JOLT: REDUCING OBJECT CHURN

AJ Shankar, Matt Arnold, Ras Bodik
UC Berkeley, IBM Research | OOPSLA 2008
What is Object Churn?

Allocation of intermediate objects with short lifespans

[Mitchell, Sevitsky 06]

```java
int foo() {
    return bar().length;
}

String bar() {
    return new String("foobaz");
}
```
Churn is a Problem

- A natural result of abstraction
  - Common in large component-based applications
- Reduces program performance
  - Puts pressure on GC
  - Inhibits parallelization (temp objects are synchronized)
  - Requires unnecessary CPU cycles
- Hard to eliminate
  - *Escape analysis?* Objects escape allocating function
  - *Refactoring?* It requires cross-component changes
What is escape analysis?

- Typical defensive copying approach to returning a compound value

```java
public class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
    public Point(Point p) { this(p.x, p.y); }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

```java
public class Component {
    private Point location;
    public Point getLocation() { return new Point(location); }
    public double getDistanceFrom(Component other) {
        Point otherLocation = other.getLocation();
        int deltaX = otherLocation.getX() - location.getX();
        int deltaY = otherLocation.getY() - location.getY();
        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
    }
}
```
What is escape analysis? cont’d

- A smart JVM can see what is going on and optimize away the allocation of the defensive copy

```java
public double getDistanceFrom(Component other) {
    Point otherLocation = new Point(other.x, other.y);
    int deltaX = otherLocation.x - location.x;
    int deltaY = otherLocation.y - location.y;
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
}
```
Point is truly thread-local and its lifetime is known to be bounded by the basic block, it can be either stack-allocated or optimized away entirely.

```java
public double getDistanceFrom(Component other) {
    int tempX = other.x, tempY = other.y;
    int deltaX = tempX - location.x;
    int deltaY = tempY - location.y;
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
}
```
Jolt: Our Contribution

- Automatic runtime churn reduction (in a JIT compiler)
  - Lightweight dynamic analyses, simple optimization
- Implemented in IBM’s J9 JVM
  - Ran on large component-based benchmarks
- Removes 4x as many allocs as escape analysis alone
  - Speedups of up to 15%
Objects Escape Allocation Context

- Traditional EA: hands tied
- Several escape analyses explore up the stack to add context [Blanchet 99, Whaley 99, Gay 00]
- Object allocation optimization based on escape analysis
  - Do not perform well component-based applications
  - Largely because many churn objects escape their allocating functions
Houston, We Have a Solution

Jolt uses a two-part solution:

1. Dynamic analyses find churn-laden subprograms
   - Rooted at a function
     - Only as many contexts as functions in program
   - Subprograms can contain many churned objects

2. Selectively inline portions of subprogram into root to create context
   - Churned objects no longer escape context
   - Can now run escape analysis
Step 1: Find Roots: Churn Analysis

- Goal: Identify roots of churn-laden subprograms
  - Operate on static call graph (JIT’s domain)
  - Use dynamic heap information to track churn

- Use three dynamic analyses inspired by [Dufour 07]:
  - Capture
  - %Capture
  - %Control
Capture

Capture(f) = # objs allocated by f or descendants that die before f returns

In example:

Capture(f) = 4

Answers: Enough churn in the subprogram rooted at f to be worth optimizing?

High Capture \rightarrow YES
%Capture

%Capture(f) = % objs allocated by f or descendants that die before f returns

In example:
%Capture(f) = 4/6

Answers: Better to root at f than at parent of f?

High %Capture $\rightarrow$ YES
%Control

%Control(f) = % objs allocated that are captured by f but not captured by descendants

In example:

%Control(f) = 3/6

Answers: Better to root at f than at child of f?

High %Control $\rightarrow$ YES
All Together Now

- Three analyses together pinpoint subprogram root

High Capture: Worth optimizing

High %Capture: Better f than parent

High %Control: Better f than child
How to Compute Analyses

- **Goals:**
  - Efficient runtime mechanism
  - Thread-safe
  - Simple to add to existing JIT code

- **Solution:** Track heap allocation pointer, GC
  - Requires thread-local heaps (TLHs) & copy collector
    - Supported by virtually all modern JVMs
  - Alternative solution works for any JVM + GC
    - Details in Appendix
1. Choose to sample function $f$
2. Track thread local heap alloc pointer through $f$’s child calls
3. Run GC at the end of $f$
4. Compute capture and control

Capture($f$) = $|\text{Capture}(f)|$
%Capture($f$) = $|\text{Capture}(f)| / |\text{Alloc}(f)|$
%Control($f$) = $|\text{Capture}(f)| - \sum |\text{Capture}(c)|$
|Alloc(f)|

Computed from sampling runs on children
Step 2: Optimize: Smart Inlining

- Churn analyses identified subprogram roots
- Now, inline subprogram to expose allocs to EA
  - Respect JIT optimization constraints (size bound)
  - We can do better than inlining whole subprogram

Only need to inline functions that add churned allocation sites to root
Step 2: Optimize: Smart Inlining

- Goal: inline descendants that expose most # of churned allocs to EA
  - While still respecting size bound
- NP-Hard problem! (can solve Knapsack)
Knapsack Approximation

- Simple poly-time approximation:
  - Inline child with greatest ratio of object allocations to code size
    - $\uparrow \%\text{capture}(f) \Rightarrow \text{objs alloc'd in c are churned}$
  - Repeat until size limit is reached
  - But greedy = short-sighted!

Diagram:
- Root
- A: Low ratio
- B: High ratio
- B will never be inlined because A will never be inlined
Would like to inline child if *its subprogram* has churn elimination potential

We already have an approximation: alloc(c)

- Recall that alloc(c) is num allocs in entire subprogram

So: feed Knapsack approx alloc(c) instead of number of local object allocations in c
Eliminating Allocations

- Once descendants have been inlined, pass to Escape Analysis
  - Use JIT’s existing EA
  - Because of smart inlining, objects’ allocation sites in f, lifetimes don’t escape f
  - EA eliminates allocations via stack allocation or scalar replacement
  - Bonus: improvements in EA == better JOLT
Experimental Methodology

- Implemented Jolt in IBM’s J9 JVM
- Fully automatic, transparent
- Ran on large-scale benchmarks
  - Eclipse
  - JPetStore on Spring
  - TPC-W on JBoss
  - SPECjbb2005
  - DaCapo
## Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Base %Objs Elim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>0.4%</td>
</tr>
<tr>
<td>JPetStore on Spring</td>
<td>0.7%</td>
</tr>
<tr>
<td>TPCW on JBoss</td>
<td>0.0%</td>
</tr>
<tr>
<td>SPECjbb2005</td>
<td>9.6%</td>
</tr>
<tr>
<td>DaCapo</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

- EA performs poorly on large component-based apps
- Median ratio: 4.3x as many objects removed by Jolt
  - Still many more to go
- Median speedup: 4.8%
Additional Experiments

- Runtime overhead acceptable
  - Average compilation overhead: 32% longer
    - Acceptable for long-running programs (< 15 s)
    - Often outweighed by speedup
  - Average profiling overhead: 1.0%
    - Run at 1 sample per 100k function invocations
- Combination of churn analyses and inlining performs better than either alone
  - In every case, Jolt outperformed separate configurations
Summary

- Jolt reduces object churn
  - Transparently at runtime
  - In large applications
  - Easy to implement
    - Uses existing JIT technologies heavily

- Two-part approach
  - Dynamic churn analyses: capture and control
    - Pinpoint roots of good subprograms
  - Smart inlining
    - Uses analyses and Knapsack approximation to inline beneficial functions into root

- Thanks!