Decrypting the Java Gene Pool.

Sébastien Marion
Richard Jones
Chris Ryder

University of Kent, Canterbury
Overview

- Pretenuring, what is it?
- Micro-patterns, what are they?
- Using micro-patterns to do pretenuring.
- What did we find?
- Performance.
- Conclusion.
Contributions

- Object lifetimes are related to code patterns.
- Knowledge bank associating patterns with object lifetime.
- GC time reductions by up to 77%.
Pretenuring

- Basic idea: In a generational GC, if we know that an object is going to live a long time, we can avoid unnecessary copying by allocating the object straight into the old generation.

**Generational GC**
- Red *objects* are those we failed to predict.
- Only objects we were able to predict long-lived are **pretenured**.

Ref: *Pretenuring for Java*, by Blackburn et al. OOPSLA’01
Pretenuring, pros and cons

Pros:

- Reduces copying.
- Makes nursery GC cheaper.
- Can greatly reduce GC time.
- Advice can be build-time and/or program-specific.

Cons:

- Wrong predictions increase pressure on the mature space, and may reduce available space for nursery.
- Wrongly predicting immortals makes their referents immortal.
- Predicting lifetimes requires accurate recording and study of program trace files.
  - Takes days/weeks.
  - Generates Gigabytes of data to analyse.
  - Has to be run for each program prior to executing it.
Blackburn et al.’s Lifetime Classification

Classifying Objects:

- **Immortal**: Dies more than halfway between its time of birth and the end of the program.
  - Avoids all copying.
  - A **copy-reserve** is necessary for non-immortal objects
    ⇒ Each object allocated uses twice as much space as its size.
  - Immortal region never copied ⇒ objects don’t require copy reserve.

- **Long-lived**: Not immortal and age greater than a certain threshold.

- **Short-lived**: Neither immortal, nor long-lived.
Blackburn et al’s Lifetime Classification

- Classifying allocation sites:
  - Points in the program at generating memory allocation.
  - E.g. ```Foo foo = new Foo();```

  - For each site, compute ratios of:
    - Short-lived objects
    - Long-lived objects
    - Immortal objects

  - Compute sites’ lifetimes using these ratios.
Motivation

- Capture programmer’s intentions.
  - Identify object lifetime from class design?

- Design Patterns:
  - Describe interactions between classes.
  - Hard to mechanically detect.

- Micro-Patterns (Java):
  - Class level properties.
  - Mechanically detectable.
Micro-patterns (MPs)

- 30 different micro-patterns.
- A class can exhibit several micro-patterns.
- Characteristics of MPs
  - Are the methods private/public/static?
  - Are the fields private/public/static?
  - Type of class: Interface? Inheriting?
  - Etc...
- E.g.:
  
  **Designator**: An interface with no members.
  **Joiner**: An empty interface joining two or more superinterfaces.
  **Stateless**: A class with no fields, other than static final ones.

*Ref: Micro Patterns in Java Code, by Gil and Maman. OOPSLA’05*
Why bother with MPs?

- Pretenuring improves performance but:
  - Requires ahead of time recording/analyses of huge traces for each program.

- MPs allow:
  - Mechanical and quick classification of each class.

- If MPs can predict object lifetimes…
  - No need to gather program traces, yet…
  - Provides program-specific advice.
Extracting knowledge

- Previously, we saw how to classify allocation sites’ lifetimes.
- Determine sets of MPs associated with each site (source and destination).
  - E.g.

```
public class Src {
    ...
    IDst bar = new Dst();
    ...
}
```

Source

Destination
Extracting knowledge

Problem:

- Extract knowledge from data

- Huge amount of data:
  - Thousands of sites.
  - 60 predictors for each.
Data-mining approach

- **Goal:**
  - Identify which MPs are good lifetime predictors.
  - Generate rules to predict lifetime.

- **Software used:**
  - Clementine 10.0, using C.5.0 algorithm.
Machine Learning Procedure (1)

- Training phase:
  - Gather trace files from as many programs as possible.
  - Compute:
    - Allocation sites - object lifetime relation.
    - Micro-pattern set - object lifetime relation.
  - Extract knowledge by datamining.
  - Create knowledge bank of rules matching MP sets to lifetimes.
Benefits

- Unlike other approaches, we gather trace files only once.

- We can predict previously unseen programs.
Machine Learning Procedure (2)

- Program specific phase:
  - Program analysis done in seconds.
  - Determine MPs for each class.
  - Identify site lifetimes by matching MPs with our knowledge bank.
  - Ahead of time, or inside class-loader.
Strategy summary

- Training phase:
  - Site $\rightarrow$ Lifetime (traces)
  - Site $\rightarrow$ \{MPSrc\} x \{MPDst\} (*Gil and Maman MP Tool*)
  - MP Rules $\rightarrow$ Lifetime (*Data-Mining*)

- Program specific phase:
  - Site $\rightarrow$ \{MPSrc\} x \{MPDst\} (*Gil and Maman’s MP tool*)
  - Site $\rightarrow$ Lifetime (*Lifetime knowledge bank matching*)
Example rule:

- Short-lived example:
  - Compound box related rules:
    - Rule #1, 99.3%:
      - Destination is compound box.
      - Example: Class `java.io.StringBuffer`
        - Explanation: Used as temporary data-structure.

*Compound box*: A class with exactly one, non primitive instance field.
Example rule:

- Immortal example:
  - Sampler related rules:
    - Rule #11, 99.6%:
      - Source is a Sampler.
      - Example: Class `java.awt.Color`
      - Explanation: Contains fields such as `red, green, blue`, which would live a very long time.

*Sampler*: A class with one or more public constructors, and at least one static field of the same type as the class.
Implementation

Our system:
- Jikes RVM 2.4.4

Input:
- Advice file mapping allocation site to lifetime.
- Parsed and stored into a HashMap, keys shared with the compiler (VM_Atoms).

Allocation:
- Short ➔ Default nursery space.
- Long ➔ Mature space.
- Immortal ➔ Immortal region.
Advice loading overhead

- **Time overhead:**
  - 0.48% on average for Spec JVM98 Speed 100.
  - Likely to be insignificant in real applications.

- **Space overhead:**
  - 52KB (13 pages):
    - Benchmark: _213_javac.
    - Number of advice entries: 656.

- Removing short-lived advice would substantially reduce overheads.
Performance of self-prediction:

- Self-prediction from program-specific traces.
- Best possible case.
- Reduces GC time:
  - Up to 80% improvements in GC time.

- Exception:
  - _228_jack performs poorly at large heap sizes (>3x minimum heap size)

Ref: Profile-based Pretenuring, by Blackburn et al. TOPLAS’07
Performance: self-predictions

Slowdowns vs Speedups

Heap size (X times minimum heap size)

ISMM'07, 21 Oct 2007 Decrypting The Java Gene Pool 23
True prediction

- Datamining rules.
- Level of confidence: 80%.
  - Good conservative/aggressive compromise.
  - Ignore rules with confidence level less than threshold.
  - If conflicts (extremely rare), use rule with highest confidence.
- Generate knowledge bank from DaCapo benchmarks.
- Apply to SpecJVM98.
- Results:
  - Up to 68% GC time reduction.
  - Pause time reduced in most cases.
GC time, 80% confidence

Speedups vs. Slowdowns

- 205_raytrace
- 209_db
- 213 javac
- 227_mtrt
- 201_compress
- 202_jess
- 222_mpegaudio
- 226_jack

Heap size (X times minimum heap size)
 Allocation sites that matter

- A few sites have a major impact on GC time.

  - _205_raytrace:
    - Sites that matter the most:
      - Lspec/benchmarks/_205_raytrace/OctNode::CreateFaces(FFFFFF)V:12
      - Lspec/benchmarks/_205_raytrace/OctNode::Initialize()V:12

  - _209_db:
    - Site that matters the most:
      - Lspec/benchmarks/_209_db/Database::read_db(Ljava/lang/String;)V:243
True prediction Vs Self prediction
GC time vs. Throughput

- **Issue:**
  - GC time improvements do not always translate into throughput improvements.
  - E.g: _209_db at 75% confidence
    - Up to 77% GC improvement
    - 13-16% throughput degradation.
Picture throughput, 80% confidence

Speedups

Slowdowns

% Execution time

-10

0

10

throughput

Heap size (X times minimum heap size)

205_raytrace
289_db
213_javac
227_ntrt
Investigation

- Investigation:
  - In SpecJVM98, GC time is small.
  - Write barrier (next slide).
  - Hardware performance counters.
Write barrier (209_db, @75%)

- No change in L1 or L2 cache misses.
- Pretenuring increases DTLB misses by 33%.
Conclusion

- Micro-patterns:
  - Relation between MPs and objects’ lifetime.
- Machine learning:
  - Learning phase performed once.
  - Program-specific advice generation is fast.
  - Even for new programs.
- GC time improvement:
  - For several programs.
Future work

- Incorporate Jikes RVM specific advice into the build-time image.

- Find other ways of identifying common coding practices such as CJKM metrics to apply machine learning.