Write Barrier Removal by Static Analysis

Karen Zee
Martin Rinard

MIT Laboratory for Computer Science
Research Goal

Use static program analysis to identify and remove unnecessary write barriers in programs that use generational garbage collection.
Write Barriers and Generational Garbage Collection
Garbage Collection

“Root Set”
Garbage Collection

“Root Set”
Garbage Collection

“Root Set”
Generational Garbage Collection

Young Generation  Old Generation
Generational Garbage Collection
Generational Garbage Collection

Young Generation

Old Generation
Generational Garbage Collection

Young Generation  Old Generation

“Root Set”
Generational Garbage Collection

Young Generation

Old Generation

“Root Set”

Remembered set of references
Write Barriers

- Code added by the compiler to every store of a reference into the heap
- Ex. Remembered set of references
  - Array of locations that could contain references into the young generation
  - Maintained at store instructions:
    
    ```
    v1.f = v2; // store
    rs[i++] = &v1.f; // write barrier
    ```
- Undesirable overhead on stores to the heap
Generational Garbage Collection

Young Generation  Old Generation

- Young-to-Young
- Young-to-Old
- Old-to-Young
- Old-to-Old
Write Barrier Removal Principle

- Write barrier unnecessary for stores that create references from younger to older objects
- Analysis approach
  - Identify stores that create references from younger to older objects
  - Specifically, references from the youngest object to other objects
- Then remove the corresponding write barriers
The Analyses

- Track youngest object
- Target Java bytecode
- Intraprocedural
- Interprocedural
  - With callee information
  - With caller information
Intraprocedural Analysis

- Forward dataflow analysis
- Computes must points-to information
- For each point in the program, the analysis generates
  - A set $V$ of local variables
  - All $v \in V$ refer to the youngest object
Using the Analysis Information

- Store statement $v1.f = v2$
- Write barrier for the store statement can be removed if
  - $v1$ refers to the youngest object
  - i.e. $v1 \in V$
## Transfer Functions

<table>
<thead>
<tr>
<th>$\text{stm}$</th>
<th>$\left<a href="V"> \text{stm} \right</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lv = \text{NEW C}$</td>
<td>{ $lv$ }</td>
</tr>
<tr>
<td>$lv1 = lv2$</td>
<td>$V \cup { lv1 }$ if $lv2 \in V$</td>
</tr>
<tr>
<td></td>
<td>$V - { lv1 }$ if $lv2 \notin V$</td>
</tr>
<tr>
<td>$lv = \text{CALL m}(\ldots)$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$lv = \ldots$</td>
<td>$V - { lv }$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$V$</td>
</tr>
</tbody>
</table>
Dataflow Framework

- Join operator is set intersection \( \cap \)
- Initial value of V
  - \( \emptyset \) at method entry
  - All program variables \( Vars \) at all other program points
class Point {
    Integer x;
    Integer y;
}

x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;

...
class Point {
    Integer x;
    Integer y;
}

x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
...
Point Example

class Point {
    Integer x;
    Integer y;
}

... x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
...

`class Point {
    Integer x;
    Integer y;
}

x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
..."
Point Example

class Point {
    Integer x;
    Integer y;
}

... 
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;
p.y = y;
...

Only young-to-old references created. Write barriers should be unnecessary!
Point Example

```java
class Point {
    Integer x;
    Integer y;
}

p = new Point();
p.x = x;
p.y = y;

// write barrier not needed
// write barrier not needed
```

Diagram:

```
v
p = new Point();
{ p }
p.x = x;
{ p }
p.y = y;
{ p }
```

```
x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x;  // write barrier not needed
p.y = y;  // write barrier not needed
...
Point Example

```
p = new Point();
{ p }
p.x = x;
{ p }
p.y = y;
{ p }
```

```
p = NEW Point;
{ p }
CALL Point(p);
∅
p.x = x;
∅
p.y = y;
∅
```
Interprocedural Information
Allocated Object Types

- Consider the method-youngest object
- Track the types of objects that invoked methods may allocate
- Use type-based reasoning to remove write barriers associated with stores that
  - Create a reference from the youngest object allocated in the current method
  - Provided that reference does not point to an object allocated by an invoked method
Allocated Type Analysis

- Flow-insensitive interprocedural analysis
- Collects types of objects allocated either
  - Directly by the given method, or
  - Indirectly by (transitively) invoked methods
Information from Callees

- For each point in the program, the analysis generates a pair $\langle V, T \rangle$
  - Set $V$ of local variables
  - All $v \in V$ refer to the method-youngest object
  - Set $T$ of types
  - Objects of type $t \in T$ may have been allocated by (transitively) invoked methods
Using the Analysis Information

- Store statement \( v1.f = v2 \)
- Write barrier for the store statement can be removed if
  - \( v1 \) refers to the method-youngest object
  - i.e. \( v1 \in V \)
  - \( v2 \) refers to an object older than the method-youngest object
  - i.e. \( \text{subtypes}(C) \cap T = \emptyset \) where \( C \) is the static type of \( v2 \)
**Modified Transfer Functions**

<table>
<thead>
<tr>
<th>st</th>
<th>([ \text{st} ](\langle V, T \rangle) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( lv = \text{NEW C} )</td>
<td>( \langle { lv }, \emptyset \rangle )</td>
</tr>
<tr>
<td>( lv1 = lv2 )</td>
<td>( \langle V \cup { lv1 }, T \rangle ) if ( lv2 \in V )</td>
</tr>
<tr>
<td></td>
<td>( \langle V \setminus { lv1 }, T \rangle ) if ( lv2 \not\in V )</td>
</tr>
<tr>
<td>( lv = \text{CALL } m(...) )</td>
<td>( \langle V', T' \rangle ) where ( V' = V \setminus { lv1 } ) and ( T' = T \cup (\bigcup_{m' \in \text{callees}(m)} \text{alloctypes}(m')) )</td>
</tr>
<tr>
<td>( lv = ... )</td>
<td>( \langle V \setminus { lv }, T \rangle )</td>
</tr>
<tr>
<td>( ... )</td>
<td>( \langle V, T \rangle )</td>
</tr>
</tbody>
</table>
Dataflow Framework

- Join operator is $\langle \cap, \cup \rangle$
- Initial value of $\langle V, T \rangle$
  - $\langle \emptyset, \text{Types} \rangle$ at method entry
  - $\langle \text{Vars}, \emptyset \rangle$ at all other program points
Point Example

\[
\begin{align*}
\langle V, T \rangle & \\
p & = \text{NEW Point} \;
\langle \{ p \}, \emptyset \rangle \\
\text{CALL Point}(p) ; & \\
\langle \{ p \}, \{ \text{Exception} \} \rangle \\
p.x & = x ; \\
\langle \{ p \}, \{ \text{Exception} \} \rangle \\
p.y & = y ; \\
\langle \{ p \}, \{ \text{Exception} \} \rangle
\end{align*}
\]

\text{alloctypes(Point())} = \{ \text{Exception} \}
Point Example

Remove write barrier for $v1.f = v2$ if:
- $v1 \in V$, and
- $\text{subtypes}(C) \cap T = \emptyset$

where $C$ is the static type of $v2$

- $\text{Type}(x) = \text{Integer}$
- $\text{Type}(y) = \text{Integer}$
- $\text{subtypes}(\text{Integer}) \cap \{\text{Exception}\} = \emptyset$
Point Example

- Type(x) = Integer
- Type(y) = Integer
- subtypes(Integer) \(\cap\) \{ Exception \} = \(\emptyset\)

Can remove write barriers!
class Point {
    Integer x;
    Integer y;
}
...

x = new Integer(11);
y = new Integer(7);
p = new Point();
p.x = x; // no write barrier
p.y = y; // no write barrier
...
class Point {
    private Integer x;
    private Integer y;
}

...x = new Integer(11);y = new Integer(7);
p = new Point();
p.x = x; // no write barrier
p.y = y; // no write barrier
...

class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}

...x = new Integer(11);y = new Integer(7);
p = new Point(x, y);
...
Point Constructor Example

class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}

x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);

alloctypes(Object()) = { }
Information from Callers

- Use information from callers
- At call sites where the youngest object is an argument to the call, propagate analysis results to entry point of invoked method
Point Constructor Example

The first argument to the Point constructor is the youngest object.

```java
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}

x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
```
class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x; // write barrier?
        this.y = y; // write barrier?
    }
}
...x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
...
Point Constructor Example

class Point {
    private Integer x;
    private Integer y;
    Point(Integer x, Integer y) {
        this.x = x;  // no write barrier!
        this.y = y;  // no write barrier!
    }
}

x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);

CALL Object(p);

\[ \langle \{ \text{this} \}, \emptyset \rangle \]
\[ \langle \{ \text{this} \}, \emptyset \rangle \]
\[ \text{this} \ x = x; \]
\[ \langle \{ \text{this} \}, \emptyset \rangle \]
\[ \text{this} \ y = y; \]
\[ \langle \{ \text{this} \}, \emptyset \rangle \]
class Point {
    private int x;
    private int y;
    Point(int x, int y) {
        this.x = new Integer(x); // no write barrier
        this.y = new Integer(y); // no write barrier
    }
}

...x = new Integer(11);
y = new Integer(7);
p = new Point(x, y);
...

Clearly the receiver object is older than the Integer objects!
The write barrier for the field assignment cannot be removed.
Solution: Transform program to reverse the order of the allocations!
Now the write barriers for the field assignments can be removed!
Allocation Order Transformations

- Make write barriers removable that previously could not be removed
- Handles recursive constructors for data structures such as trees
Experimental Results
Methodology

- Implemented in Flex compiler infrastructure
  - Ahead-of-time Java bytecode compiler
  - Generates native code or C
- Measured
  - Write barrier counts
  - Execution time
  - Write barrier characterization
  - Analysis time
Dynamic Write Barriers Removed

- BH
- BiSort
- Em3d
- Health
- MST
- Perimeter
- Power
- TreeAdd
- Voronoi

Intraprocedural
Interprocedural
Transformed
Normalized Execution Time

<table>
<thead>
<tr>
<th>Intraprocedural</th>
<th>Interprocedural</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>BiSort</td>
<td>Em3d</td>
</tr>
<tr>
<td>Health</td>
<td>MST</td>
<td>Perimeter</td>
</tr>
<tr>
<td>Power</td>
<td>TreeAdd</td>
<td>Voronoi</td>
</tr>
</tbody>
</table>
## Analysis Time

<table>
<thead>
<tr>
<th></th>
<th>Intraprocedural (s)</th>
<th>Interprocedural (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>15</td>
<td>38</td>
</tr>
<tr>
<td>BiSort</td>
<td>16</td>
<td>34</td>
</tr>
<tr>
<td>Em3d</td>
<td>16</td>
<td>38</td>
</tr>
<tr>
<td>Health</td>
<td>16</td>
<td>38</td>
</tr>
<tr>
<td>MST</td>
<td>16</td>
<td>35</td>
</tr>
<tr>
<td>Perimeter</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>Power</td>
<td>15</td>
<td>38</td>
</tr>
<tr>
<td>TreeAdd</td>
<td>12</td>
<td>37</td>
</tr>
<tr>
<td>Voronoi</td>
<td>14</td>
<td>42</td>
</tr>
</tbody>
</table>
Other Collectors

- Analysis assumptions
  - Objects initially allocated in young generation
  - Older objects promoted before younger
- Concurrent collectors may not promote objects in age-order
- Other collectors may not always allocate objects initially in the young generation
An Optimistic Extension

Young Generation  Old Generation

Reserve a bit in the object header
An Optimistic Extension

Young Generation Old Generation

Remembered set of objects
An Optimistic Extension

Young Generation  Old Generation

Remembered set of objects
An Optimistic Extension

Young Generation  Old Generation

Remembered set of references
An Optimistic Extension

- Out-of-order promotion
- Allocation in other generations
- Multi-threaded programs
- Low overhead
## Overhead of Optimistic Extension

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Without Allocation Order Transformations</th>
<th>With Allocation Order Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>0.36%</td>
<td>-0.95%</td>
</tr>
<tr>
<td>BiSort</td>
<td>-0.33%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>Em3d</td>
<td>0.02%</td>
<td>0.17%</td>
</tr>
<tr>
<td>Health</td>
<td>-0.08%</td>
<td>0.37%</td>
</tr>
<tr>
<td>MST</td>
<td>0.55%</td>
<td>0.19%</td>
</tr>
<tr>
<td>Perimeter</td>
<td>0.37%</td>
<td>0.37%</td>
</tr>
<tr>
<td>Power</td>
<td>0.23%</td>
<td>-0.12%</td>
</tr>
<tr>
<td>TreeAdd</td>
<td>0.06%</td>
<td>0.22%</td>
</tr>
<tr>
<td>Voronoi</td>
<td>0.09%</td>
<td>-0.83%</td>
</tr>
</tbody>
</table>
Related Work
Write Barriers for Generational GC

- Lieberman and Hewitt, 1983 – Entry table
- Ungar, 1984 – Remembered set of objects
- Moon, 1984 – Hardware write barriers
- Shaw, 1988 – Virtual memory write barriers
- Sobalvarro, 1988 – Card and word marking
- Appel, 1989 – Remembered set of addresses
- Wilson and Moher, 1989 – Card marking
- Hudson and Diwan, 1990 – Hashed remembered set
- Zorn, 1990 – Write-protected pages
- Chambers, 1992 – 3 SPARC instructions
- Hölzle, 1993 – 2 SPARC instructions
- Hosking and Hudson, 1993 – Remembered sets + card-marking
Write Barrier Evaluations

- Zorn, 1990
  - Word marking in hardware, software, and page protection
  - Lisp system
- Hosking, Moss, and Stefanović, 1992
  - Remembered sets, card marking, page protection
  - Smalltalk system
- Jones and Lins, 1996
- Blackburn and McKinley, 2002
  - Inlining strategies for remembered sets of references
  - Java system (Jikes RVM)
Type-based Garbage Collection

- Prolific Types
  - Shuf, Gupta, Bordawekar, and Singh, 2002

- Connectivity-Based Garbage Collection
  - Hirzel, Henkel, Diwan, and Hind, 2002
Conclusion

- Static program analysis for removing write barriers used for generational collection
  - Interprocedural information
  - Allocation order transformations
  - Reasonable analysis times
  - In many cases, remove almost all removable write barriers
- Optimistic extension for collectors that may not promote objects in age order
Questions
Q: How does this analysis scale?
A: Because we use interprocedural information, it may seem like the analysis would not scale well. However, the main analysis is actually intraprocedural, even though it uses type information that has to be computed using an interprocedural analysis. Now, that analysis, though interprocedural, is flow-insensitive, and context-insensitive. It can actually be computed very efficiently in linear time. So, in an efficient implementation, we expect the analysis to have very reasonable scaling properties.
Q: Does the analysis support dynamic class loading?
A: Like many other analyses of this type, dynamic class loading may invalidate some analysis results. However, the optimistic extension will allow the system to operate correctly even in the presence of invalid analysis results. This makes the optimistic extension a good, low-overhead solution for using our analysis in the presence of dynamic class loading.