Write Barrier Removal by Static Analysis

Karen Zee and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{kkg, rinard}@lcs.mit.edu

ABSTRACT
We present a set of static analyses for removing write barriers in programs that use generational garbage collection. To our knowledge, these are the first analyses for this purpose. Our Intraprocedural analysis uses a flow-sensitive pointer analysis to locate variables that must point to the most recently allocated object, then eliminates write barriers on stores to objects accessed via one of these variables. The Callee Type Extension incorporates information about the types of objects allocated in invoked methods, while the Caller Context Extension incorporates information about the most recently allocated object at call sites that invoke the currently analyzed method. Results from our implemented system show that our Full Interprocedural analysis, which incorporates both extensions, can eliminate the majority of the write barriers in most of the programs in our benchmark set, producing modest performance improvements of up to 7% of the overall execution time. Moreover, by dynamically instrumenting the executable, we are able to show that for all but two of our nine benchmark programs, our analysis is close to optimal in the sense that it eliminates the write barriers for almost all store instructions observed not to create a reference from an older object to a younger object.

Keywords
Program analysis, pointer analysis, generational garbage collection, write barriers

1. INTRODUCTION
Generational garbage collectors have become the memory management alternative of choice for many safe languages. The basic idea behind generational collection is to segregate objects into different generations based on their age. Generations containing recently allocated objects are typically collected more frequently than older generations; as young objects age by surviving collections, the collector promotes them into older generations. Generational collectors therefore work well for programs that allocate many short-lived objects and some long-lived objects — promoting long-lived objects into older generations enables the garbage collector to quickly scan the objects in younger generations.

Before it scans a generation, the collector must locate all references into that generation from older generations. Write barriers are the standard way to locate these references — at every instruction that stores a heap reference into an object, the compiler inserts code that updates an intergenerational reference data structure. This data structure enables the garbage collector to find all references from objects in older generations to objects in younger generations and use these references as roots during the collections of younger generations. The write barrier overhead has traditionally been accepted as part of the cost of using a generational collector.

This paper presents a set of new program analyses that enables the compiler to statically eliminate write barriers for instructions that never create a reference from an object in an older generation to an younger generation. The basic idea is to use pointer analysis to locate store instructions that always write the most recently allocated object. Because this object is the youngest object, such a store instruction will never create a reference from an older object to a younger object. The write barrier for this instruction is therefore superfluous and the transformation eliminates it.

We have implemented several analyses that use this basic approach to write barrier elimination:

- Intraprocedural Analysis: This analysis analyzes each method separately from all other methods. It uses a flow-sensitive, intraprocedural pointer analysis to find variables that must refer to the most recently allocated object. At method entry, the analysis conservatively assumes that no variable points to the most recently allocated object. After each method invoca-

---

1This research was supported in part by an NSF Fellowship, DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

2This analysis assumes the most recently allocated object is always allocated in the youngest generation. In some cases it may be desirable to allocate large objects in older generations. A straightforward extension of our analysis would statically identify objects that might be allocated in older generations and suppress write barrier elimination for stores that write these objects.
tion site, the analysis also conservatively assumes that no variable refers to the most recently allocated object.

- **Callee Type Extension:** This extension augments the Intraprocedural analysis with information from invoked methods. It finds variables that refer to the object most recently allocated within the currently analyzed method (the method-youngest object). It also tracks the types of objects allocated by each invoked method. For each program point, it extracts a pair \((V, T)\), where \(V\) is the set of variables that refer to the method-youngest object and \(T\) is a set of the types of objects potentially allocated by methods invoked since the method-youngest object was allocated. If a store instruction writes a reference to an object \(o\) of type \(C\) into the method-youngest object, and \(C\) is not a supertype of any type in \(T\), the transformation can eliminate the write barrier — the method-youngest object is younger than the object \(o\).

- **Caller Context Extension:** This extension augments the Intraprocedural analysis with information about the points-to information at call sites that may invoke the currently analyzed method. If the receiver object of the currently analyzed method is the most recently allocated object at all possible call sites, the algorithm can assume that the \texttt{this} variable refers to the most recently allocated object at the entry point of the currently analyzed method.

- **Full Interprocedural** This analysis combines the Callee Type Extension and the Caller Context Extension to obtain an analysis that uses both type information from callees and points-to information from callers.

Our experimental results show that, for our set of benchmark programs, the Full Interprocedural analysis is often able to eliminate a substantial number of write barriers, producing modest overall performance improvements of up to a 7% reduction in the total execution time. Moreover, by instrumenting the benchmarks to dynamically observe the age of the source and target objects at each store instruction, we are able to show that in all but two of our nine benchmarks, the analysis is able to eliminate the write barriers at virtually all of the store instructions that do not create a reference from an older object to a younger object during the execution on the default input from the benchmark suite. In other words, the analysis is basically optimal for these benchmarks. Finally, this optimality requires information from both the calling context and the called methods. Neither the Callee Type Extension nor the Caller Context Extension by itself is able to eliminate a significant number of write barriers.

This paper provides the following contributions:

- **Write Barrier Removal:** It identifies write barrier removal as an effective means of improving the performance of programs that use generational garbage collection.

- **Analysis Algorithms:** It presents several new static analysis algorithms that enable the compiler to automatically remove unnecessary write barriers. To the best of our knowledge, these are the first algorithms to use program analysis to eliminate write barriers.

- **Experimental Results:** It presents a complete set of experimental results that characterize effectiveness of the analyses on a set of benchmark programs. These results show that the Full Interprocedural analysis is able to remove the majority of the write barriers for most of the programs in our benchmark suite, producing modest performance benefits of up to a 7% reduction in the total execution time.

The remainder of this paper is structured as follows. Section 2 presents an example that illustrates how the algorithm works and how it can be used to remove unnecessary write barriers. Section 3 presents the analysis algorithms. We discuss experimental results in Section 4, related work in Section 5, and conclude in Section 6.

2. AN EXAMPLE

Figure 1 presents a binary tree construction example. In addition to the \texttt{left} and \texttt{right} fields, which implement the tree structure, each tree node also has a \texttt{depth} field that refers to an \texttt{Integer} object containing the depth of the subtree rooted at that node. In this example, the main method invokes the \texttt{buildTree} method, which calls itself recursively to create the left and right subtrees before creating the root \texttt{TreeNode}. The \texttt{linkTree} method links the left and right subtrees into the the current node, and invokes the \texttt{linkDepth} method to allocate the \texttt{Integer} object that holds the depth and link this new object into the tree.

We focus on the two store instructions generated from lines 1 and 2 in Figure 1; these store instructions link the left and
right subtrees into the receiver of the linkTree method. In the absence of any information about the relative ages of the three objects involved (the left tree node, the right tree node, and the receiver), the implementation must conserva-
vically generate write barriers at each store operation. But in this particular program, these write barriers are super-
fuous: the receiver object is always younger than the left and right tree nodes. This program is an example of a com-
mon pattern in many object-oriented programs in which the program allocates a new object, then immediately invokes a method to initialize the object. Write barriers are often unnecessary for these assignments because the object being initialized is often the most recently allocated object.2

In our example, the analysis allows the compiler to omit the unnecessary write barriers as follows. The analysis first determines that, at all call sites that invoke the linkTree method, the receiver object of linkTree is the most recently allocated object. It then analyzes the linkTree method with this information. Since no allocations occur between the entry point of the linkTree method and store instruction at line 1, the receiver object remains the most recently allocated object, so the write barrier at this store instruction can be safely removed.

In between lines 1 and 2, the linkTree method invokes the linkDepth method, which allocates a new Integer object to hold the depth. After the call to linkDepth, the receiver object is no longer the most recently allocated object. But during the analysis of the linkTree method, the algorithm tracks the types of the objects that each invoked method may create. At line 2, the analysis records the fact that the receiver referred to the most recently allocated object when the linkTree method was invoked, that the linkTree method itself has allocated no new objects so far, and that the linkDepth method called by the linkTree method allocates only integer objects. The store instruction from line 2 creates a reference from the receiver object to a TreeNode object. Because TreeNode is not a superclass of Integer, the referred TreeNode object must have existed when the linkTree method started its execution. Because the receiver was the most recently allocated object at that point, the store instruction at line 2 creates a reference to an object that is at least as old as the receiver. The write barrier at line 2 is therefore superfluous and can be safely removed.

3. THE ANALYSIS

Our analysis has the following structure: it consists of a purely intraprocedural framework, and two interprocedural extensions. The first extension, which we call the Callee Type Extension, incorporates information about called methods. The second extension, which we call the Caller Context Extension, incorporates information about the calling context. With these two extensions, which can be applied separately or in combination, we have a set of four analyses, which are given in Table 2.

Note that even for the common case of constructors that initialize a recently allocated object, the receiver of the constructor may not be the most recently allocated object — object allocation and initialization are separate operations in Java bytecode, and other object allocations may occur between when an object is allocated and when it is initialized.

<table>
<thead>
<tr>
<th></th>
<th>With Callee Type Extension</th>
<th>With Caller Context Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intraprocedural</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Callee Only</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Caller Only</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Full Interprocedural</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 2: The Four Analyses

The remainder of this section is structured as follows. We present the analysis features in Section 3.1 and the program representation in Section 3.2. In Section 3.3 we present the Intraprocedural analysis. We present the Callee Only analysis in Section 3.4, and the Caller Only analysis in Section 3.5. In Section 3.6, we present the Full Interprocedural analysis. Finally, in Section 3.7, we describe how the analysis results are used to remove unnecessary write barriers.

3.1 Analysis features

Our analyses are flow-sensitive, forward dataflow analyses that compute must points-to information at each program point. The precise nature of the computed dataflow facts depends on the analysis. In general, the analyses work with a set of variables \( V \) that must point to the object most recently allocated by the current method, and optionally a set of types \( T \) of objects allocated by invoked methods.

3.2 Program Representation

In the rest of this paper, we use \( v, v_0, v_1, \ldots \), to denote local variables, \( m, m_0, m_1, \ldots \), to denote methods, and \( C, C_0, C_1, \ldots \), to denote types. The statements that are relevant to our analyses are as follows: the object allocation statement 

\[
\text{v = NEW C,}
\]

the move statement 

\[
v_1 = v_2,
\]

and the call statement 

\[
\text{v = CALL m(v_1, \ldots, v_k).}
\]

In the given form, the first parameter to the call, \( v_1 \), points to the receiver object if the method \( m \) is an instance method.3

We assume that a preceding stage of the compiler has constructed a control flow graph for each method and a call graph for the entire program. We use \( \text{entry}_m \) to denote the entry point of the method \( m \). For each statement \( \text{st} \) in the program, \( \text{Pred(\text{st})} \) is the set of predecessors of \( \text{st} \) in the control flow graph. We use \( \text{\text{st}} \) to denote the program point immediately before \( \text{st} \), and \( \text{\text{st}} \) to denote the program point immediately after \( \text{st} \). For each such program point \( p \) (of the form \( \text{\text{st}} \) or \( \text{\text{st}} \)), we denote \( A(p) \) to be the information computed by the analysis for that program point. We use \( \text{CALLer(s)}(m) \) to denote the set of call sites that may invoke the method \( m \).

3.3 The Intraprocedural Analysis

The simplest of our set of analyses is the Intraprocedural analysis. It is a flow-sensitive, forward dataflow analysis that generates, for each program point, the set of variables that must point to the most recently allocated object, known as the \( m\text{-object} \). We call a variable that points to the \( m\text{-object} \) an \( m\text{-variable} \).

The property lattice is \( \mathcal{P}(\text{Var}) \) (the powerset of the set of

3In Java, an instance method is the same as a non-static method.
variables $\text{Var}$ with normal set inclusion as the ordering relation, where $\text{Var}$ is the set of all program variables. The meet operator used to combine dataflow facts at control-flow merge points is the usual set intersection operator: $\cap \equiv \cap$.

Figure 3 presents the transfer functions for the analysis. In the case of an allocation statement "$v = \text{NEW } C$," the new object clearly becomes the most recently allocated object. Since $v$ is the only variable pointing to this newly-allocated object, the transfer function returns the singleton $\{v\}$. For a call statement "$v = \text{CALL } m_2(v_1, \ldots, v_k)$," the transfer function returns $\emptyset$, since in the absence of any interprocedural information, the analysis must conservatively assume that the called method may allocate any number or type of objects. For a move statement "$v_1 = v_2\,$" where the source of the move, $v_2$, is an $m$-variable, the destination of the move, $v_1$, becomes an $m$-variable. The transfer function therefore returns the union of the current set of $m$-variables with the singleton $\{v\}$. For a move statement where the source of the move is not an $m$-variable, or for any other type of assignment (i.e., a load from a field or a static field), the destination of the move may not be an $m$-variable after the move. The transfer function therefore returns the current set of $m$-variables less the destination variable. Other statements leave the set of $m$-variables unchanged.

The analysis result satisfies the following equations:

$$A(\cdot st) = \begin{cases} \emptyset, & \text{if } st = \text{entry}_m \\ \cap \{A(st') \mid st' \in \text{PRED}(st)\}, & \text{otherwise} \end{cases}$$

$$A(st) = [st](A(\cdot st))$$

The first equation states that the analysis result at the program point immediately before $st$ is $\emptyset$ if $st$ is the entry point of the method; otherwise, the result is the meet of the analysis results for the program points immediately after the predecessors of $st$. As we want to compute the set of variables that definitely point to the most recently allocated object, we use the meet operator (set intersection). The second equation states that the analysis result at the program point immediately after $st$ is obtained from applying the transfer function for $st$ to the analysis result at the program point immediately before $st$.

The analysis starts with the set of $m$-variables initialized to the empty set for the entry point of method and to the full set of variables $\text{Var}$ (the top element of our property lattice) for all the other program points, and uses an iterative algorithm to compute the greatest fixed point of the aforementioned equations under subset inclusion.

### 3.4 The Callee Only Analysis

The Callee Type Extension builds upon the framework of the Intraprocedural analysis, and extends it by using information about the types of objects allocated by invoked methods.

This extension stems from the following observation. The Intraprocedural analysis loses all information at call sites because it must conservatively assume that the invoked method may allocate any number or type of objects. The Callee Type Extension allows us to retain information across a call by computing summary information about the types of the objects that the invoked methods may allocate.

To do so, the Callee Type Extension relaxes the notion of the $m$-object. In the Intraprocedural analysis, the $m$-object is simply the most recently allocated object. In the Callee Type Extension, the $m$-object is the object most recently allocated by any statement in the currently analyzed method. The analysis then computes, for each program point, a tuple $(V, T)$ containing a variable set $V$ and a type set $T$. The variable set $V$ contains the variables that point to the $m$-object (the $m$-variables), and the type set $T$ contains the types of objects that may have been allocated by methods invoked since the allocation of the $m$-object.

The property lattice is now

$$L = P(\text{Var}) \times P(\text{Types})$$

where $\text{Var}$ is the set of all program variables and $\text{Types}$ is the set of all types used by the program. The ordering relation on this lattice is

$$(V_1, T_1) \subseteq (V_2, T_2) \iff (V_1 \subseteq V_2 \land T_1 \supseteq T_2)$$

and the corresponding meet operator is

$$(V_1, T_1) \cap (V_2, T_2) = (V_1 \cap V_2 , T_1 \cap T_2)$$

The top element is $T = (\text{Var}, \emptyset)$. This lattice is in fact the cartesian product of the lattices $(P(\text{Var}), \subseteq, \cup, \cap, \text{Var}, \emptyset)$ and $(P(\text{Types}), \supseteq, \cap, \cup, \emptyset, \text{Types})$. These two lattices have different ordering relations because their elements have different meanings: $V \in P(\text{Var})$ is must information, while $T \in P(\text{Types})$ is may information.

Figure 4 presents the transfer functions for the Callee Only analysis. Except for call statements, the transfer functions treat the variable set component of the tuple in the same way as in the Intraprocedural analysis. For call statements of unanalyzable methods (for example, native methods), the transfer function produces the (very) conservative approximation $(\emptyset, \emptyset)$. For other call statements, the transfer function returns the variable set unchanged, but adds to the type set the types of objects that may be allocated during the call. Due to dynamic dispatch, the method invoked at $st$ may be one of a set of methods, which we obtain from the call graph using the auxiliary function $\text{CALLEES}(st)$. To determine the types of objects allocated by any particular method, we use another auxiliary function $\text{ALLOCATED_TYPES}$. The set of types that may be allocated during the call at $st$ is simply the union of the result of the $\text{ALLOCATED_TYPES}$ function applied to each component of the set $\text{CALLEES}(st)$. The only other transfer function that modifies the type set is the
allocation statement, which returns $\emptyset$ as the second compo-
ponent of the tuple.

The $\text{CALLEES}$ function can be obtained directly from the pro-
gram call graph, while the $\text{ALLOCATED_TYPES}$ function can be
efficiently computed using a simple flow-insensitive analysis that
determines the least fixed point for the equation given in Figure 5.

The analysis solves the dataflow equations in Figure 4 using a stan-
dard work list algorithm. It starts with the entry point of
the method initialized to $\langle \emptyset, \emptyset \rangle$ and all other pro-
gram points initialized to the top element $(\text{Var}, \emptyset)$. It com-
putes the greatest fixed point of the equations as the solution.

### 3.5 The Caller Only Analysis

The Caller Context Extension stems from the observation that
the Intraprocedural analysis has no information about
the $m$-object at the entry point of the method. The Caller
Context Extension augments this analysis to determine if
the $m$-object is always the receiver of the currently analyzed
method. If so, it analyzes the method with the $\text{this}$ variable
as an element of the set of variables $V$ that must point to
the $m$-object at the entry point of the method.

With the Caller Context Extension, the property lattice,
associated ordering relation, and meet operator are the same
as for the Intraprocedural analysis. Figure 6 presents the
additional dataflow equation that defines the dataflow result
at the entry point of each method. The equation basically
states that if the receiver object of the method is the $m$-
object at all call sites that may invoke the method, then
the $\text{this}$ variable refers to the $m$-object at the start of the
method. Note that because class (static) methods have no
receiver, $V$ is always $\emptyset$ at the start of these methods. It is
straightforward to extend this treatment to handle call sites
in which an $m$-object is passed as a parameter other than the
receiver.

Within strongly-connected components of the call graph, the
analysis uses a fixed point algorithm to compute the greatest
fixed point of the combined intraprocedural and interpro-
cedural equations. It initializes the analysis with $\{\text{this}\}$ at
each method entry point, $\text{Var}$ at all other program points
within the strongly-connected component, then iterates to a
fixed point. Between strongly-connected components, the
algorithm simply propagates the caller context information
in a top-down fashion, with each strongly-connected com-
ponent analyzed before any of the components that contain
methods that it may invoke.

### 3.6 The Full Interprocedural Analysis

The Full Interprocedural analysis combines the Callee Type
Extension and Caller Context Extension. The transfer func-
tions are the same as for the Callee Only analysis, given in
Table 4. Likewise, the property lattice, associated ordering
relation and meet operator are the same as for the Callee
Only analysis. The analysis result at the entry point of the
method, however, is subject to the equation given in Figure 7.

With this extension, the analysis will recognize that it can use
$\langle \{\text{this}\}, \emptyset \rangle$ as the analysis result at the entry point
$entry_m$ of a method $m$ if, at all call sites that may invoke
$m$, the receiver object of the method is the $m$-object and the
type set is $\emptyset$. Note that if we expand our definition of the
safe method, we can additionally propagate type set informa-
tion from the calling context into the called method.

Like the algorithm from the Caller Only analysis, the al-
gorithm for the Full Interprocedural analysis uses a fixed
point algorithm within strongly-connected components and
propagates caller context information in a top-down fashion
between components. It initializes the analysis algorithm to
compute the greatest fixed point of the dataflow equations.

### 3.7 How to Use the Analysis Results

It is easy to see how the results of the Intraprocedural analy-
sis can be used to remove unnecessary write barriers. Since an $m$-
variable must point to the most recently allocated ob-
ject, the write barrier can be removed for any store to an
object pointed to by an $m$-variable, since the reference cre-
at must point from a younger object to an older one. The
results of the Caller Only analysis are used in the same way.

It is less obvious how the analysis results are used when the
Callee Type Extension is applied, since the results now in-
clude a type set in addition to the variable set. Consider a
store of the form "$v_1.f = v_2\$," and the analysis result
$(V,T)$ computed for the program point immediately before
the store. If $v_1 \in V$, then $v_1$ must point to the $m$-object.

Any object allocated more recently than the $m$-object must
have type $C$ such that $C \in T$. If the actual (i.e., dynamic)
type of the object pointed to by $v_2$ is not included in $T$,
then the object that $v_2$ points to must be older than the
object that $v_1$ points to. The write barrier associated with
\[
\text{ALLOCATED\_TYPES}(m) = \{ v \mid v = \text{NEW } C \in m \} \cup \bigcup_{st \in m} \bigcup_{st_j \in \text{CALLERS}(st)} \text{ALLOCATED\_TYPES}(m_j)
\]

Figure 5: Equation for the ALLOCATED\_TYPES Function

\[
A(\text{entry}_m) = \begin{cases} 
\{\text{this}\} & \text{if } m \text{ is an instance method and} \\
V \text{ st } \in \text{CALLERS}(m), v_1 \in V & \text{where } V = A(st) \text{ and} \\
& \text{st is of the form } "v = \text{CALL } m(v_1, \ldots, v_k)" \\
\emptyset & \text{otherwise}
\end{cases}
\]

Figure 6: Equation for the Entry Point of a Method \( m \) for the Caller Only Analysis

the store can therefore be removed if \( v_1 \in V \), and if the type of \( v_2 \) is not an ancestor of any type in \( T \). Note that \( v_2 \not\in T \) is not a sufficient condition since the static type of \( v_2 \) may be different from its dynamic type. The analysis results are used in this way whenever the Callee Type Extension is applied (i.e., for both the Callee Only and the Full Interprocedural analyses).

4. EXPERIMENTAL RESULTS

We next present experimental results that characterize the effectiveness of our optimization. In general, the Full Interprocedural analysis is able to remove the majority of the write barriers for most of our applications. For applications that execute many write barriers per second, this optimization can deliver modest performance benefits of up to 7% of the overall execution time. There is synergistic interaction between the Callee Type Extension and the Caller Context Extension; in general, the analysis must use both extensions to remove a significant number of write barriers.

4.1 Methodology

We implemented all four of our write barrier elimination analyses in the MIT Flex compiler system, an ahead-of-time compiler for Java programs written in Java. This system, including our implemented analyses, is available under the GNU GPL at www.flexc.lcs.mit.edu. The Flex runtime uses a copying generational collector with two generations, the nursery and the tenured generation. It uses remembered sets to track pointers from the tenured generation into the nursery [18, 1]. Our remembered set implementation uses a statically allocated array to store the addresses of the created references. Each write barrier therefore executes a store into the next free element of the array and increments the pointer to that element. By manually tuning the size of the array to the characteristics of our applications, we are able to eliminate the array overflow check that would otherwise be necessary for this implementation.

We do not include results for \( \text{tsp} \) because it uses a non-deterministic, probabilistic algorithm, causing the number of write barriers executed to be vastly different in each run of the same executable. In addition, for three of the benchmarks (\( \text{bh} \), \( \text{power} \), and \( \text{treeadd} \)) we modified the benchmarks to construct the MathVector, Leaf, and TreeNode data structures, respectively, in a bottom-up instead of a top-down manner.

We present results for the following compiler options:

- **Baseline**: No optimization, all writes to the heap have associated write barriers.
4.2 Eliminated Write Barriers

Figure 8 presents the percentage of write barriers that the different analyses eliminated. There is a bar for each version of each application; this bar plots \((1 - W/W_B) \times 100\%\) where \(W\) is the number of write barriers dynamically executed in the corresponding version of program and \(W_B\) is the number of write barriers executed in the Baseline version of the program. For bh, health, perimeter, and treeadd, the Full Interprocedural analysis eliminated over 80% of the write barriers. It eliminated less than 20% only for bisort and em3d. Note the synergistic interaction that occurs when exploiting information from both the called methods and the calling context. For all applications except health, the Caller Only and Callee Only versions of the analysis are able to eliminate very few write barriers. But when combined, as in the Full Interprocedural analysis, in many cases the analysis is able to eliminate the vast majority of the write barriers.

Figure 8: Percentage Decrease in Write Barriers Executed

To evaluate the optimality of our analysis, we used the MIT Flex compiler system to produce a version of each application in which each write instruction is instrumented to determine if, during the current execution of the program, that write instruction ever creates a reference from an older object to a younger object. If the instruction ever creates such a reference, the write barrier is definitely necessary, and cannot be removed by any age-based algorithm whose goal is to eliminate write barriers associated with instructions that always create references from younger objects to older objects. There are two possibilities if the store instruction never creates a reference from an older object to a younger object: 1) Regardless of the input, the store instruction will never create a reference from an older object to a younger object. In this case, the write barrier can be statically removed. 2) Even though the store instruction did not create a reference from an older object to a younger object in the current execution, it may do so in other executions for other inputs. In this case, the write barrier cannot be statically removed.

Figure 9 presents the results of these experiments. We present one bar for each application and divide each bar into three categories:

- **Unremovable Write Barriers:** The percentage of executed write barriers from instructions that create a reference from an older object to a younger object.

- **Removed Write Barriers:** The percentage of executed write barriers that the Full Interprocedural anal-
analysis eliminates.

- **Potentially Removable:** The rest of the write barriers, i.e., the percentage of executed write barriers that the Full Interprocedural analysis failed to eliminate, but are from instructions that never create a reference from an older object to a younger object when run on our input set.

These results show that for all but two of our applications, our analysis is almost optimal in the sense that it managed to eliminate almost all of the write barriers that can be eliminated by any age-based write barrier elimination scheme.

### 4.3 Execution Times

We ran each version of each application (without instrumentation) four times, measuring the execution time of each run. The times were reproducible; see Figure 15 for the raw execution time data and the standard deviations. Figure 10 presents the mean execution time for each version of each application, with this execution time normalized to the mean execution time of the Baseline version. In general, the benefits are rather modest, with the optimization producing overall performance improvements of up to 7%. Six of the applications obtain no significant benefit from the optimization, even though the analysis managed to remove the vast majority of the write barriers in some of these applications.

Figure 11 presents the write barrier densities for the different versions of the different applications. The write barrier density is simply the number of write barriers executed per second, i.e., the number of executed write barriers divided by the execution time of the program. These numbers clearly show that to obtain significant benefits from write barrier elimination, two things must occur: 1) The Baseline version of the application must have a high write barrier density, and 2) The analysis must eliminate most of the write barriers.

### 4.4 Analysis Times

Figure 12 presents the analysis times for the different applications and analyses. We include the Full Dynamic Interprocedural analysis in this table — this version of the analysis includes callee allocated type information for call sites that (because of dynamic dispatch) have multiple potentially invoked methods. As the times indicate, including the dynamically dispatched call sites significantly increases the analysis times. Including these sites does not significantly improve the ability of the compiler to eliminate write barriers, however, since the Full Interprocedural analysis is already nearly optimal for seven out of nine of our benchmark programs.

### 4.5 Discussion

The experimental results show that, for many of our benchmark programs, our analysis is able to remove a substantial number of the write barriers. The performance improvement from removing these write barriers depends on the inherent write barrier density of the application — the larger the write barrier density, the larger the performance improvement. While the performance impact of the optimization will clearly vary based on the performance characteristics of the particular execution platform, the optimization produces modest performance increases on our platform.

By instrumenting the application to find store instructions
that create a reference from an older object to a younger object, we are able to obtain a conservative upper bound for the number of write barriers that any age-based write barrier elimination algorithm would be able to eliminate. Our results show that in all but two cases, our algorithm achieves this upper bound.

We anticipate that future analyses and transformations will focus on changing the object allocation order to expose additional opportunities to eliminate write barriers. In general, this may be a non-trivial task to automate, since it may involve hoisting allocations up several levels in the call graph and even restructuring the application to change the allocation strategy for an entire data structure.

5. RELATED WORK

There is a vast body of literature on different approaches to write barriers for generational garbage collection. Comparisons of some of these techniques can be found in [19, 12, 13]. Several researchers have investigated implementation techniques for efficient write barriers [7, 10, 11]; the goal is to reduce the write barrier overhead. We view our techniques as orthogonal and complementary: the goal of our analyses is not to reduce the time required to execute a write barrier, but to find superfluous write barriers and simply remove them from the program. To the best of our knowledge, our algorithms are the first to use program analysis to remove these unnecessary write barriers.

6. CONCLUSION

Write barrier overhead has traditionally been an unavoidable price that one pays to use generational garbage collection. But as the results in this paper show, it is possible to develop a relatively simple interprocedural algorithm that can, in many cases, eliminate most of the write barriers in the program. The key ideas are to use an intraprocedural must points-to analysis to find variables that point to the most recently allocated object, then extend the analysis with information about the types of objects allocated in invoked methods and information about the must points-to relationships in calling contexts. Incorporating these two kinds of information produces an algorithm that can often effectively eliminate virtually all of the unnecessary write barriers.

7. ACKNOWLEDGEMENTS

C. Scott Ananian implemented the Flex compiler infrastructure on which the analysis was implemented. Many thanks to Alexandru Salcianu for his help in formalizing the analyses.

8. REFERENCES


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>Intraprocedural</th>
<th>Callee Only</th>
<th>Caller Only</th>
<th>Interprocedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>8589477</td>
<td>8589477</td>
<td>8589477</td>
<td>8589477</td>
<td>1047474</td>
</tr>
<tr>
<td>bisort</td>
<td>3911959</td>
<td>3911959</td>
<td>3911959</td>
<td>3911959</td>
<td>3649763</td>
</tr>
<tr>
<td>em3d</td>
<td>836173</td>
<td>836173</td>
<td>836173</td>
<td>836173</td>
<td>820103</td>
</tr>
<tr>
<td>health</td>
<td>11971243</td>
<td>11971243</td>
<td>11971243</td>
<td>2105955</td>
<td>457489</td>
</tr>
<tr>
<td>mst</td>
<td>6344130</td>
<td>5373698</td>
<td>5373698</td>
<td>5373698</td>
<td>229964</td>
</tr>
<tr>
<td>perimeter</td>
<td>3170579</td>
<td>3170579</td>
<td>3170579</td>
<td>3170579</td>
<td>453024</td>
</tr>
<tr>
<td>power</td>
<td>23556</td>
<td>23556</td>
<td>23556</td>
<td>23556</td>
<td>13504</td>
</tr>
<tr>
<td>treeadd</td>
<td>2097309</td>
<td>2097309</td>
<td>2097309</td>
<td>2087309</td>
<td>106</td>
</tr>
<tr>
<td>voronoi</td>
<td>8426852</td>
<td>8426852</td>
<td>8426852</td>
<td>8426852</td>
<td>6266806</td>
</tr>
</tbody>
</table>

**Figure 14: Dynamic Write Barrier Counts**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>Intraprocedural</th>
<th>Callee Only</th>
<th>Caller Only</th>
<th>Interprocedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>45.8 ± 0.3</td>
<td>45.6 ± 0.2</td>
<td>45.42 ± 0.03</td>
<td>45.43 ± 0.02</td>
<td>45.13 ± 0.08</td>
</tr>
<tr>
<td>bisort</td>
<td>0.820 ± 0.007</td>
<td>0.821 ± 0.003</td>
<td>0.818 ± 0.001</td>
<td>0.819 ± 0.001</td>
<td>0.823 ± 0.003</td>
</tr>
<tr>
<td>em3d</td>
<td>1.081 ± 0.005</td>
<td>1.178 ± 0.006</td>
<td>1.107 ± 0.007</td>
<td>1.109 ± 0.009</td>
<td>1.107 ± 0.004</td>
</tr>
<tr>
<td>health</td>
<td>19.16 ± 0.02</td>
<td>19.09 ± 0.04</td>
<td>19.06 ± 0.02</td>
<td>19.39 ± 0.04</td>
<td>18.36 ± 0.02</td>
</tr>
<tr>
<td>mst</td>
<td>6.35 ± 0.01</td>
<td>6.32 ± 0.05</td>
<td>6.28 ± 0.02</td>
<td>6.25 ± 0.01</td>
<td>6.10 ± 0.02</td>
</tr>
<tr>
<td>perimeter</td>
<td>1.54 ± 0.01</td>
<td>1.54 ± 0.01</td>
<td>1.539 ± 0.008</td>
<td>1.536 ± 0.005</td>
<td>1.45 ± 0.01</td>
</tr>
<tr>
<td>power</td>
<td>7.189 ± 0.006</td>
<td>7.14 ± 0.01</td>
<td>7.137 ± 0.008</td>
<td>7.140 ± 0.003</td>
<td>7.220 ± 0.006</td>
</tr>
<tr>
<td>treeadd</td>
<td>2.19 ± 0.02</td>
<td>2.182 ± 0.002</td>
<td>2.179 ± 0.002</td>
<td>2.177 ± 0.002</td>
<td>2.079 ± 0.002</td>
</tr>
<tr>
<td>voronoi</td>
<td>10.34 ± 0.01</td>
<td>10.316 ± 0.006</td>
<td>10.280 ± 0.004</td>
<td>10.299 ± 0.007</td>
<td>10.197 ± 0.005</td>
</tr>
</tbody>
</table>

**Figure 15: Average Execution Times of Benchmark Programs**


