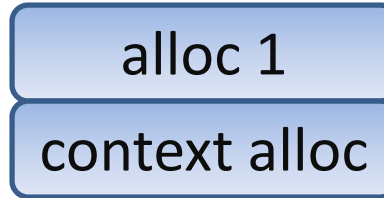
```
...  
E e = new E();  
//alloc2
```

```
...  
e.m = b;
```

shadow stack



shadow heap



copy graph

