



PRECISE MEMORY LEAK DETECTION FOR JAVA SOFTWARE USING CONTAINER PROFILING

[ICSE'08, GUOQING XU]

-1. 予備知識

- メモリリーク(リーク)
 - 省略
- コンテナ(コレクション):
 - (ざっくりと) ヒープオブジェクトの集合
 - 配列、木構造、スタック...などなど
 - コンテナのルートから到達可能なすべてのオブジェクトを含む
 - コンテナ in コンテナもあり得る
 - 基本三操作を持つ前提
 - ADD : コンテナ内に 1 オブジェクトを追加
 - GET : コンテナ内の 1 オブジェクトを参照
 - REMOVE : コンテナ内の 1 オブジェクトを削除

0. 概要

- 提案: コンテナに注目した動的リーク検知手法 (in Java)
 - スライド内ではCB(Container-Based)と略記
- モチベ: 大半のJavaでのリークはコンテナ型やろ
 - Sun bug repository[23]のバグも大半がコンテナ由来っぽいぞ !
- 概説:
 - 1. コンテナへの操作にアノテーションを付与、**モデル化**
 - 2. リークが起きている可能性が高い期間 (**leak region**) を特定
 - 3. 期間内に存在する各コンテナを**二つの尺度**で評価/リーク判定
 - **MC**(Memory Contribution)と**SC**(Staleness Contribution)
- 実験結果:
 - 公認バグ計3つのリーク箇所を発見 (in JDK, in SPECjbb)
 - 実行オーバーヘッドも現実的 (平均 +88.5%)

[23] bugs.sun.com/bugdatabase.

1. 背景(関連研究)

- 静的解析: やっぱりメモリリークの判定は動的に限る (超意訳)
 - 静的では解決が難しいこと
 - ヒープモデリング、マルチスレッドモデリング、リフレクション、etc...
- 動的解析: 既存手法はリークの原因特定に寄与するか疑問
 - 主な解析メソッド
 - Growing types[14, 17]: 増え続ける型がリーク
 - Staleness[1] : ずっとアクセスされないオブジェクトがリーク



追跡対象がオブジェクト単位
→ オブジェクトの情報を集約 → ボトムアップ型の判定

- CB: はじめっからコンテナに着目
 - ◎◎の付け所がシャープでしょ
 - コンテナ単位で情報を集約 → トップダウン型の判定

[1] Bell: Bit-encoding online memory leak detection. [ASPLOS'06]

[14] Cork: Dynamic memory leak detection for garbage-collected languages. [POPL'07]

[17] Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. [ECOOP'03]

1. 背景(星取表)

| | 対象言語 | 計装粒度 | 尺度 | 原因特定 | 精度 | オバヘ |
|-----------------|---------|---------------------|-----|-----------|--------------|-----|
| Container Based | Java | Source & Bytecode | G/S | Top down | FN 有り | 低~中 |
| Cork [14] | Java/C# | Compiler & Bytecode | G | Bottom up | FP, FN 有り | 低 |
| LeakBot [17] | Java | Bytecode | G | Bottom up | FP, FN 有り | 低 |
| Sleigh [1] | Java/C# | Compiler & Bytecode | S | Bottom up | FP 有り | 低 |

| | | | | | | |
|-----------------|-------|--------|-----|-----------|----------|---|
| BIGLeak (参考) | C/C++ | Binary | G/S | Bottom up | FP 有り | 大 |
|-----------------|-------|--------|-----|-----------|----------|---|

尺度 G: Growth S: Staleness

[1] Bell: Bit-encoding online memory leak detection. [ASPLOS'06]

[14] Cork: Dynamic memory leak detection for garbage-collected languages. [POPL'07]

[17] Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. [ECOOP'03]

1. 背景(星取表)

| | 対象言語 | 計装粒度 | 尺度 | 原因特定 | 精度 | オバヘ |
|-----------------|---------|---------------------|-----|-----------|--------------|-----|
| Container Based | Java | Source & Bytecode | G/S | Top down | FN 有り | 低~中 |
| Cork [14] | Java/C# | Compiler & Bytecode | G | Bottom up | FP, FN 有り | 低 |
| LeakBot [17] | Java | Bytecode | G | Bottom up | FP, FN 有り | 低 |
| Sleigh [1] | Java/C# | Compiler & Bytecode | S | Bottom up | FP 有り | 低 |
| BIGLeak (参考) | C/C++ | Binary | G/S | Bottom up | FP 有り | 大 |

尺度 G: Growth S: Staleness

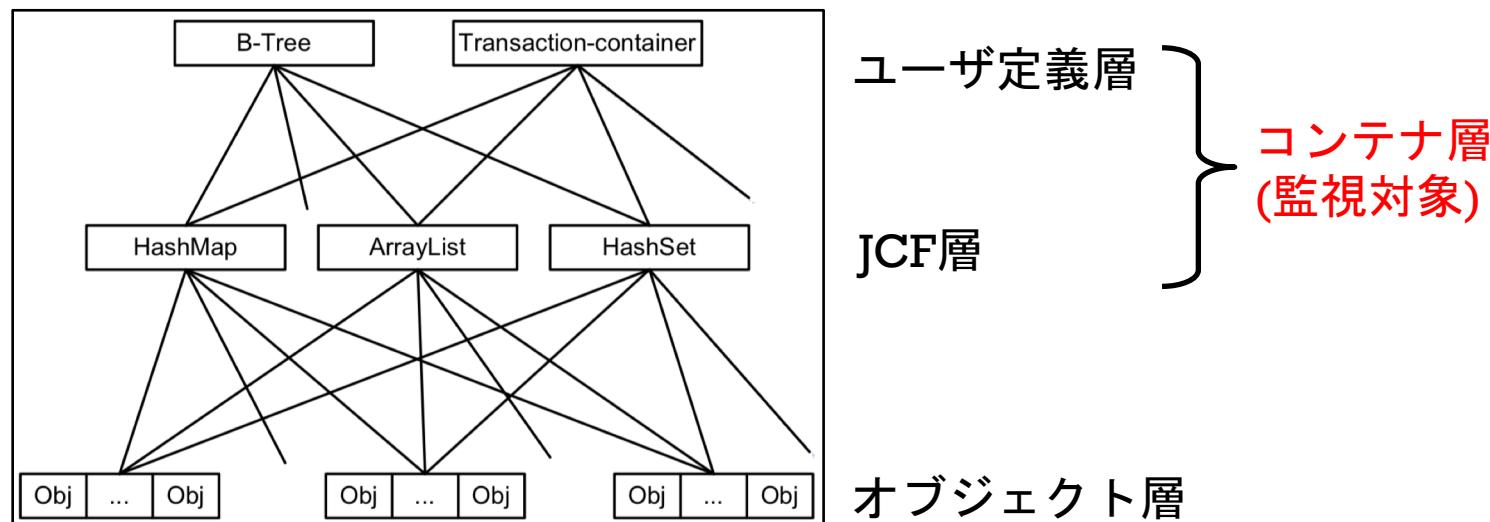
[1] Bell: Bit-encoding online memory leak detection. [ASPLOS'06]

[14] Cork: Dynamic memory leak detection for garbage-collected languages. [POPL'07]

[17] Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. [ECOOP'03]

2. 問題設定

- Java Collection Frameworkとユーザ定義コンテナの操作に着目
 - ADD(σ , o) : コンテナ σ に要素(オブジェクト)oを追加
 - GET(σ) : コンテナ σ から要素を1つ取得
 - REMOVE(σ , o) : コンテナ σ 内から要素oを1つ削除
- 個々のオブジェクトの関連性は無視
 - 値が変わった? 知らんな。
 - 参照が変わった? コンテナ内にいるならいいじゃん



2. 問題設定

- CBのざっくりフローチャート
 - ①コンテナ操作のモデル化
 - 対象プログラムソースコードにモデリング関数を挿入(4: Annotation)
 - ②プログラム実行、情報収集
 - モデリング関数とJVM監視ツールで収集(4: Profiling)
 - ③計測終了、解析開始
 - リークが起きている時間の特定(3: Leak region)
 - 各コンテナのMC、SCを計算(3: MC, SC)
 - ④解析終了、レポート化
 - 各コンテナのLCを計算(3: LC)
 - リークと関係しているコールサイトを計算(4: Data analysis)

3. 提案手法(LEAK REGION)

- **Leak region** $[\tau_s, \tau_e]$: リークが起きてそうな時間領域

- GCごとに計測されるメモリ消費量の遷移
- ユーザ定義のパラメータ

} 決定要因

- 手順:

■

■

■

■

■

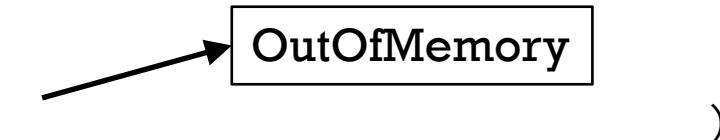
■

■

■

■

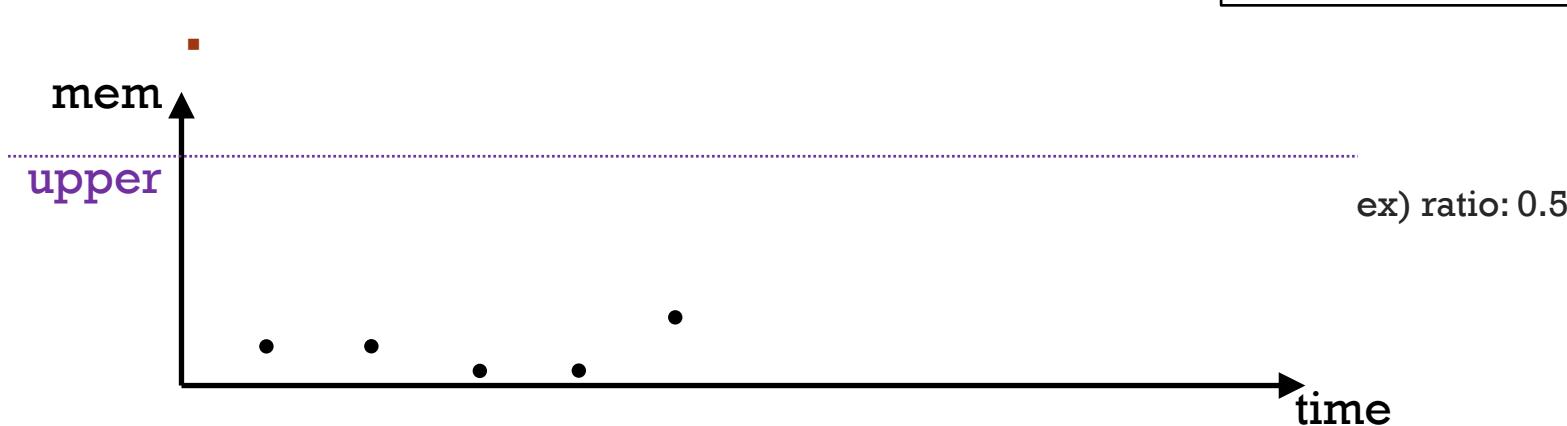
■



OutOfMemory

)

$m(x)$: 時刻 x のメモリ消費量
 τ_{gc} : GC直後の時刻
 ss : GCのサブシーケンス

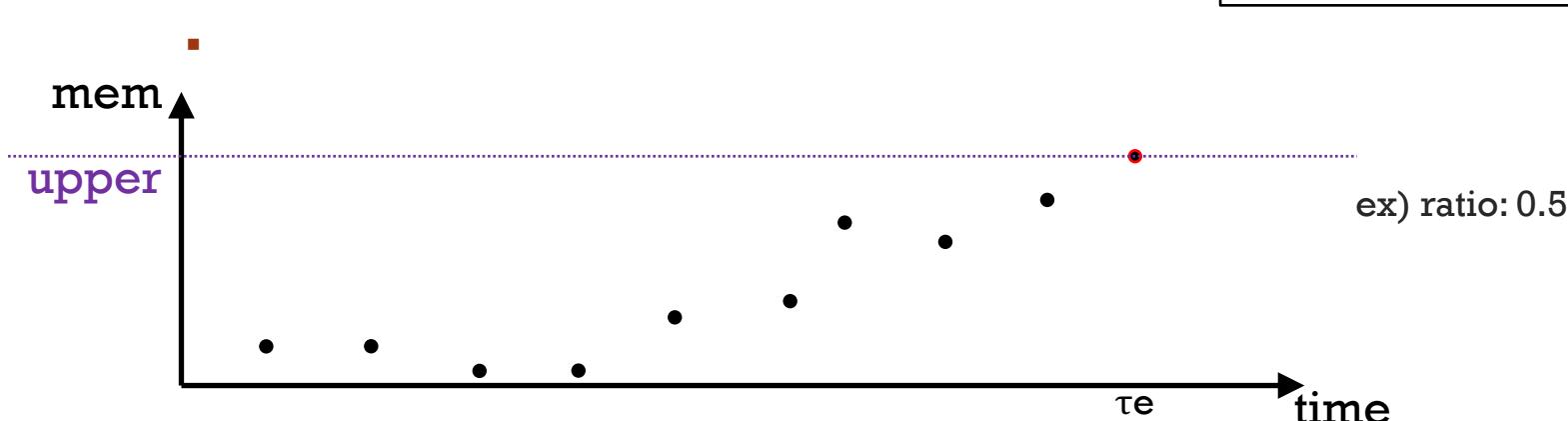


3. 提案手法(LEAK REGION)

- **Leak region** $[\tau_s, \tau_e]$: リークが起きてそうな時間領域
 - GCごとに計測されるメモリ消費量の遷移
 - ユーザ定義のパラメータ
- 決定要因
- 手順:
 - ① τ_e : 計測終了点が決定 (プログラム終了時 or ユーザ任意の時刻)
 -
 -
 -
 -
 -
 -

OutOfMemory

$m(x)$: 時刻 x のメモリ消費量
 τ_{gc} : GC直後の時刻
 ss : GCのサブシーケンス

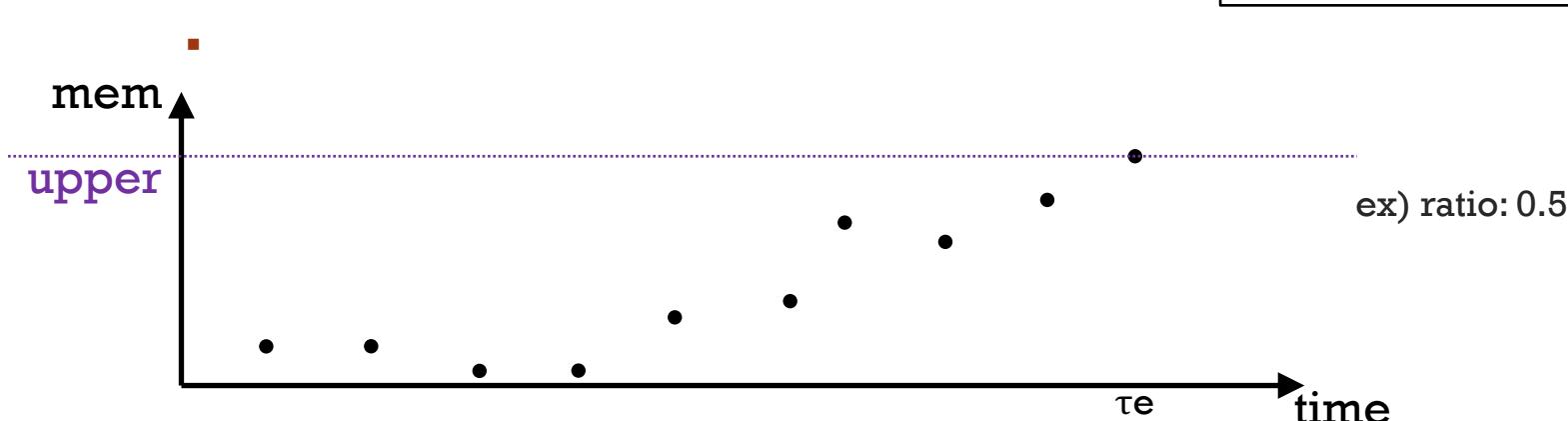


3. 提案手法(LEAK REGION)

- **Leak region** $[\tau_s, \tau_e]$: リークが起きてそうな時間領域
 - GCごとに計測されるメモリ消費量の遷移
 - ユーザ定義のパラメータ
- 決定要因
- 手順:
 - ① τ_e : 計測終了点が決定 (プログラム終了時 or ユーザ任意の時刻)
 - ② τ_s : 評価開始点が決定 (以下の条件を満たすように)
 - Cond1: $\forall i (m(\tau_s) \leq m(\tau_{gc_i}) \leq m(\tau_e)) \quad \tau_{gc_i} \in [\tau_s, \tau_e]$
 -
 -
 -
 -

OutOfMemory

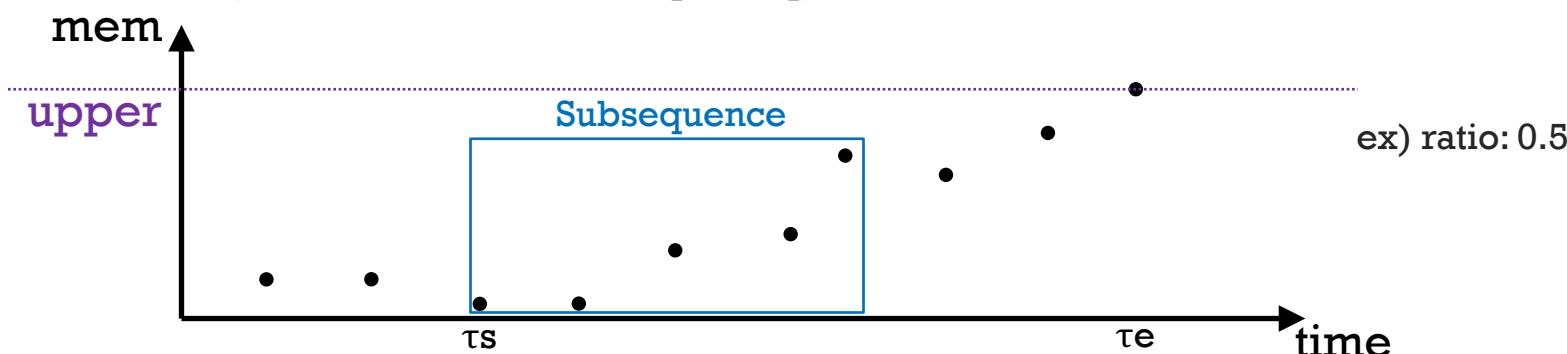
$m(x)$: 時刻 x のメモリ消費量
 τ_{gc} : GC直後の時刻
 ss : GCのサブシーケンス



3. 提案手法(LEAK REGION)

- **Leak region** $[\tau_s, \tau_e]$: リークが起きてそうな時間領域
 - GCごとに計測されるメモリ消費量の遷移
 - ユーザ定義のパラメータ
- 決定要因
- 手順:
 - ① τ_e : 計測終了点が決定 (プログラム終了時 or ユーザ任意の時刻)
 - ② τ_s : 評価開始点が決定 (以下の条件を満たすように)
 - Cond1: $\forall i (m(\tau_s) \leq m(\tau_{gc_i}) \leq m(\tau_e)) \quad \tau_{gc_i} \in [\tau_s, \tau_e]$
 - Cond2: $\exists ss$
 - $\forall j (m(\tau_{gc_j}) \leq m(\tau_{gc_j+1})) \quad \tau_{gc_j} \in ss$
 - $\#GC_in_ss \geq 2$
 - $(\#GC_in_ss / \#GC_in_[\tau_s, \tau_e]) > user_def_ratio$

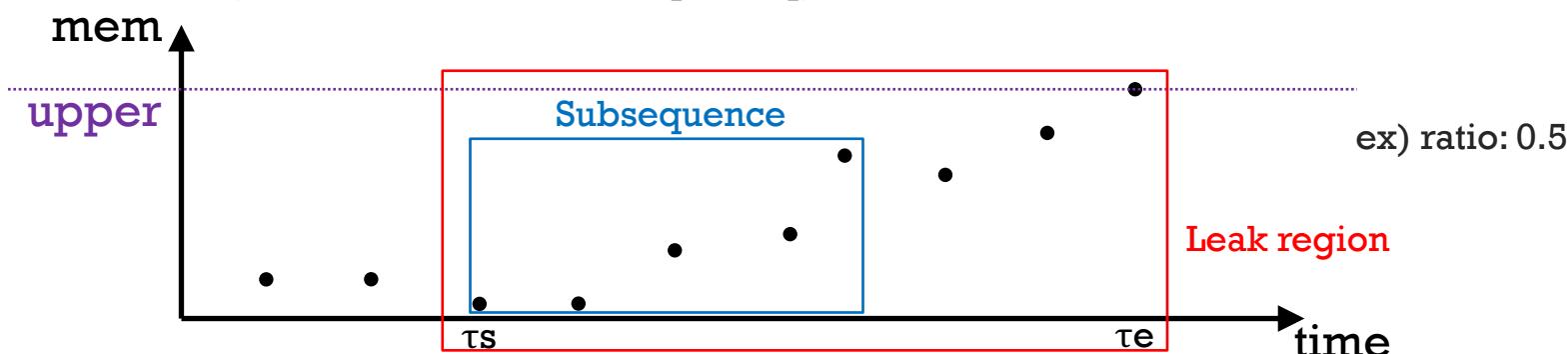
$m(x)$: 時刻 x のメモリ消費量
 τ_{gc} : GC直後の時刻
 ss : GCのサブシーケンス



3. 提案手法(LEAK REGION)

- **Leak region** $[\tau_s, \tau_e]$: リークが起きてそうな時間領域
 - GCごとに計測されるメモリ消費量の遷移
 - ユーザ定義のパラメータ
- 決定要因
- 手順:
 - ① τ_e : 計測終了点が決定 (プログラム終了時 or ユーザ任意の時刻)
 - ② τ_s : 評価開始点が決定 (以下の条件を満たすように)
 - Cond1: $\forall i (m(\tau_s) \leq m(\tau_{gc_i}) \leq m(\tau_e)) \quad \tau_{gc_i} \in [\tau_s, \tau_e]$
 - Cond2: $\exists ss$
 - $\forall j (m(\tau_{gc_j}) \leq m(\tau_{gc_j+1})) \quad \tau_{gc_j} \in ss$
 - $\#GC_in_ss \geq 2$
 - $(\#GC_in_ss / \#GC_in_[\tau_s, \tau_e]) > user_def_ratio$

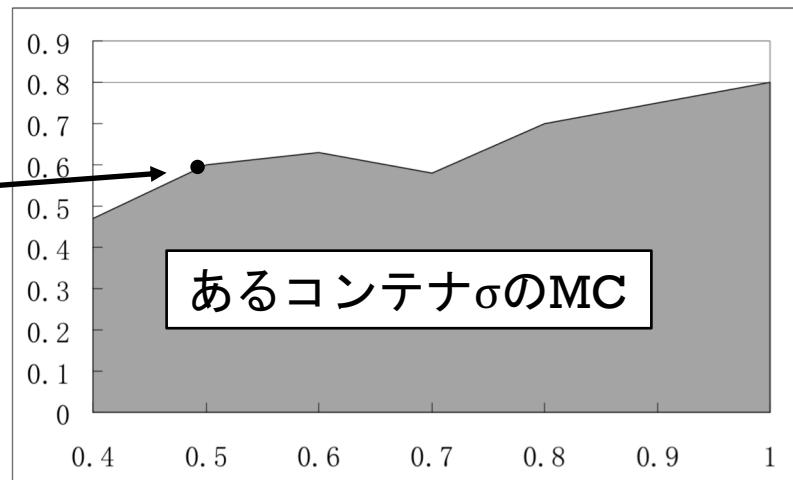
$m(x)$: 時刻 x のメモリ消費量
 τ_{gc} : GC直後の時刻
 ss : GCのサブシーケンス



3. 提案手法(MEMORY CONTRIBUTION)

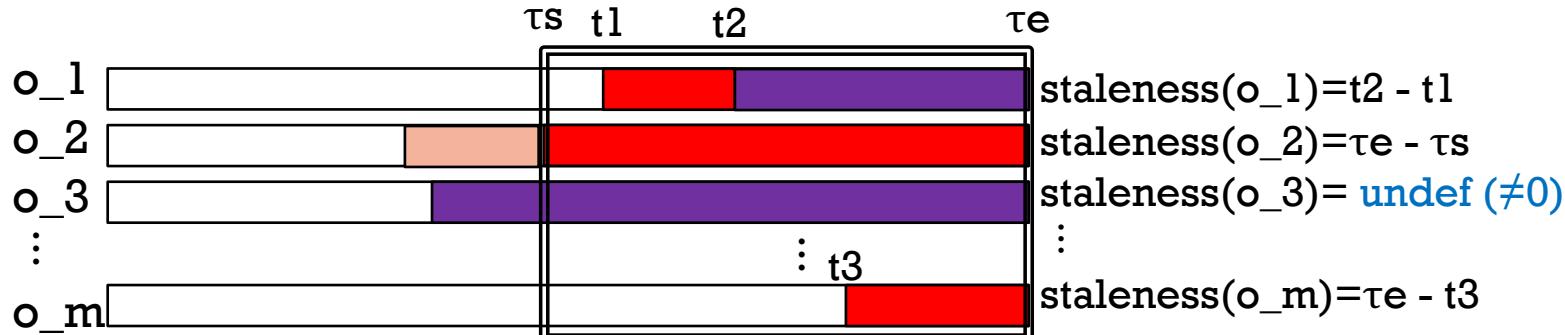
- **MemoryContribution(MC)**: (ざっくり) 下図の色付き領域の面積
 - 図のX軸: Leak region区間長を1とした正規化時刻
 - 図のY軸: 各コンテナが占めるメモリの割合 (ヒープサイズ全体を1)
- 実際には近似で $MC(\sigma) \doteq \sum_{\tau^{GC} \in [\tau_s, \tau_e]} \left(\frac{\tau^{GC}_{i+N} - \tau^{GC}_i}{\tau_e - \tau_s} \right) * \frac{mem(\sigma)_i}{total_i}$ $N=15,50,85$
- 当然 $0 \leq MC \leq 1$
 - あるコンテナのMCが大
 - → **Leak region**内でそのコンテナがヒープを占める割合が大

例えばこの点は
 τ_s と τ_e の中間ぐらいで
ヒープの約6割を
占めているの意



3. 提案手法(STALENESS CONTRIBUTION)

- **StalenessContribution(SC):**
 - (ざっくり)コンテナオブジェクトの正規化Stalenessの平均値
 - Staleness: 最終アクセスor割り付け ~ 判定時刻or解放 の間隔
 - をさらに $\tau_e - \tau_s = 1$ として正規化
 - $SC(\sigma_j^n) = \frac{1}{n} \sum_{o_i \in \sigma_j^n} \frac{staleness(o_i)}{\tau_e - \tau_s}$
 - Leak region開始時にn個のオブジェクトoを持つコンテナ σ_j
 - もちろん $0 \leq SC \leq 1$
 - あるコンテナのSCが大
 - →そのコンテナはアクセス後すぐ解放されないオブジェクトが多い
- : staleness ■ : dead (after deallocation)



3. 提案手法(LEAKING CONFIDENCE)

- LeakingConfidence(LC): MCとSCから導出されるリークっぽさ
- $LC = SC * MC^{1-SC}$
 - SCの指數関数という位置づけ (著者の考え: 重要度は $SC > MC$)
- Of course, $0 \leq LC \leq 1$
 - $MC \approx 0 \rightarrow LC \approx 0$:十分小サイズなコンテナはスルー(＼ω＼)
 - $MC \approx 1 \rightarrow LC \approx SC$:MCが大きいならLCはSC依存
 - $SC \approx 0 \rightarrow LC \approx 0$:短命なコンテナは大きさに関わらずスルー(＼ω＼)
 - $SC \approx 1 \rightarrow LC \approx 1$:曲者じゃ！出合え、出合え！
- LCはリークと非リークのコンテナで大きな差が出る

| <i>ID</i> | <i>Type</i> | <i>LC</i> | <i>MC</i> | <i>SC</i> |
|-----------|--------------------|-----------|-----------|-----------|
| 11324773 | util.HashMap | 0.449 | 0.824 | 0.495 |
| 18429817 | util.LinkedList | 0.165 | 0.820 | 0.194 |
| 8984226 | util.LinkedList | 0.050 | 0.809 | 0.062 |
| 2263554 | util.WeakHashMap | 0.028 | 0.820 | 0.034 |
| 15378471 | util.LinkedList | 0.018 | 0.029 | 0.256 |
| 5192610 | swing.JLayeredPane | 0.011 | 0.824 | 0.013 |
| 30675736 | swing.JPanel | 0.011 | 0.824 | 0.013 |
| 19526581 | swing.JRootPane | 0.011 | 0.824 | 0.013 |
| 17933228 | util.Hashtable | 0.000023 | 0.0007 | 0.026 |

サンプル(SunBug#6209673)
表の先頭HashMapは
実際にリーク

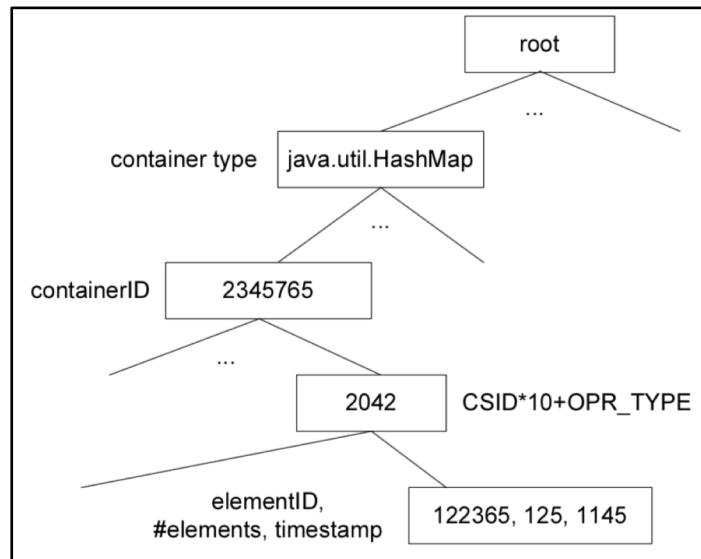
4. 実装(ANNOTATION)

- ソースコード内のコンテナ操作の前後にモデリング用関数を計装
 - using Soot[24]
 - モデリング用関数: _before, _afterのサフィックスがついてる関数
 - 操作(OPR)の(種類, CallSite)を(コンテナ, 要素, タイムスタンプ)と関連付け
 - 計装の大半は自動的に計装可能
 - 一部は手動で付与
 - 実験では更にエスケープ解析を用いて計装対象を絞っている
 - 曰く、初見ユーザもやり方を覚えれば使えるよ、とのこと

```
class Java_util_HashMap {  
    static void put_after(int csID, Map receiver, Object key,  
                         Object value, Object result) {  
        /* if key does not exist in the map */  
        if (result == null) {  
            /* use user-defined hash code as ID */  
            Recorder.v().useUserDefHashCode();  
            /* record operation ADD(receiver,key) */  
            Recorder.v().record(csID, receiver, key,  
                                receiver.size()-1, Recorder.EFFECT_ADD);  
        }  
    }  
}
```

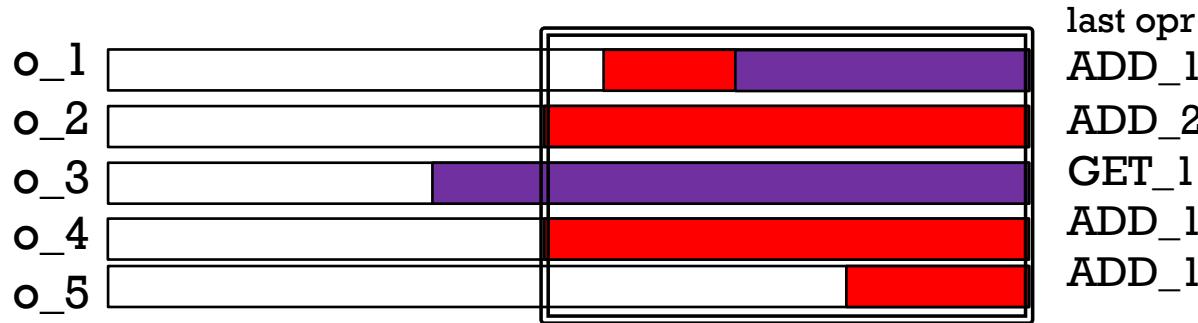
4. 実装(PROFILING)

- 計測時、図のようなメタデータ集合体(とGC情報)を構築
 - モデリング用関数によってコンテナ操作の情報を収集
 - for calculating SC
 - リフレクションによって各コンテナのサイズ情報を収集
 - for calculating MC
 - JVMTIによってGCの時刻、GC後のメモリ消費量を収集
 - for determining Leak region
 - JVM Tool Interface: Java bytecodeの監視/計装ツール



4. 実装(DATA ANALYSIS)

- τ_e が決定(データ収集が終了)したら、収集データから解析を開始
 - Leak regionの決定 → MC, SC, LCの導出
 - レポート出力
 - レポートにはstalenessの高いコールサイト(cs)3つも記載



- staleness(ADD_1) = (stl(o_1)+stl(o_4)+stl(o_5)) / 3
- staleness(ADD_2) = stl(o_2) / 1
- staleness(GET_1) = 0
- レポートにはコンテナをLC順で報告、各コンテナ最大3箇所csの情報

5. 実験(LEAK ANALYSIS)

- 環境: 2.4GHz dual-core PC with 2GB RAM
- リーク位置の特定
 - コンテナサイズの計測の間隔の変化による出力の変化も測定
 - 1/15, 1/50, 1/85 のレートでサンプリング
- ヒープ上限:
 - 各プログラムのデフォルト
- 対象プログラム:
 - JDK bug #6209673 in Sun bug database
 - JDK bug #6559589 in Sun bug database
 - SPECjbb bug

} Java AWT/Swing 系のバグ
AWT/Swingにはリークが多いため選択

5. 実験(LEAK ANALYSIS) IN JDK BUG #6209673

- 報告されたリーク原因
 - コンテナ: volatileMap (HashMap)
 - cs: RepaintManger:591行目(GET)

- 結果:
 - バグ有りver.とバグ解消済みver.を比較
 - 検証した結果、報告されたコンテナ、csが正しいことを確認
 - やったぜ

- 著者考察:
 - 1/85はHashMapについては報告
LinkedListについては未報告
 - →精度が伴わない可能性

```

Container:11324773 type: java.util.HashMap
(LC: 0.449, SC: 0.495, MC: 0.825)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.507)
Container:18429817 type: java.util.LinkedList
(LC: 0.165, SC: 0.194, MC: 0.820)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.246)
Container:8984226 type: java.util.LinkedList
(LC: 0.051, SC: 0.062, MC: 0.809)
---cs: java.awt.DefaultKeyboardFocusManager:851 (0.063)
---cs: java.awt.DefaultKeyboardFocusManager:740 (0.025)
Data analyzed in 149203ms
(a) 1/15gc sampling rate
  
```

```

Container:29781703 type: java.util.HashMap
(LC: 0.443, SC: 0.480, MC: 0.855)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.480)
Container:2263554 type: class java.util.LinkedList
(LC: 0.145, SC: 0.172, MC: 0.814)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.017)
Container:399262 type: class javax.swing.JPanel
(LC: 0.038, SC: 0.044, MC: 0.860)
---cs: javax.swing.JComponent:796 (0.044)
Data analyzed in 21593ms
(b) 1/50gc sampling rate
  
```

```

Container:15255515 type: java.util.HashMap
(LC: 0.384, SC: 0.426, MC: 0.835)
---cs: javax.swing.RepaintManager:591 (0.426)
Container:19275647 type: java.util.LinkedList
(LC: 0.064, SC: 0.199, MC: 0.244)
---cs: java.awt.SequencedEvent:176 (0.204)
---cs: java.awt.SequencedEvent:179 (0.010)
---cs: java.awt.SequencedEvent:128 (1.660E-4)
Container:28774302 type: javax.swing.JPanel
(LC: 0.036, SC: 0.042, MC: 0.839)
---cs: javax.swing.JComponent:796 (0.042)
Data analyzed in 10547ms
(c) 1/85gc sampling rate
  
```

5. 実験(LEAK ANALYSIS) IN JDK BUG #6209673

- 報告されたリーク原因
 - コンテナ: volatileMap (HashMap)
 - cs: RepaintManger:591行目(GET)

- 結果:
 - バグ有りver.とバグ解消済みver.を比較
 - 検証した結果、報告されたコンテナ、csが正しいことを確認
 - やったぜ

- 著者考察:
 - 1/85はHashMapについては報告
LinkedListについては未報告
 - →精度が伴わない可能性

```

Container:11324773 type: java.util.HashMap
(LC: 0.449, SC: 0.495, MC: 0.825)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.507)
Container:18429817 type: java.util.LinkedList
(LC: 0.165, SC: 0.194, MC: 0.820)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.246)
Container:8984226 type: java.util.LinkedList
(LC: 0.051, SC: 0.062, MC: 0.809)
---cs: java.awt.DefaultKeyboardFocusManager:851 (0.063)
---cs: java.awt.DefaultKeyboardFocusManager:740 (0.025)
Data analyzed in 149203ms
(a) 1/15gc sampling rate

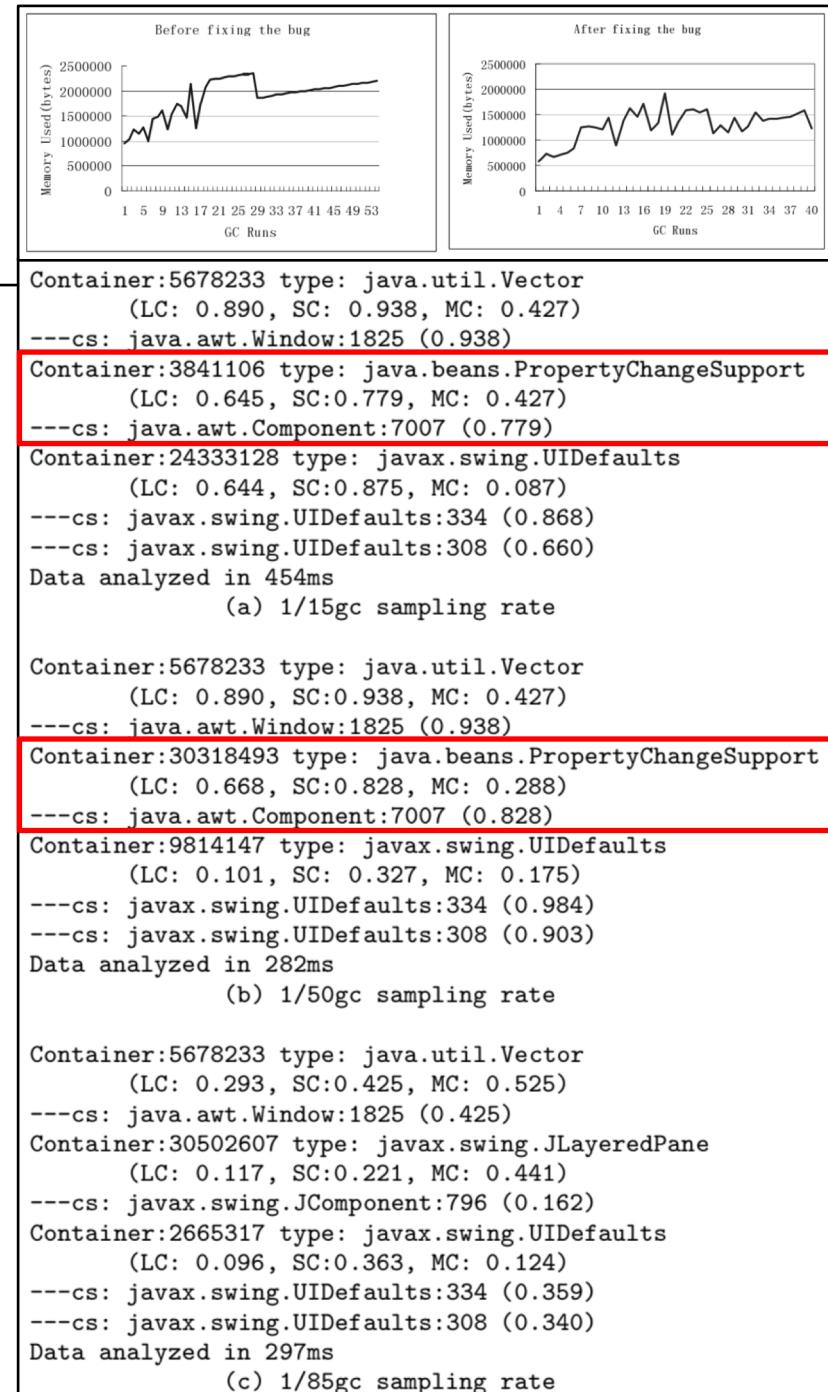
Container:29781703 type: java.util.HashMap
(LC: 0.443, SC: 0.480, MC: 0.855)
---cs: javax.swing.RepaintManager:591 (Average staleness: 0.480)
Container:2263554 type: class java.util.LinkedList
(LC: 0.145, SC: 0.172, MC: 0.814)
---cs: java.awt.DefaultKeyboardFocusManager:738 (0.017)
Container:399262 type: class javax.swing.JPanel
(LC: 0.038, SC: 0.044, MC: 0.860)
---cs: javax.swing.JComponent:796 (0.044)
Data analyzed in 21593ms
(b) 1/50gc sampling rate

Container:15255515 type: java.util.HashMap
(LC: 0.384, SC: 0.426, MC: 0.835)
---cs: javax.swing.RepaintManager:591 (0.426)
Container:19275647 type: java.util.LinkedList
(LC: 0.064, SC: 0.199, MC: 0.244)
---cs: java.awt.SequencedEvent:176 (0.204)
---cs: java.awt.SequencedEvent:179 (0.010)
---cs: java.awt.SequencedEvent:128 (1.660E-4)
Container:28774302 type: javax.swing.JPanel
(LC: 0.036, SC: 0.042, MC: 0.839)
---cs: javax.swing.JComponent:796 (0.042)
Data analyzed in 10547ms
(c) 1/85gc sampling rate

```

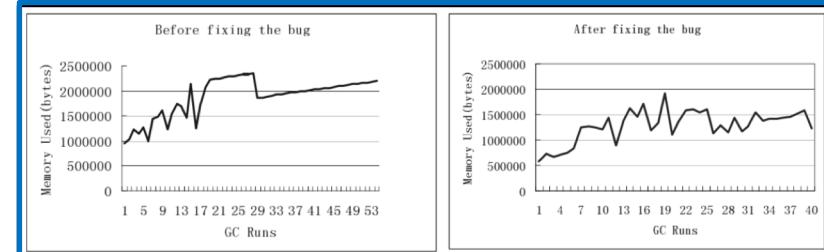
5. 実験(LEAK ANALYSIS) IN JDK BUG #6559589

- 報告されたリーク原因
 - ownedWindowList (Vector)
 - これは性質的にリークではない
 - PropertyChangeSupport
 - こっちが本命
 - cs: Component: 7007行目(ADD)
- 結果:
 - 調べたところ、
オーバーライドが不正確だった
 - 適切な削除用メソッド呼ばれず
 - 当該箇所を修正(右上)したら
リークの解消が見られた
 - 当時このバグは未解決
- 著者考察:
 - 1/85は見逃してますねえ...



5. 実験(LEAK ANALYSIS) IN JDK BUG #6559589

- 報告されたリーク原因
 - ownedWindowList (Vector)
 - これは性質的にリークではない
 - PropertyChangeSupport
 - こっちが本命
 - cs: Component: 7007行目(ADD)
- 結果:
 - 調べたところ、
オーバーライドが不正確だった
 - 適切な削除用メソッド呼ばれず
 - 当該箇所を修正(右上)したら
リークの解消が見られた
 - 当時このバグは未解決
- 著者考察:
 - 1/85は見逃してますねえ...



```

Container:5678233 type: java.util.Vector
(LC: 0.890, SC: 0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)
Container:3841106 type: java.beans.PropertyChangeSupport
(LC: 0.645, SC: 0.779, MC: 0.427)
---cs: java.awt.Component:7007 (0.779)
Container:24333128 type: javax.swing.UIManager
(LC: 0.644, SC: 0.875, MC: 0.087)
---cs: javax.swing.UIManager:334 (0.868)
---cs: javax.swing.UIManager:308 (0.660)
Data analyzed in 454ms
(a) 1/15gc sampling rate

Container:5678233 type: java.util.Vector
(LC: 0.890, SC: 0.938, MC: 0.427)
---cs: java.awt.Window:1825 (0.938)
Container:30318493 type: java.beans.PropertyChangeSupport
(LC: 0.668, SC: 0.828, MC: 0.288)
---cs: java.awt.Component:7007 (0.828)
Container:9814147 type: javax.swing.UIManager
(LC: 0.101, SC: 0.327, MC: 0.175)
---cs: javax.swing.UIManager:334 (0.984)
---cs: javax.swing.UIManager:308 (0.903)
Data analyzed in 282ms
(b) 1/50gc sampling rate

Container:5678233 type: java.util.Vector
(LC: 0.293, SC: 0.425, MC: 0.525)
---cs: java.awt.Window:1825 (0.425)
Container:30502607 type: javax.swing.JLayeredPane
(LC: 0.117, SC: 0.221, MC: 0.441)
---cs: javax.swing.JComponent:796 (0.162)
Container:2665317 type: javax.swing.UIManager
(LC: 0.096, SC: 0.363, MC: 0.124)
---cs: javax.swing.UIManager:334 (0.359)
---cs: javax.swing.UIManager:308 (0.340)
Data analyzed in 297ms
(c) 1/85gc sampling rate
  
```

5. 実験(LEAK ANALYSIS) IN SPECJBB2000 BUG

- 1/50のみ
- 報告されたリーク原因
 - longBTree
 - cs: District: 264行目(ADD)
- 結果:
 - 追加されたインスタンス内のメソッドの呼び出しを調べると、正しく繰り返されていなかった
 - 当該箇所を修正するとリークの改善が見られた
- 著者考察:
 - このプログラムに無知でも2,3時間で原因の特定ができた
 - Cork[14]のチームは1日かかった

```

Container:4451472 type: java.util.Hashtable
(LC: 0.135, SC: 0.190, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:225 (0.214)
---cs: spec.jbb.StockLevelTransaction:211 (0.190)
Container:7776424 type: java.util.Hashtable
(LC: 0.110, SC: 0.157, MC: 0.659)
---cs: spec.jbb.StockLevelTransaction:211 (0.157)
---cs: spec.jbb.StockLevelTransaction:225 (0.114)
Container:28739781 type: java.util.Hashtable
(LC: 0.102, SC: 0.146, MC: 0.654)
---cs: spec.jbb.StockLevelTransaction:211 (0.146)
---cs: spec.jbb.StockLevelTransaction:225 (0.122)
Data analyzed in 4078ms
(a) before modeling of longBTree, using 1/50gc

Container:27419736 type: spec.jbb.infra.Collections.longBTree
(LC: 0.687, SC: 0.758, MC: 0.666)
---cs: spec.jbb.District:264 (0.826)
---cs: spec.jbb.StockLevelTransaction:225 (0.624)
---cs: spec.jbb.StockLevelTransaction:211 (0.519)
Container:21689791 type: spec.jbb.infra.Collections.longBTree
(LC: 0.685, SC: 0.757, MC: 0.662)
---cs: spec.jbb.District:264 (0.783)
---cs: spec.jbb.StockLevelTransaction:211 (0.370)
---cs: spec.jbb.District:406 (2.944E-4)
Container:27521273 type: spec.jbb.infra.Collections.longBTree
(LC: 0.667, SC: 0.727, MC: 0.727)
---cs: spec.jbb.Warehouse:456 (0.798)
---cs: spec.jbb.District:264 (0.784)
---cs: spec.jbb.StockLevelTransaction:211 (0.484)
Data analyzed in 7579ms
(b) after modeling of longBTree, using 1/50gc

```

5. 実験(LEAK ANALYSIS)

FALSE POSITIVE

- さくっと
- 誤検出が起きないか、リークのない検体に適用/実験
- 結果: 全検体でLCは十分小さな値に
 - 恐らく最大のLC
 - →リークが起きていないければ、LCが大きくなることはない

| | LC | | LC | | LC |
|----------|--------|--------|--------|---------|--------|
| antlr | 4.1E-5 | chart | 2.7E-6 | fop | 1.3E-5 |
| hsqldb | 4.4E-7 | jython | 5.0E-8 | luindex | 9.1E-5 |
| lusearch | 2.3E-2 | pmd | 4.3E-6 | xalan | 5.2E-5 |
| jflex | 1.8E-7 | | | | |

5. 実験(OVERHEAD)

- 環境は
- 動的オーバーヘッド実験
 - サンプリングレートは1/15と1/50のみ
 - 静的エスケープ解析による計装の削減を実行
- ヒープ上限
 - 各プログラムのデフォルト
 - 大きめの固定サイズ: 512MB
- 対象プログラム:
 - 前述のリークありプログラム3つ
 - リークなしプログラム10コ

5. 実験(OVERHEAD)

| Program | (a) | | | RT _o (s) | (c) 1/15gc | | | | (d) 1/50gc | | | | %OH |
|----------|------|------------------|-------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|--------|
| | #IS | #IS _e | IT(s) | | #GC _d | RT _d (s) | #GC _l | RT _l (s) | #GC _d | RT _s (s) | #GC _l | RT _l (s) | |
| antlr | 176 | 123 | 87 | 17.9 | 387 | 18.4 | 10 | 18.1 | 387 | 18.4 | 10 | 18.1 | 0.7% |
| chart | 894 | 867 | 202 | 8.5 | 5368 | 38.0 | 185 | 35.4 | 4109 | 36.5 | 185 | 35.1 | 313.2% |
| fop | 1378 | 1375 | 125 | 4.5 | 693 | 8.6 | 24 | 7.8 | 545 | 8.9 | 24 | 6.4 | 44.6% |
| hsqldb | 684 | 674 | 116 | 4.3 | 54 | 4.7 | 8 | 4.4 | 54 | 4.7 | 8 | 4.4 | 1.6% |
| jython | 443 | 416 | 135 | 7.3 | 1653 | 31.8 | 126 | 28.2 | 1440 | 31.4 | 126 | 28.5 | 298.3% |
| luindex | 442 | 409 | 65 | 19.5 | 1446 | 24.4 | 40 | 23.7 | 1390 | 23.9 | 40 | 23.7 | 21.2% |
| lusearch | 442 | 388 | 81 | 2.9 | 418 | 9.1 | 21 | 3.9 | 326 | 8.2 | 23 | 3.2 | 11.7% |
| pmd | 814 | 690 | 111 | 5.9 | 2938 | 26.9 | 716 | 18.4 | 2766 | 25.2 | 37 | 6.6 | 10.8% |
| xalan | 755 | 752 | 114 | 1.4 | 655 | 7.7 | 30 | 4.0 | 605 | 6.2 | 18 | 3.7 | 165.9% |
| jflex | 522 | 438 | 92 | 45.1 | 4171 | 170.7 | 1493 | 130.3 | 2126 | 165.8 | 665 | 88.05 | 95.2% |
| bug 1 | 3109 | 2768 | 487 | — | 18630 | 600 | 7420 | 600 | 11457 | 600 | 1983 | 600 | — |
| bug 2 | 3105 | 2770 | 502 | 38.1 | 512 | 53.0 | 243 | 42.3 | 413 | 52.2 | 37 | 42 | 10.5% |
| specjbb | 74 | 73 | 142 | — | 18605 | 3600 | 15080 | 3600 | 16789 | 3600 | 10810 | 3600 | — |

- (a): エスケープ解析による計装箇所の変化
 - #IS : エスケープ解析なしで計装された関数の数
 - #IS_e: エスケープ解析ありで計装された関数の数
 - IT(s): エスケープ解析あり自動計装にかかった時間
- (b): nativeの実行時間
- 結果: エスケープ解析による追跡対象の削減ができた
 - 最大124CS削減

5. 実験(OVERHEAD)

| Program | (a) | | | (b) | | (c) 1/15gc | | | | (d) 1/50gc | | | (e) |
|----------|------|------------------|-------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|--------|
| | #IS | #IS _e | IT(s) | RT _o (s) | #GC _d | RT _d (s) | #GC _l | RT _l (s) | #GC _d | RT _s (s) | #GC _l | RT _l (s) | %OH |
| antlr | 176 | 123 | 87 | 17.9 | 387 | 18.4 | 10 | 18.1 | 387 | 18.4 | 10 | 18.1 | 0.7% |
| chart | 894 | 867 | 202 | 8.5 | 5368 | 38.0 | 185 | 35.4 | 4109 | 36.5 | 185 | 35.1 | 313.2% |
| fop | 1378 | 1375 | 125 | 4.5 | 693 | 8.6 | 24 | 7.8 | 545 | 8.9 | 24 | 6.4 | 44.6% |
| hsqldb | 684 | 674 | 116 | 4.3 | 54 | 4.7 | 8 | 4.4 | 54 | 4.7 | 8 | 4.4 | 1.6% |
| jython | 443 | 416 | 135 | 7.3 | 1653 | 31.8 | 126 | 28.2 | 1440 | 31.4 | 126 | 28.5 | 298.3% |
| luindex | 442 | 409 | 65 | 19.5 | 1446 | 24.4 | 40 | 23.7 | 1390 | 23.9 | 40 | 23.7 | 21.2% |
| lusearch | 442 | 388 | 81 | 2.9 | 418 | 9.1 | 21 | 3.9 | 326 | 8.2 | 23 | 3.2 | 11.7% |
| pmd | 814 | 690 | 111 | 5.9 | 2938 | 26.9 | 716 | 18.4 | 2766 | 25.2 | 37 | 6.6 | 10.8% |
| xalan | 755 | 752 | 114 | 1.4 | 655 | 7.7 | 30 | 4.0 | 605 | 6.2 | 18 | 3.7 | 165.9% |
| jflex | 522 | 438 | 92 | 45.1 | 4171 | 170.7 | 1493 | 130.3 | 2126 | 165.8 | 665 | 88.05 | 95.2% |
| bug 1 | 3109 | 2768 | 487 | — | 18630 | 600 | 7420 | 600 | 11457 | 600 | 1983 | 600 | — |
| bug 2 | 3105 | 2770 | 502 | 38.1 | 512 | 53.0 | 243 | 42.3 | 413 | 52.2 | 37 | 42 | 10.5% |
| specjbb | 74 | 73 | 142 | — | 18605 | 3600 | 15080 | 3600 | 16789 | 3600 | 10810 | 3600 | — |

- (c): 1/15サンプリングでのGC数と実行時間
 - 左2列: ヒープ上限: デフォルト
 - 右2列: ヒープ上限: 512MB
- (d): 1/50サンプリングでのGC数と実行時間
- 結果: おおよそ 1/50 の方がオーバーヘッドは少ない
 - トラバース回数、ディスクアクセス、スレッド同期などが減ったため
 - GC数が少ないと誤差レベル
- 結果: 同じくヒープサイズを大きくするとオバヘは減少

5. 実験(OVERHEAD)

| Program | (a) | | | (b) | | (c) 1/15gc | | | | (d) 1/50gc | | | | (e) %OH |
|----------|------|------------------|-------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|------------------|---------------------|--------|------------|
| | #IS | #IS _e | IT(s) | RT _o (s) | #GC _d | RT _d (s) | #GC _l | RT _l (s) | #GC _d | RT _s (s) | #GC _l | RT _l (s) | | |
| antlr | 176 | 123 | 87 | 17.9 | 387 | 18.4 | 10 | 18.1 | 387 | 18.4 | 10 | 18.1 | 0.7% | |
| chart | 894 | 867 | 202 | 8.5 | 5368 | 38.0 | 185 | 35.4 | 4109 | 36.5 | 185 | 35.1 | 313.2% | |
| fop | 1378 | 1375 | 125 | 4.5 | 693 | 8.6 | 24 | 7.8 | 545 | 8.9 | 24 | 6.4 | 44.6% | |
| hsqldb | 684 | 674 | 116 | 4.3 | 54 | 4.7 | 8 | 4.4 | 54 | 4.7 | 8 | 4.4 | 1.6% | |
| jython | 443 | 416 | 135 | 7.3 | 1653 | 31.8 | 126 | 28.2 | 1440 | 31.4 | 126 | 28.5 | 298.3% | |
| luindex | 442 | 409 | 65 | 19.5 | 1446 | 24.4 | 40 | 23.7 | 1390 | 23.9 | 40 | 23.7 | 21.2% | |
| lusearch | 442 | 388 | 81 | 2.9 | 418 | 9.1 | 21 | 3.9 | 326 | 8.2 | 23 | 3.2 | 11.7% | |
| pmd | 814 | 690 | 111 | 5.9 | 2938 | 26.9 | 716 | 18.4 | 2766 | 25.2 | 37 | 6.6 | 10.8% | |
| xalan | 755 | 752 | 114 | 1.4 | 655 | 7.7 | 30 | 4.0 | 605 | 6.2 | 18 | 3.7 | 165.9% | |
| jflex | 522 | 438 | 92 | 45.1 | 4171 | 170.7 | 1493 | 130.3 | 2126 | 165.8 | 665 | 88.05 | 95.2% | |
| bug 1 | 3109 | 2768 | 487 | — | 18630 | 600 | 7420 | 600 | 11457 | 600 | 1983 | 600 | — | |
| bug 2 | 3105 | 2770 | 502 | 38.1 | 512 | 53.0 | 243 | 42.3 | 413 | 52.2 | 37 | 42 | 10.5% | |
| specjbb | 74 | 73 | 142 | — | 18605 | 3600 | 15080 | 3600 | 16789 | 3600 | 10810 | 3600 | — | |

- (e): 追加オーバーヘッド
 - (e) = (d).RT_l / (b)
- 結果: 平均実行オーバーヘッドは+88.5%
 - デバッグには十分許容可
 - 運用には重すぎ（とのことらしいです）
 - (小泉)オーバーヘッドが少ないか大きいかで二極化してるんですが...

6. まとめ

- Container-basedによるメモリリークの検出手法
 - アイデア:
 - Javaのコンテナに注目、コンテナに適したモデル化
 - Leak regionによるリーク時期の絞り込み
 - メモリ消費量, Stalenessにより各コンテナを評価
 - 結果:
 - 再現されたリークバグの原因を正しく特定
 - オーバーヘッドもデバッグには有用な域
 - 補足:
 - プログラム開発時期での適用を想定
 - ソースコードがある前提
 - 致命的なリーク原因の特定に重点を置く

その他(MC近似)

- MCの精密な計算にはGCごとの各コンテナのサイズが必要
 - コンテナのルートから到達可能なオブジェクトを調べる必要
 - 毎回GC毎にトラバースするのはオーバーヘッドが馬鹿にならない
 - →近似しようそうしよう
- 実験も踏まえて、GC50回ごとにサンプリングがいい感じ
 - 精度を保つつつオーバーヘッドを削減
 - トラバースしなかった期間、サイズは変わらないと仮定
 - 近似MCは $\text{time}(50\text{GC}) * \text{size}(\sigma)$ の総和
- 同時に、記録していた操作データをディスクにダンプ
 - 自身がメモリを圧迫しないように

感想

- 未解決バグ(#6559589)の原因特定はインパクトがデカい
- 検査対象を絞るという潔さ
- 動的オーバーヘッドを軽減するテクニックを多く取り入れていた

- デバッグという観点では強力
 - セキュリティ方面に使おうとすると、制限が多い
 - ソースコードが必要
 - Java Collection Framework限定
 - 限定ってわけではないんだろうけど...