Write Barrier Removal by Static Analysis [*]

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

ABSTRACT

We present a new analysis for removing unnecessary write barriers in programs that use generational garbage collection. To our knowledge, this is the first static program analysis for this purpose. Our algorithm uses a pointer analysis to locate assignments that always create a reference from a younger object to an older object, then transforms the program to remove the write barriers normally associated with such assignments. We have implemented two transformations that reorder object allocations; these transformations can significantly increase the effectiveness of our write barrier removal algorithm.

Our base technique assumes that the collector promotes objects in age order. We have developed an extension that enables the optimistic removal of write barriers, with the collector lazily adding each newly promoted object into a remembered set of objects whenever the compiler may have removed write barriers involving the object at statements that have yet to execute. This mechanism enables the application of our technique to virtually any memory management system that uses write barriers to enable generational garbage collection.

Results from our implemented system show that our technique can remove substantial numbers of write barriers from the majority of the programs in our benchmark set, producing modest performance improvements of up to 6% of the overall execution time. Moreover, by dynamically instrumenting the executable, we are able to show that for six of our nine benchmark programs, our analysis is close to optimal in the sense that it removes the write barriers for almost all assignments that do not, in the observed execution, create a reference from an older object to a younger object. Finally, our results show that the overhead of our optimistic extension is negligible.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, memory management (garbage collection), optimization*

General Terms

Algorithms, Performance, Experimentation

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 such references. At every statement that stores a reference into an object, the compiler inserts code that updates an intergenerational reference data structure. This data structure enables the collector to find all references from objects in older generations to objects in younger generations and to use these references as roots for the collections of younger generations. The write barrier overhead has traditionally been accepted as part of the cost of using a generational collector.

1.1 Analysis

This paper presents a new program analysis that enables the compiler to statically remove write barriers for statements that never create a reference from an object in an older generation to an object in a younger generation. The basic idea is to use pointer analysis to locate statements that always write the most recently allocated object. Because such a statement will never create a reference from an older object to a younger object, its write barrier is superfluous and the transformation removes it. We have implemented a sequence of analyses which use this basic approach:

[*] This research was supported by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-73513, and the Singapore-MIT Alliance.

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.
Copyright 2002 ACM 1-58113-417-1/02/0011 ...$5.00.

- **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 invocation site, the analysis also conservatively assumes that no variable refers to the most recently allocated object.
- **Callee 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 $\langle \mathrm{V}, \mathrm{T} \rangle$, 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 statement 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 remove the write barrier—the method-youngest object is younger than the object `o`.
- **Caller Extension:** This extension augments the Intraprocedural analysis with points-to information from 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 `this` variable refers to the most recently allocated object at the entry point of the currently analyzed method.
- **Full Interprocedural Analysis:** This analysis combines the Callee extension and the Caller extension to obtain an analysis that uses both type information from callees and points-to information from callers.

1.1.1 Changing Object Allocation Order

To increase the effectiveness of the write barrier removal algorithm, we have implemented two transformations that change the object allocation order to match the direction of references between the objects. In situations in which the program allocates a sequence of objects in succession while creating references between the objects, the transformations change the object allocation order to ensure that references always point from younger objects to older objects.

1.1.2 Multiple Threads

In the presence of multiple threads of control, the object identified by the analysis as the most recently allocated object may not be the youngest object. Specifically, parallel threads may allocate younger objects, then store references to these younger objects into objects read by the currently analyzed method. For multithreaded programs, our analysis avoids removing write barriers for statements that may create references to such younger objects as follows.

Instead of analyzing multithreaded programs written in standard Java, our algorithm analyzes programs written in a race-free version of Java [8].[1] Race freedom guarantees that all interactions between threads are separated by explicit synchronization primitives such as lock acquire and release. In particular, for a first thread to obtain a reference to an object allocated by a parallel thread, the first thread must either execute a `monitorenter` primitive or call the `wait` method of an object between the time the parallel thread allocates the younger object and the time the first thread reads the reference to that object. Our analysis therefore conservatively assumes that after each `monitorenter` primitive or `wait` method invoked, no variable points to the most recently allocated or method-youngest object. This approach ensures that the algorithm never removes the write barrier for a statement that may create a reference to a younger object allocated in a parallel thread.

1.1.3 Allocation In Older Generations

In some circumstances, some memory management systems may allocate a new object directly in an older generation. Our technique can accommodate this action in one of two ways: 1) by suppressing write barrier removal for such objects, or 2) by putting such objects in a root set of objects that may contain references into younger generations.

1.2 Optimistic Write Barrier Removal

So far, our analyses assume that the collector promotes objects in age order. We extend our technique to work with collectors that may promote objects out of order as follows.

We first reserve a bit in the object header; this bit is set whenever the program is executing a region of code that contains statements 1) that write the object, and 2) for which the compiler has removed the associated write barriers. The generated code sets the bit when the object is created and clears the bit when all such statements have completed and execution leaves the region. When the collector promotes an object, it checks to see if its header bit is set. If so, it adds the object to a remembered set of objects that the collector scans for references to objects in younger generations at the beginning of each collection. It does not remove the object from this set until the header bit is clear. This extension ensures that if the compiler removes a write barrier that would have trapped a newly created reference from a promoted object to an object in a younger generation, the promoted object will be part of the root set of objects.

1.2.1 Low Overhead

A major advantage of this extension is that it incurs very little overhead. The header bit of an object can be set with no additional overhead when the object is initialized, while the cost of clearing the header bit is small compared to the cost of the removed write barriers. And because our analysis targets the most recently allocated object, the collector almost never promotes an object that has its header bit set. The remembered set of objects should therefore contain very few objects and the overhead of scanning these objects should be negligible. Our experimental results confirm that the overhead of this extension is very small.

1.2.2 General Multithreaded Programs

Another advantage of this extension is that it allows the application of our techniques to general multithreaded Java programs, not just safe multithreaded programs as discussed

[1] An alternative approach, the optimistic extension discussed in Section 1.2, enables the application of our technique to general multithreaded programs.

in Section 1.1.2. Even if multithreading causes the program to generate a reference from an older object to a younger object without a write barrier, the only potential problem arises when the collector promotes the older object before 1) it promotes the younger object and 2) executes the statement that creates the reference from the older object to the younger object. And in this case, the collector will add the older object to the root set, ensuring that it will recognize the intergenerational reference.

1.2.3 *Allocation in Older Generations*

Finally, the optimistic extension enables the compiler to remove write barriers for objects that may be allocated directly in an older generation when they are created. In this case, the generated code can simply set the header bit and place the object in the remembered set of objects. The effect is the same as if the object was allocated in the youngest generation, then immediately promoted. Of course, the system can also suppress write barrier removal for such objects.

1.3 Experimental Results

We have implemented our techniques in the MIT Flex system, an ahead-of-time compiler and program analysis system that compiles Java byte codes to C or native code. Our experimental results show that, for our set of benchmark programs, the combination of the Full Interprocedural analysis and the object allocation order transformations is often able to remove a substantial number of write barriers, producing modest overall performance improvements of up to a 6% 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 statement, we show that in all but three of our nine benchmarks, the analysis removes the write barriers at virtually *all* of the store statements 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. This optimality requires information from both the calling context and the called methods—neither the Callee extension nor the Caller extension by itself is able to remove a significant number of write barriers. Finally, our results show that the optimistic extension outlined in Section 1.2 imposes very little overhead.

1.4 Contributions

This paper provides the following contributions:

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

- **Allocation Order Transformations:** It presents two transformations that change the object allocation order to enhance the effectiveness of the write barrier removal algorithm.

- **An Optimistic Extension:** It presents an extension to our analysis that allows the compiler to remove write barriers even in the face of collectors that would otherwise violate the assumptions of our analysis algorithms. The underlying technique can also be used for the optimistic removal of write barriers in general; it allows the compiler to safely remove even write barriers that would otherwise be necessary for the correct execution of the program.

- **Experimental Results:** It presents a complete set of experimental results that characterize the effectiveness of the analyses on a set of benchmark programs. These results show that our technique is able to remove substantial numbers of write barriers from the majority of the programs in our benchmark suite, producing modest performance benefits of up to a 6% 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 removes unnecessary write barriers. Section 3 discusses ways to increase the precision of our analysis and alternatives to its basic approach. Section 4 presents the analysis algorithms. Section 5 presents the allocation order transformations, and Section 6 discusses the optimistic extension to our basic analysis. We discuss experimental results in Section 7, related work in Section 8, and conclude in Section 9.

2. AN EXAMPLE

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

```
class TreeNode {
    TreeNode left;
    TreeNode right;
    Integer depth;
    static public void main(String[] arg) {
        buildTree(10);
    }
    void linkDepth(int d) {
      depth = new Integer(d);
    }
    void linkTree(TreeNode l, TreeNode r, int d) {
1:      left = l;
        linkDepth(d);
2:      right = r;
    }
    static TreeNode buildTree(int d) {
        if (d <= 0) return null;
        TreeNode l = buildTree(d-1);
        TreeNode r = buildTree(d-1);
        TreeNode t = new TreeNode();
        t.linkTree(l, r, d);
        return t;
    }
}
```

Figure 1: Binary Tree Construction Example

We focus on the two store statements in lines 1 and 2 in Figure 1; these statements 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 conservatively generate write barriers for each of these statements. But in this particular program, these write barriers are superfluous: the receiver object is always younger than the left and right tree nodes. This program is an example of a common 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]

Our analysis allows the compiler to omit the unnecessary write barriers as follows. It 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 the store statement at line 1, the receiver object remains the most recently allocated object, so the compiler can safely remove the write barrier at this statement.

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 statement 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 statement 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.

2.1 The Optimistic Extension in the Example

We have so far assumed that the collector promotes objects in age order. We next consider the behavior of our example in the context of a collector that may promote objects out of order. We can apply our analysis to such collectors using an optimistic extension, which performs the following additional tasks.

The optimistic extension first identifies the TreeNode object created in the buildTree method as an object for which write barriers have been removed. It therefore initializes TreeNode objects created at that allocation site with their header bit set. It next determines where to insert instructions to clear the header bit. As the analysis does not propagate information about most recently allocated objects from called methods to callers, the compiler does not remove write barriers for the TreeNode object created in the buildTree method after the method returns. Thus, the compiler can safely insert the instructions to clear the header bit of the TreeNode object immediately after the last use of the TreeNode object, which occurs in the call to linkTree.

Now suppose that the collector promotes the receiver of the linkTree method before line 1 executes, and before it promotes the TreeNode objects referred to by local variables l and r out of the youngest generation. Because the header bit of the receiver is set, the collector adds the receiver to the remembered set of objects. At the beginning of the next collection, the collector will scan the remembered set of objects, find the references from this object to objects in the younger generations, and ensure that these references are in the root set for current and future collections. So even though the statements at lines 1 and 2 execute without write barriers, the intergenerational references that they create will be in the root set.

We expect that in almost all cases, the buildTree method will complete its execution before the collector promotes its receiver object. In this case, the collector will not add the TreeNode object to the remembered set of objects, and the only overhead of our extension is clearing the header bit. Thus, this technique enables the compiler to remove write barriers for a large variety of collectors with a performance overhead that is negligible when compared with the performance improvement due to removed write barriers.

Finally, note that the extension also works with memory management systems that may occasionally decide to allocate an object directly in an older generation when it is created. Assume, for example, that an execution of the buildTree method decides to allocate the new TreeNode object directly in an older generation. In this case, it would also add the newly created object to the remembered set of objects. The intergenerational references created in lines 1 and 2 of the linkTree method will therefore be in the root set for future collections even though the corresponding write barriers have been removed. Another alternative, of course, is to generate two versions of the code: one without write barriers (this version executes when the object is allocated in the youngest generation) and one with write barriers (this version executes when the object is allocated in an older generation).

3. ALTERNATIVES

The basic principle behind our analysis is to remove write barriers associated with statements that always create references from younger to older objects. In this section we discuss ways to increase the precision of our analysis and alternatives to its basic approach.

3.1 Increased Precision

It is possible to generalize our analysis to maintain more information about the relative order in which objects are created. Such a generalization would group the objects in the program into a statically determined number of sets, then maintain a partial order on the sets, with one set ordered before another if all objects in the first set were created before all objects in the other set. The analysis could then remove

[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.

all write barriers associated with statements that create a reference from an object in a younger set to an object in an older set. From this perspective, our current analysis maintains three sets: the set consisting of the method-youngest object, the set consisting of all objects whose class is in the type set T described in Section 4.4 (these objects may be allocated after the method-youngest object), and all other objects (these objects are all allocated before the method-youngest object).

We anticipate that the choice of object sets would determine, in large part, the effectiveness of the analysis in enabling the optimization. One alternative would group all objects with the same class or object creation site together in the same set. A more sophisticated (and potentially more effective) approach would classify objects based on their participation in different data structures, then correlate differences in data structure participation with differences in allocation order. Such an approach might be realized by reasoning about reachability properties in the various data structures of the program.

Finally, the compiler may correlate the direction of references in recursive linked data structures with the object allocation order. For example, the compiler might recognize that the parent of each subtree of a binary tree is younger than all of its children. This information could be used to remove write barriers associated with statements that create references from objects higher in the tree to objects lower in the tree. Our implemented analysis, in combination with our allocation order transformation, currently obtains the benefits of this approach for programs that create recursive data structures in which the direction of the newly created references corresponds to the order in which the program creates the objects in the data structure.

The primary reason not to increase the precision of the analysis is the potential to increase the analysis time without a corresponding increase in removed write barriers. In our benchmark set, the only significant remaining optimization opportunities available to an analysis with increased precision would require the analysis to classify objects based on participation in different data structures, then reason about the relative ages of the objects in the resulting sets.

3.2 Return Values

As currently formulated, our analysis does not attempt to recognize when a method returns the most recently allocated object. It would, however, be straightforward to augment it to do so—at call sites that invoke such methods, the analysis would simply replace the current method-youngest object with the returned object. This extension would enable the removal of write barriers for statements after the call site that create references from the returned object.

But this extension also has the potential to eliminate optimization opportunities. Consider a call site to a method that returns the most recently allocated object. With the current analysis, the method-youngest object after the call site is the same as the method-youngest object before the call site. The analysis therefore enables the removal of write barriers that involve this method-youngest object both before and after the call site. Consider what would happen if the analysis replaced the method-youngest object at the call site with the returned object. After the call site, the analysis would lose all optimization opportunities involving the method-youngest object from before the call site.

A way to eliminate this effect is to maintain multiple method-youngest objects. We anticipate that such an analysis would also maintain information about the relative allocation order of the different method-youngest objects. With this extension, the analysis would enable the removal of write barriers involving a set of recently allocated objects instead of only one recently allocated object. The only justification for implementing this extension would be increased optimization opportunities. For our benchmark set, the additional optimization opportunities are limited—only one of our benchmarks would benefit at all, and for that benchmark, the extension would enable the removal of less than 10% of the write barriers.

3.3 References From Youngest Generation

Any write barrier associated with a statement that creates a reference from an object in the youngest generation to any other object is unnecessary. We can exploit this property with a straightforward generalization of our optimistic extension. The compiler first identifies a region of code that creates multiple references from an object that is likely to be resident in the youngest generation. It then generates code that, at the beginning of the region, tests if the object is, in fact, currently allocated in the youngest generation. If so, it sets the header bit in the object and executes a version of the code that has no write barriers associated with statements that create references from the object. If not, it executes a version of the code that contains the write barriers. At the end of the region, the generated code clears the bit. Whenever the collector promotes an object with its header bit set, it adds that object to a remembered set of objects for future collections; the object remains in this set until its bit is clear.

One of the issues that affects the success of this extension is the policy that determines when to apply the transformation. Factors that affect the success of the policy include the number of write barriers in the region of code that creates references from the object and the likelihood that the object is allocated in the youngest generation.

We have implemented a specialized version of this idea whose policy focuses on newly allocated objects. The goal is to remove write barriers in the method that allocates the object and in the invoked constructor. At the allocation site, the generated code sets the header bit directly with no dynamic test (the new object will be allocated in the youngest generation). The compiler then transforms the allocating method and the invoked constructor to remove all write barriers associated with statements that create a reference from the newly allocated object. At the end of the allocating method, the generated code clears the header bit. Before applying the transformation, the compiler first checks that the number of removed write barriers justifies the overhead associated with clearing the header bit (setting the header is integrated into the object initialization in a way that imposes no additional overhead). This approach has substantially lower analysis cost than the static approach presented in this paper and is able to remove comparable numbers of write barriers. The only disadvantage is the small dynamic overhead associated with clearing the header bit. The experimental results from programs compiled with our optimistic extension, which use this mechanism, show that the overhead is negligible.

3.4 References To Oldest Generation

In some memory management systems, any write barrier associated with a statement that creates a reference to an object in the oldest generation is unnecessary. In these systems, this generation is involved only in collections of the complete heap, so there is no need to track references into the oldest generation from other generations.

One way to apply this principle is to identify a region of code that creates multiple references to an object that is likely to be allocated in the oldest generation. The generated code would then test if the object is, in fact, allocated in this generation. If so, it would execute a version of the code that has no write barriers associated with statements that create references to the object. If not, it would execute a version containing write barriers.

Once again, an important issue is determining which objects are likely to be allocated in the oldest generation. One approach is to simply allocate certain classes of objects immediately in the oldest generation when they are created, then remove the write barrier associated with any statement that creates a reference to any such object. A key issue is the policy that determines which objects should be allocated in the oldest generation when they are created.

It is possible to generalize this basic principle to remove the write barrier associated with any assignment that always creates a reference to an object allocated in a region of memory that participates only in collections of the complete heap. One alternative is to segregate the heap into two regions, only one of which participates in partial collections, then remove write barriers that always create references to objects allocated in the other region [31].

3.5 Preemptive Remembered Set Insertion

If the program repeatedly writes references into a given object, the system can simply add that object to the remembered set of objects, then remove the write barriers associated with the writes. Before applying the transformation, the compiler should check that the removed write barrier overhead is larger than the resulting scanning overhead. For objects allocated in the youngest generation, the optimistic extension is typically a better alternative, because the object would be inserted into the remembered set of objects only if it is promoted while its header bit is set.

3.6 Object Inlining and Correlated Promotion

Object inlining transforms the program to allocate a parent and child object together in the same memory space, eliminating the reference from the parent to the child object [15]. Because the program contains fewer statements that create references, a side effect of the transformation is a reduction in the number of executed write barriers.

It is also possible to discover related objects with similar lifetimes in the computation, then require the memory management system to coordinate the promotion of these objects so that they are always resident in the same generation. This property would allow the compiler to remove all write barriers associated with statements that create references between related objects. This basic approach would also enable write barrier removal even in the context of memory management approaches such as the train algorithm [22]. These approaches store objects together in units of memory called cars, and use write barriers to identify references to objects in a car.

3.7 Static Lifetime Analysis

It is also possible to enable write barrier removal by correlating object lifetimes with method invocation lifetimes. Escape analysis, for example, can be used to allocate objects on the call stack instead of on the heap [6, 7, 12, 34, 28], enabling the removal of all write barriers for statements that create references to stack-allocated objects.[3] Similar reasoning applies to systems that use region-based memory management [32].

4. THE ANALYSIS

Our analysis consists of a purely intraprocedural framework and two interprocedural extensions. The first extension incorporates information about the types of objects allocated in called methods.The second 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. Figure 2 illustrates these different analyses.

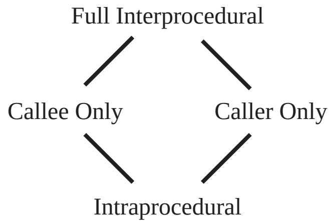

Figure 2: Different Versions of the Analysis

The remainder of this section is structured as follows. We present the analysis features in Section 4.1 and the program representation in Section 4.2. In Section 4.3 we present the Intraprocedural analysis. We present the Callee Only analysis in Section 4.4, the Caller Only analysis in Section 4.5, and the Full Interprocedural analysis in Section 4.6. In Section 4.7, we describe how the compiler uses the analysis results to remove unnecessary write barriers. In Section 4.8 we discuss how to apply our technique to partial programs.

4.1 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 V of variables that must point to the object most recently allocated by the current method, and optionally a set T of the types of objects allocated by invoked methods.

4.2 Program Representation

We use $\mathtt{v}$, $\mathtt{v}_0$, $\mathtt{v}_1$, ..., to denote local variables, $\mathtt{m}$, $\mathtt{m}_0$, $\mathtt{m}_1$, ..., to denote methods, and $\mathtt{C}$, $\mathtt{C}_0$, $\mathtt{C}_1$, ..., to denote

[3] Memory management systems that may move objects must update references from the stack-allocated object to the moved object. If the memory management system relies on the write barrier to locate such references, the compiler could not remove write barriers associated with statements that create references from stack-allocated objects. It would be possible, however, to develop stack-tracing techniques that obviate the need for these kinds of write barriers, in which case the compiler could also remove write barriers for statements that create references from stack-allocated objects.

types. Vars is the set of all local variables; Types is the set of all types. The statements that are relevant to our analyses are object allocation statements of the form $\mathtt{v} = \mathtt{NEW\ C}$, move statements of the form $\mathtt{v}_1 = \mathtt{v}_2$, call statements of the form $\mathtt{v} = \mathtt{CALL}\ \mathtt{m}(\mathtt{v}_1, \ldots, \mathtt{v}_k)$, and synchronization statements of the form `monitorenter`. In the given form, the first parameter to a call statement, $\mathtt{v}_1$, points to the receiver object if the method `m` is an instance method.[4]

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 $\mathit{entry}_{\mathtt{m}}$ to denote the entry point of the method `m`. For each statement `st`, $\textsc{pred}(\mathtt{st})$ is the set of predecessors of `st` in the control flow graph. We use $\bullet\mathtt{st}$ to denote the program point immediately before `st`, and $\mathtt{st}\bullet$ to denote the program point immediately after `st`. For each program point p, $A(p)$ is the information computed by the analysis for that program point. We use $\textsc{Callers}(\mathtt{m})$ to denote the set of call sites that may invoke the method `m`, $\textsc{Callees}(\mathtt{st})$ to denote the set of methods that may be invoked at a call site `st`, and $\textsc{Descendants}(\mathtt{C})$ to denote `C` and the types that inherit from `C` (we obtain this information from the class hierarchy).

4.3 The Intraprocedural Analysis

The Intraprocedural analysis generates, for each program point, the set of variables that must point to the most recently allocated object, which we call the *m-object.* We call a variable that points to the m-object an *m-variable.*

The property lattice is $\mathcal{P}(\text{Vars})$, the powerset of the set of all local variables. The ordering relation is: $\text{V}_1 \sqsubseteq \text{V}_2$ iff $\text{V}_1 \supseteq \text{V}_2$. The join operator used to combine dataflow facts at control flow merge points is set intersection: $\sqcup \equiv \cap$.

Figure 3 presents the transfer functions for the analysis. After an allocation statement $\mathtt{v} = \mathtt{NEW\ C}$ allocates a new object, only `v` points to the most recently allocated object. For a call statement $\mathtt{v} = \mathtt{CALL}\ \mathtt{m}_2(\mathtt{v}_1, \ldots, \mathtt{v}_k)$, the transfer function returns $\emptyset$, since in the absence of any interprocedural information, the analysis conservatively assumes that the called method may allocate any number or type of objects. For the synchronization statement `monitorenter`, the transfer function also returns $\emptyset$; in the absence of interthread information, the analysis must conservatively assume that after a synchronization statement, the currently analyzed method may obtain a reference to an object allocated by a parallel thread, and that object may be the most recently allocated object. Note that calls to `wait` are handled by the transfer function for call statements.

After a move statement $\mathtt{v}_1 = \mathtt{v}_2$, $\mathtt{v}_1$ is an m-variable if $\mathtt{v}_2$ is an m-variable. For any other type of assignment (i.e., a load or when $\mathtt{v}_2$ is not an m-variable), the destination of the move is not an m-variable after the move. Other statements leave the set of m-variables unchanged.

The analysis result satisfies the following equations:

$$\begin{aligned} A(\bullet\mathtt{st}) &= \begin{cases} \emptyset & \text{if } \mathtt{st} \equiv \mathit{entry}_{\mathtt{m}} \\ \bigsqcup_{\mathtt{st}' \in \textsc{pred}(\mathtt{st})} A(\mathtt{st}'\bullet) & \text{otherwise} \end{cases} \\ A(\mathtt{st}\bullet) &= [\![\mathtt{st}]\!](A(\bullet\mathtt{st})) \end{aligned}$$

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 join 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 set intersection as the join operator. 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`.

`st`	$[\![\mathtt{st}]\!](\text{V})$
$\mathtt{v} = \mathtt{NEW\ C}$	$\{\mathtt{v}\}$
$\mathtt{v}_1 = \mathtt{v}_2$	$\begin{cases} \text{V} \cup \{\mathtt{v}_1\} & \text{if } \mathtt{v}_2 \in \text{V} \\ \text{V} \setminus \{\mathtt{v}_1\} & \text{if } \mathtt{v}_2 \notin \text{V} \end{cases}$
$\mathtt{v} = \mathtt{CALL}\ \mathtt{m}_2(\mathtt{v}_1, \ldots, \mathtt{v}_k)$	$\emptyset$
any other assignment to `v`	$\text{V} \setminus \{\mathtt{v}\}$
`monitorenter`	$\emptyset$
other statements	V

Figure 3: Transfer Functions for the Intraprocedural Analysis

The analysis starts with the set of m-variables initialized to the empty set for the entry point of the method and to the full set of variables Vars (the bottom element of our property lattice) for all the other program points, then iterates to a fixed point.

4.4 The Callee Only Analysis

The Callee extension stems from the observation that the Intraprocedural analysis loses all information at call sites—it conservatively assumes that the invoked method may allocate any number or type of objects. The Callee extension addresses this source of imprecision by augmenting the Intraprocedural analysis to maintain information about the types of objects that invoked methods may allocate.

To do so, the Callee 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 extension, the m-object is the object most recently allocated by a statement in the currently analyzed method. The analysis then computes, for each program point, a tuple $\langle \text{V}, \text{T} \rangle$ 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

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

where Vars is the set of all local variables and Types is the set of all types used by the program. The ordering relation on this lattice is

$$\langle \text{V}_1, \text{T}_1 \rangle \sqsubseteq \langle \text{V}_2, \text{T}_2 \rangle \text{ iff } (\text{V}_1 \supseteq \text{V}_2) \wedge (\text{T}_1 \subseteq \text{T}_2)$$

and the corresponding join operator is

$$\langle \text{V}_1, \text{T}_1 \rangle \sqcup \langle \text{V}_2, \text{T}_2 \rangle = \langle \text{V}_1 \cap \text{V}_2, \text{T}_1 \cup \text{T}_2 \rangle$$

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

[4] In Java, an instance method is the same as a non-static method.

$\mathtt{st}$	$[\![\mathtt{st}]\!](\langle \mathrm{V}, \mathrm{T}\rangle)$
$\mathtt{v} = \mathtt{NEW\ C}$	$\langle \{\mathtt{v}\}, \emptyset \rangle$
$\mathtt{v}_1 = \mathtt{v}_2$	$\begin{cases} \langle \mathrm{V} \cup \{\mathtt{v}_1\}, \mathrm{T}\rangle & \text{if } \mathtt{v}_2 \in \mathrm{V} \\ \langle \mathrm{V} \setminus \{\mathtt{v}_1\}, \mathrm{T}\rangle & \text{if } \mathtt{v}_2 \notin \mathrm{V} \end{cases}$
$\mathtt{v} = \mathtt{CALL}\ \mathtt{m}_0(\mathtt{v}_1, \ldots, \mathtt{v}_k)$	$\begin{cases} \langle \emptyset, \mathrm{Types}\rangle & \text{if } \neg\textsc{Analyzable}(\mathtt{st}) \\ \langle \mathrm{V}', \mathrm{T}'\rangle & \text{otherwise} \end{cases}$ where $\mathrm{V}' = \mathrm{V} \setminus \{\mathtt{v}\}$ and $\mathrm{T}' = \mathrm{T} \cup \left(\bigcup_{\mathtt{m} \in \textsc{Callees}(\mathtt{st})} \textsc{Allocated\_Types}(\mathtt{m}) \right)$
any other assignment to $\mathtt{v}$	$\langle \mathrm{V} \setminus \{\mathtt{v}\}, \mathrm{T}\rangle$
`monitorenter`	$\langle \emptyset, \mathrm{Types}\rangle$
other statements	$\langle \mathrm{V}, \mathrm{T}\rangle$

Figure 4: Transfer Functions for the Callee Only and Full Interprocedural Analysis

$$\textsc{Allocated\_Types}(\mathtt{m}) = \{\mathtt{C} | \mathtt{v} = \mathtt{NEW\ C} \in \mathtt{m}\} \cup \left(\bigcup_{\substack{\mathtt{st}\,\in\,\mathtt{m} \\ \mathtt{st} \text{ of the form } \mathtt{v} = \mathtt{CALL}\ \mathtt{m}_0(\ldots)}} \left(\bigcup_{\mathtt{m}_1 \in \textsc{Callees}(\mathtt{st})} \textsc{Allocated\_Types}(\mathtt{m}_1) \right) \right)$$

Figure 5: Equation for the Allocated_Types Function

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 unanalyzable calls (for example, calls to native methods, calls to `wait`, or calls to methods that transitively invoke `wait` or `monitorenter`), the transfer function produces the (very) conservative approximation $\langle \emptyset, \mathrm{Types}\rangle$. For other call statements, the transfer function removes the variable assigned to the return value from the variable set (if it is in this set) and 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 Callees. To determine the types of objects allocated by any particular method, we use another auxiliary function Allocated_Types. The set of types that may be allocated during the call at `st` is simply the union of the result of the Allocated_Types function applied to each component of the set Callees(`st`). The only other transfer function that modifies the type set is the allocation statement, which returns $\emptyset$ as the second component of the tuple.

The 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 standard work list algorithm. It starts with the entry point of the method initialized to the top element $\langle \emptyset, \mathrm{Types}\rangle$ and all other program points initialized to the bottom element $\langle \mathrm{Vars}, \emptyset\rangle$, then iterates to a fixed point.

4.5 The Caller Only Analysis

The Caller 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 extension augments this analysis to determine if the m-object is always the receiver of the currently analyzed method on method entry. If so, it analyzes the method with the `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. In the Caller Only analysis, the property lattice, associated ordering relation, and join 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 states that if the receiver object of the method is the m-object at all call sites that may invoke the method, then the `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 fixed point of the combined interprocedural and intraprocedural equations. It initializes the analysis with $\{\mathtt{this}\}$ or $\emptyset$ at each method entry point, Vars 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 component analyzed before any of the components that contain methods that it may invoke.

4.6 The Full Interprocedural Analysis

The Full Interprocedural analysis combines the Callee extension and Caller extension. The transfer functions, property lattice, associated ordering relation and join operator are the same as for the Callee Only analysis. As the equation in Figure 7 indicates, the analysis uses $\langle \{\mathtt{this}\}, \emptyset\rangle$ as the analysis result at the entry point $entry_{\mathtt{m}}$ 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 we can extend this definition to additionally propagate type set information from the calling context into the called method.

$$A(\bullet \mathit{entry}_{\mathtt{m}}) = \begin{cases} \{\mathtt{this}\} & \text{if } \mathtt{m} \text{ is an instance method and} \\ & \quad \forall\ \mathtt{st} \in \textsc{Callers}(\mathtt{m}),\ \mathtt{v}_1 \in \mathrm{V} \\ & \quad\ \text{where } \mathrm{V} = A(\bullet\mathtt{st}) \text{ and} \\ & \quad\ \mathtt{st} \text{ is of the form } \mathtt{v} = \mathtt{CALL}\ \mathtt{m}(\mathtt{v}_1,\ldots,\mathtt{v}_k) \\ \emptyset & \text{otherwise} \end{cases}$$

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

$$A(\bullet \mathit{entry}_{\mathtt{m}}) = \begin{cases} \langle\{\mathtt{this}\},\emptyset\rangle & \text{if } \mathtt{m} \text{ is an instance method and} \\ & \quad \forall\ \mathtt{st} \in \textsc{Callers}(\mathtt{m}),\ \mathtt{v}_1 \in \mathrm{V}, \mathrm{T} = \emptyset \\ & \quad\ \text{where } \langle \mathrm{V},\mathrm{T}\rangle = A(\bullet\mathtt{st}) \text{ and} \\ & \quad\ \mathtt{st} \text{ is of the form } \mathtt{v} = \mathtt{CALL}\ \mathtt{m}(\mathtt{v}_1,\ldots,\mathtt{v}_k) \\ \langle\emptyset, \mathrm{Types}\rangle & \text{otherwise} \end{cases}$$

Figure 7: Equation for the Entry Point of a Method m for the Full Interprocedural Analysis

The Full Interprocedural analysis uses a fixed point algorithm within strongly-connected components and propagates caller context information in a top-down fashion between strongly connected components. It computes the least fixed point of the dataflow equations.

4.7 How to Use the Analysis Results

The compiler uses the results of the Intraprocedural and Caller Only analyses to remove unnecessary write barriers as follows. Since each m-variable must point to the most recently allocated object, the compiler removes the write barrier at all statements that use an m-variable to create a reference from the m-object to another object. Because the m-object is the youngest object, all such references point from a younger object to an older object. More precisely, the compiler removes the write barrier associated with the statement $\mathtt{st} \equiv \mathtt{v}_1.\mathtt{f} = \mathtt{v}_2$ whenever $\mathtt{v}_1 \in \mathrm{V} = A(\bullet\mathtt{st})$.

The compiler uses the results of the Callee Only and Full Interprocedural analyses as follows. Consider a statement of the form $\mathtt{v}_1.\mathtt{f} = \mathtt{v}_2$, and the analysis result $\langle \mathrm{V},\mathrm{T}\rangle$ at the program point immediately before the statement. If $\mathtt{v}_1 \in \mathrm{V}$, then $\mathtt{v}_1$ must refer to the m-object. Any object allocated more recently than the m-object must have type $\mathtt{C}$ such that $\mathtt{C} \in \mathrm{T}$. If the actual (i.e., dynamic) type of the object that $\mathtt{v}_2$ refers to is not in T, then it must be older than the object that $\mathtt{v}_1$ refers to, and the statement must create a reference from a younger object to an older object. The compiler therefore removes the write barrier associated with the statement $\mathtt{st} \equiv \mathtt{v}_1.\mathtt{f} = \mathtt{v}_2$ whenever $\mathtt{v}_1 \in \mathrm{V}$ and $\textsc{Descendants}(\mathtt{C}) \cap \mathrm{T} = \emptyset$, where $\langle \mathrm{V},\mathrm{T}\rangle = A(\bullet\mathtt{st})$ and $\mathtt{C}$ is the type of the object to which $\mathtt{v}_2$ refers.

4.8 Analyzing Partial Programs

In some situations, the compiler may not have the entire program available for analysis. We can adapt our algorithms to work effectively in this situation as follows.

We first consider the case when a potentially invoked method may be unavailable to the analysis. In this case, the Callee Only and Full Interprocedural analyses would conservatively set the analysis result for the program point immediately following the corresponding call site to $\langle\emptyset, \mathrm{Types}\rangle$.

We next consider the case when some of the callers of an analyzed method may be unavailable. In this case, the Caller Only and Full Interprocedural analyses may be unable to determine if the m-object is always the receiver of the analyzed method. The compiler can respond to such situations by generating two versions of the analyzed method: one that assumes the receiver is the m-object, and another that does not. At each call site it can examine the receiver to determine which version to invoke. Note that this mechanism may increase the optimization opportunities available to the compiler in a variety of contexts. For example, a just-in-time compiler could use this mechanism to optimize only selected call sites, while a whole-program compiler could use the mechanism to optimize only those call sites that invoke the method with the receiver as the m-object.

Finally, we address a potential problem associated with dynamic class loading. Dynamically loading a class that overrides a method in a superclass may increase the number of methods that may be invoked at a given call site. This increase may invalidate the results of the Callee Only and Full Interprocedural analyses. The way to eliminate this problem is to record the dependences of these analysis results on the properties of the class hierarchy. When the program dynamically loads a new class, the system would use these dependences to dynamically recompute the analysis results for the affected methods and recompile any invalidated code.

5. ALLOCATION TRANSFORMATIONS

We present two transformations that reorder object allocation sites to expose additional write barrier removal opportunities to our analyses. Consider the example in Figure 8. The `Dictionary` constructor allocates an object of type `java.util.HashMap`, then stores a reference to this object into the `m` field of the receiver object. The statement that creates this reference will always require a write barrier because it creates a reference from an older object to a younger object.

Figure 8 shows the example in Java source code, in which a call to a constructor appears as a single statement. But in Java bytecode, a call to a constructor consists of two statements: an allocation statement, which is invisible to the programmer, and the actual call to the constructor, for which the receiver object is implicitly the first argument. The pseudo code in Figure 9 shows the implicit allocation statements for the `Dictionary` example.

Our first transformation finds assignments in constructors that create references from the object under construc-

```
class Dictionary {
    java.util.Map m;
    static public void main(String[] arg) {
        Dictionary d = new Dictionary(10);
    }
    Dictionary(int size) {
        this.m = new java.util.HashMap(size);
    }
}
```

Figure 8: Dictionary Example

```
class Dictionary {
    java.util.Map m;
    static public void main(String[] arg) {
        tmp_0 = new <Dictionary>;
        Dictionary(tmp_0, 10);
    }
    Dictionary(int size) {
        tmp_1 = new <java.util.HashMap>;
        java.util.HashMap(tmp_1, size);
        this.m = tmp_1;
    }
}
```

Figure 9: Dictionary Allocation Statements

```
class Dictionary {
    java.util.Map m;
    static public void main(String[] arg) {
        tmp_2 = new <java.util.HashMap>;
        tmp_0 = new <Dictionary>;
        Dictionary(tmp_0, 10, tmp_2);
    }
    Dictionary(int size, Map tmp_1) {
        java.util.HashMap(tmp_1, size);
        this.m = tmp_1;
    }
}
```

Figure 10: Transformed Dictionary Program

tion to newly allocated objects. It then lifts the allocation sites for these new objects out of the constructor so that these new objects are allocated before the constructed object rather than after. In our example, the transformation lifts the `HashMap` allocation out of the `Dictionary` constructor and into `main`, to a point just prior to the allocation of the `Dictionary` object (see Figure 10). The uninitialized `HashMap` object is then passed as an additional parameter to the `Dictionary` constructor. As a result of this transformation, the assignment of field `m` in the `Dictionary` constructor no longer requires a write barrier because it creates a reference from a younger object to an older one. Moreover, because the allocation statements are not visible to the programmer, the semantics of the transformed code is unchanged from the original.

Our second transformation changes the allocation order of data structures allocated in recursive methods. Consider the example in Figure 11. In this example, the `LinkedList` constructor allocates an object of type `LinkedList`, then stores a reference to this object into the `next` field of the receiver object. This assignment will always require a write barrier because it creates a reference from an older object to a younger object. We cannot use the first transformation because the recursive call in the constructor prevents it from lifting the allocation inside the `LinkedList` constructor into its callers. Our second transformation addresses this problem by transforming the code so that it builds the `LinkedList` bottom-up instead of top-down.

The pseudo code in Figure 12 shows the implicit allocation statements for our linked list construction example; the transformation operates on the program represented at this level. The transformation first generates a new auxiliary static method as follows. Starting with the code from the constructor, the transformation pushes the allocation of the `LinkedList` object from the caller into the body of the new static method. It also adds a return statement to return the initialized `LinkedList` object to `main`. Figure 13 shows our linked list construction example after these transformation steps. Note that the transformed code is semantically equivalent to the original.

After performing these modifications, the transformation applies standard code motion techniques to reorder the allocations of the two `LinkedList` objects. Figure 14 shows the final transformed code for the `LinkedList` example. As a result of the transformation, the assignment of the `next` field no longer requires a write barrier.

This transformation also works for constructors containing multiple recursive calls, as in the case of a constructor for a binary tree.

6. THE OPTIMISTIC EXTENSION

Many generational collectors always promote objects in an order that is consistent with the allocation order. But researchers have also proposed more complicated collectors that may violate the promotion order; i.e., may promote later allocated objects before earlier allocated objects. Concurrent collectors, for example, may exhibit this behavior [16]. We have extended our technique to work with these collectors as follows.

6.1 The Algorithm

We describe how our extension works in the context of a generational collector that uses a remembered set of references to track references from objects in older generations to objects in younger generations [1]. This set is often implemented as an array of pointers to references, with the references in the array used as roots for collections of younger generations. Note that our algorithm is not limited to collectors that use this specific data structure; it generalizes to work with collectors that use arbitrary intergenerational reference data structures.

The write barrier removal algorithm presented in this paper identifies a region of the program and a recently allocated object for which it has removed write barriers over that region. Roughly speaking, each such region starts at the object allocation site and ends at the last removed write barrier for the object. We call this region the object's *write barrier removal region*. Our algorithm reserves a bit in the header of each object; this bit is set whenever the program's execution is within the object's write barrier removal region.

The algorithm also augments the collector with a remembered set of objects. It preserves the invariant that if a first object in an older generation refers to a second object in a younger generation, and that reference is not in the remembered set of references, then the first object is in the

```
class LinkedList {
    LinkedList next;
    static public void main(String[] arg) {
        LinkedList ll = new LinkedList(10);
    }
    LinkedList(int n) {
        if (n == 1) this.next = null;
        else this.next = new LinkedList(n-1);
    }
}
```

Figure 11: Linked List Example

```
class LinkedList {
    LinkedList next;
    static public void main(String[] arg) {
        tmp_0 = new <LinkedList>;
        LinkedList(tmp_0, 10);
    }
    LinkedList(int n) {
        if (n == 1) this.next = null;
        else {
            tmp_1 = new <LinkedList>;
            LinkedList(tmp_1, n-1);
            this.next = tmp_1;
        }
    }
}
```

Figure 12: Linked List Allocation Statements

```
class LinkedList {
    LinkedList next;
    static public void main(String[] arg) {
        tmp_0 = LinkedListCreator(10);
    }
    static LinkedList LinkedListCreator(int n) {
        tmp_2 = new <LinkedList>;
        if (n == 1) tmp_2.next = null;
        else {
            tmp_1 = LinkedListCreator(n-1);
            tmp_2.next = tmp_1;
        }
        return tmp_2;
    }
}
```

Figure 13: Linked List Creator

```
class LinkedList {
    LinkedList next;
    static public void main(String[] arg) {
        tmp_0 = LinkedListCreator(10);
    }
    static LinkedList LinkedListCreator(int n) {
        if (n == 1) {
            tmp_2 = new <LinkedList>;
            tmp_2.next = null;
        } else {
            tmp_1 = LinkedListCreator(n-1);
            tmp_2 = new <LinkedList>;
            tmp_2.next = tmp_1;
        }
        return tmp_2;
    }
}
```

Figure 14: Transformed Linked List Program

remembered set of objects. The algorithm maintains this invariant as follows.

At the beginning of the write barrier removal region, it initializes the object (which was just allocated in the youngest generation) with its header bit set. When execution leaves the region, it clears the header bit. When the collector promotes an object, it checks to see if the header bit is set. If so, it adds the object to the remembered set of objects.

At the start of each collection, the collector scans all of the objects in the remembered set of objects. Whenever it finds a reference from one of these objects to an object in a younger generation, it treats the reference as a root for the current collection. It also checks the header bit in the object. If the header bit is set, it leaves the object in the remembered set of objects (execution is still within the object's write barrier removal region and future statements may write references into the object without a write barrier). If the header bit is clear, it removes the object from the remembered set of objects (execution has left the object's write barrier removal region and all future statements that write a reference into the object will execute with write barriers). It also adds all references from that object to an object in a younger generation to the remembered set of references, ensuring that they will be treated as roots for future collections.

6.2 Implications

Consider several implications of this extension. First, we expect the promotion of an object with its bit set to be extremely rare because the bit is typically cleared soon after allocation. Second, the extension applies the classic systems tuning approach of making the common case faster at the cost of making the uncommon case slower [25]. Specifically, our technique eliminates the overhead of executing the (usually superfluous) write barriers associated with recently created objects. A new overhead is the need for the collector to check the bit in the header of each object that it promotes. Note that because the majority of objects die before they are promoted, we expect the number of removed write barriers to be much larger than the number of header bits that the collector tests. The experimental results presented in Section 7 show that the resulting overhead is negligible—in particular, setting and clearing bits and the garbage collector modifications that check a bit in each promoted object do not significantly increase the execution time of any of our benchmarks.

6.3 Header Bit Sets and Clears

There are two primary challenges associated with the optimistic extension: determining which sites allocate objects with removed write barriers (these sites should set the header bit in the allocated object), and determining where to place the instructions that clear the header bit. Our analysis separates the program into two kinds of regions: regions in which the control flow may lead to a removed write barrier for the current m-object (we call these regions *live regions*), and regions with no such control-flow path (we call these regions *dead regions*). The transformation inserts clear instructions at all control-flow edges that lead from an node in a live region to a node in a dead region.

There is a small complication associated with placing clear instructions in the face of interprocedural write barrier removal. To simplify the presentation, we first present an algo-

$$\begin{array}{rcl} A_t(\bullet\texttt{st}) & = & \begin{cases} \texttt{true} & \text{if } \texttt{st} \equiv \mathit{entry}_{\texttt{m}} \\ \bigsqcup_{\texttt{st}' \in \textsc{pred}(\texttt{st})} A_t(\texttt{st}'\bullet) & \text{otherwise} \end{cases} \\ A_t(\texttt{st}\bullet) & = & A_t(\bullet\texttt{st}) \wedge (\texttt{st} \text{ is not of the form } \texttt{v} = \texttt{NEW C}) \end{array}$$

Figure 15: Equations for Determining if the m-object may be the Receiver

$$\begin{array}{rcl} A_l(\texttt{st}\bullet) & = & \begin{cases} \texttt{false} & \text{if } \texttt{st} \equiv \mathit{exit}_{\texttt{m}} \\ \bigsqcup_{\texttt{st}' \in \textsc{succ}(\texttt{st})} A_l(\bullet\texttt{st}') & \text{otherwise} \end{cases} \\ A_l(\bullet\texttt{st}) & = & (A_l(\texttt{st}\bullet) \vee \textsc{use}(\texttt{st})) \wedge (\neg\textsc{def}(\texttt{st})) \\ \textsc{use}(\texttt{st}) & = & \textsc{Wb\_Removed}(\texttt{st}) \vee \textsc{Wb\_Removed\_Call}(\texttt{st}) \\ \textsc{def}(\texttt{st}) & = & (\texttt{st} \text{ is of the form } \texttt{v} = \texttt{NEW C}) \end{array}$$

Figure 16: Equations for m-object Liveness Analysis

rithm that places all clear instructions for a given m-object in the same method that allocated the object. If some of the last removed write barriers for that object occur in methods (transitively) invoked by the allocating method, our initial algorithm places the clear instructions immediately after the corresponding call sites in the allocating methods. In Section 6.3.5 we describe an extension that eliminates this restriction to, when appropriate, place the clear instructions in invoked methods. Our implemented compiler uses this extension.

6.3.1 Receiver Object Reachability

Figure 15 presents the dataflow equations for the analysis that determines if the m-object may be the receiver object. This analysis is a forward dataflow analysis that computes a single boolean value $A_t(p)$ for each program point p. This value is `true` if there is a path from the entry point of the method to p that does not contain an object allocation site.

The bit clear placement algorithm uses this analysis to ensure that it only clears the header bit of objects allocated in the current method. Specifically, if $A_t(p)$ is true and the set of of m-variables at p is non-empty, the analysis concludes that the m-variables may refer to the receiver at p. The algorithm uses this information to prevent the insertion of bit clear instructions at program points where the m-variables may refer to the receiver.

6.3.2 Finding Live Regions

The algorithm for determining where in the control flow graph to insert instructions to set and clear the header bit of objects uses a special form of liveness analysis which tracks the liveness (with respect to write barrier removal) of the current m-object. We say that the m-object is *live* at a given program point if there is a path from that point to a removed write barrier that involved that m-object. Our analysis is a flow sensitive, backward dataflow analysis that generates, for each program point p, a boolean value $A_l(p)$ that is `true` if the current m-object is live at that program point and `false` otherwise.

Figure 16 presents the dataflow equations for this analysis. The property lattice is the set {`true`, `false`}, where the top element is `true` and the bottom element is `false`. The join operator used to combine dataflow facts at control flow split points is logical or: $\sqcup \equiv \vee$. The analysis is a standard use/def analysis in which each allocation statement defines the current m-object. A store statement uses the m-object if it may write a reference into the object without executing the corresponding write barrier; a call statement uses the m-object if the (transitively) invoked method(s) may write a reference into the current m-object without executing a write barrier. The equation for the $\textsc{Wb\_Removed\_Call}$ function (see Figure 17) uses the auxiliary function $\textsc{Descendants}(\texttt{C})$, which returns the types that inherit from the type `C` (we obtain this information from the class hierarchy).

In the Caller Only and Full Interprocedural analyses, a call statement uses the current m-object if 1) the call statement passes the m-object to the invoked method as the receiver, and 2) a (potentially transitively) invoked method contains a statement that may write a reference into this object, and the write barrier associated with that statement was removed. Figure 18 presents the mutually recursive functions $\textsc{Wb\_Removed\_Call}$ and $\textsc{Wb\_Removed\_Method}$, which identify this situation. In the Intraprocedural and Callee Only analyses, the $\textsc{Wb\_Removed\_Call}$ function is simply $\textsc{Wb\_Removed\_Call}(\texttt{st}) = \texttt{false}$ because these analyses do not propagate information from the calling context into the invoked method.

6.3.3 Setting the Header Bit

Using the results of the liveness analysis, the compiler generates code that sets the header bit in objects as follows. It flags any allocation statement `st` for which $A_l(\texttt{st}\bullet)$ is `true`; i.e., for which the analysis has removed write barriers associated with stores to objects created at that allocation site. The compiler then generates code that initializes the objects allocated at those statements with their header bit set.

6.3.4 Clearing the Header Bit

The compiler uses the results of the liveness analysis to insert clear instructions as follows. It inserts an instruction to clear the header bit of the current m-object on the control flow edge from `st` to `st′` whenever $A_l(\bullet\texttt{st}) \wedge (\neg A_l(\bullet\texttt{st}')) \wedge (\neg A_t(\bullet\texttt{st}))$. In other words, the compiler inserts code to clear the header bit in the current m-object as soon as the execution completes its last use by reaching a program point in the current method where all statements with removed write barriers that store into the current m-object have executed (unless the clear instruction may clear the header bit

$$\textsc{Wb\_Removed}(\texttt{st}) \quad = \quad \begin{cases} \texttt{true} & \text{if } \texttt{st} \text{ is of the form } \texttt{v}_1.\text{f} = \texttt{v}_2 \text{ and} \\ & \quad \texttt{v}_1 \in \text{V and } \textsc{Descendants}(\texttt{C}) \cap \text{T} = \emptyset, \\ & \quad\quad \text{where } \langle \text{V}, \text{T} \rangle = A(\bullet\texttt{st}) \text{ and } \texttt{C} \text{ is the type of } \texttt{v}_2 \\ \texttt{false} & \text{otherwise} \end{cases}$$

Figure 17: Equation for the Wb_Removed **Function**

$$\begin{aligned} \textsc{Wb\_Removed\_Call}(\texttt{st}) \quad &= \quad \begin{cases} \texttt{true} & \text{if } \texttt{st} \text{ is of the form } \texttt{v} = \texttt{CALL m}(\texttt{v}_1, \ldots, \texttt{v}_k) \text{ and } \texttt{v}_1 \in \text{V and T} = \emptyset \text{ and} \\ & \quad \exists\ \texttt{m} \in \textsc{Callees}(\texttt{st}).\ \textsc{Wb\_Removed\_Method}(\texttt{m}) \\ & \quad \text{where } \langle \text{V}, \text{T} \rangle = A(\bullet\texttt{st}) \\ \texttt{false} & \text{otherwise} \end{cases} \\ \textsc{Wb\_Removed\_Method}(\texttt{m}) \quad &= \quad \exists\ \texttt{st} \in \texttt{m}.\ (\textsc{Wb\_Removed}(\texttt{st}) \vee \textsc{Wb\_Removed\_Call}(\texttt{st})) \wedge A_t(\bullet\texttt{st}) \end{aligned}$$

Figure 18: Equations for the Wb_Removed_Call **and** Wb_Removed_Method **Functions**

of the receiver object). The goal is to clear the header bit as soon as possible after the last removed write barrier.

Note that the clear instruction placement condition suppresses the insertion of the clear instruction if it may clear the header bit in the receiver object. Without this restriction, the algorithm could clear the header bit of the receiver after its last use within the current method. If the caller contained statements with removed write barriers that store into the current m-object and some of these statements execute after the current method returned, the header bit of the m-object would be clear when these statements executed, violating the correctness of the header bit clear placement algorithm.

But if the m-object may be either the receiver or an object allocated in the current method, suppressing the insertion of the clear instructions may cause the header bit of an object allocated in the current method to remain set for the rest of the execution of the program, in which case the object will remain in the remembered set of objects until it dies. We expect this case to be extremely rare in practice. One alternative is to dynamically test whether the object is the receiver or an object allocated in the current method. Another is to suppress write barrier removal for such objects. Still another is to replicate regions of the control-flow graph to eliminate the merges that cause the m-object to be either the receiver or a locally allocated object.

6.3.5 An Optimization

In our implementation, we further improve the placement of the clear instructions using the following optimization. When the last use of the current m-object occurs at a call statement—more specifically, when $A_l(\bullet\texttt{st}) \wedge (\neg A_l(\bullet\texttt{st}'))$ is $\texttt{true}$ for all $\texttt{st}' \in \textsc{succ}(\texttt{st})$, and $\texttt{st}$ is of the form $\texttt{v} = \texttt{CALL m}(\texttt{v}_1, \ldots, \texttt{v}_k)$—the compiler can further reduce the live range of the current m-object by inserting the clear instructions in the invoked method instead of after the call statement. To do so, the compiler first creates a specialized version of each of the potentially invoked methods. It then relaxes the clear instruction placement condition to allow the clear instructions to clear the header bit of the receiver in addition to objects allocated within the current method. Specifically, the compiler inserts an instruction to clear the header bit of the current m-object on the control flow edge from $\texttt{st}$ to $\texttt{st}'$ whenever $A_l(\bullet\texttt{st}) \wedge (\neg A_l(\bullet\texttt{st}'))$.

With this technique, the compiler can push the clear instructions down multiple levels of the call chain, reducing the live range of the current m-object and the probability that the garbage collector will promote the object with its header bit set.

Note that this optimization applies only to the Caller Only and Full Interprocedural analyses. Since the Intraprocedural and Callee Only analyses do not use information from callers, the placement of the clear instructions is already optimal in the sense that the algorithm could not move a clear instruction to a different method or to a different location in the control flow graph of the same method without either violating correctness of the clear placement algorithm or increasing the time between an m-object's last use and the subsequent clear of its header bit.

7. EXPERIMENTAL RESULTS

We next present experimental results that characterize the effectiveness of our optimization. For applications that execute many write barriers per second, this optimization can deliver modest performance benefits of up to 6% of the overall execution time. There is synergistic interaction between the Callee extension and the Caller extension; in general, the analysis must use both extensions to remove a significant number of write barriers.

7.1 Methodology

We implemented all four of our write barrier removal analyses, our two allocation order transformations, and the optimistic extension in the MIT Flex compiler system, an ahead-of-time compiler for programs written in Java. This system, including our implemented analyses, is available at `www.flexc.lcs.mit.edu`. The runtime uses a copying generational collector with two generations, the nursery and the tenured generation. It uses a remembered set of references to track references from the tenured generation into the nursery [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. Our write barriers are therefore somewhat more efficient than they would be in a general system designed to execute arbitrary programs with no *a priori* information about the behavior of the program.

We present results for our analysis running on the Java version of the Olden Benchmarks [10, 9]:

- **BH:** Implements the Barnes-Hut N-body solver [3].
- **BiSort:** Implements bitonic sort [5].
- **Em3d:** Models the propagation of electromagnetic waves through objects in three dimensions [14].
- **Health:** Simulates the Health-care system in Colombia [26].
- **MST:** Computes the minimum spanning tree of a graph using Bentley's algorithm [4].
- **Perimeter:** Computes the total perimeter of a region in a binary image represented by a quadtree [29].
- **Power:** Maximizes the economic efficiency of a community of power consumers [27].
- **TreeAdd:** Sums the values of the nodes in a binary tree using a recursive depth-first traversal.
- **TSP:** Solves the traveling salesman problem [24].
- **Voronoi:** Computes a Voronoi diagram for a random set of points [17].

We do not include results for 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.

We present results for the following compiler options:

- **Baseline:** No optimization, all statements that write references into the heap have associated write barriers.
- **Intraprocedural:** The Intraprocedural analysis described in Section 4.3.
- **Callee Only:** The analysis described in Section 4.4, which uses information about the types of objects allocated in invoked methods.
- **Caller Only:** The analysis described in Section 4.5, which uses information about the contexts in which the method is invoked. Specifically, the analysis determines if the receiver of the analyzed method is always the most recently allocated object and, if so, exploits this fact in the analysis of the method.
- **Full Interprocedural:** The analysis described in Section 4.6, which uses both information about the types of objects allocated in invoked methods and the contexts in which the analyzed method is invoked.
- **Optimistic Extension:** The Full Interprocedural analysis in combination with the extension described in Section 6, which interacts with the garbage collector by setting and clearing a bit in the object header to identify objects with removed write barriers.

Each of our analyses may also be used in combination with allocation order transformations.

For each application and each of the analyses, we used the MIT Flex compiler to generate two executables: an instrumented executable that counts the number of executed write barriers, and an uninstrumented executable without these counts. For all versions except the Baseline version, the compiler uses the analysis results to remove unnecessary write barriers. We then ran these executables on a 900MHz Intel Pentium-III CPU with 512MB of memory running Debian GNU/Linux 2.2. Figure 19 gives the input parameters we used for each application.

Benchmark	Input Parameters Used
BH	4096 bodies, 10 time steps
BiSort	250000 numbers
Em3d	2000 nodes, out-degree 100
Health	5 levels, 500 time steps
MST	1024 vertices
Perimeter	16 levels
Power	10000 customers
TreeAdd	20 levels
Voronoi	20000 points

Figure 19: Input Parameters Used on the Java Version of the Olden Benchmarks

7.2 Removed Write Barriers

Figures 20 and 21 present the percentage of write barriers that the different analyses removed with and without allocation order transformations, respectively. 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 the program and W_B is the number of write barriers executed in the Baseline version of the program.

Without allocation order transformations, the Full Interprocedural analysis removed over 80% of the write barriers for Perimeter, over 60% of the write barriers for MST, and over 20% of the write barriers for Voronoi, but less than 20% for BH, BiSort, Em3d, Health, Power, and TreeAdd.

With allocation order transformations, the Full Interprocedural analysis removed over 80% of the write barriers for BH, Perimeter, and TreeAdd. It removed less than 20% only for BiSort, Em3d, and Health.

Note the synergistic interaction that occurs when exploiting information from both the called methods and the calling context. For all applications except TreeAdd, the Callee Only analysis and Caller Only analysis are each able to remove very few write barriers. But when combined, as in the Full Interprocedural analysis, in many cases the analysis is able to remove the vast majority of the write barriers.

To evaluate the optimality of our analysis, we used the MIT Flex compiler system to produce a version of each application in which each store statement is instrumented to determine if, during the current execution of the program, that statement ever creates a reference from an older object to a younger object. If the statement ever creates such a reference, we say that the write barrier is *unremovable*, since it may trap a newly created reference from an object in an older generation to an object in a younger generation.[5] There are two possibilities if the statement never creates a reference from an older object to a younger object: 1) Re-

[5] Of course, there may be a valid program transformation that ensures that the statement always creates references from younger objects to older objects. And as discussed in Section 3, there may be other transformations or analyses that could enable the removal of the write barrier.

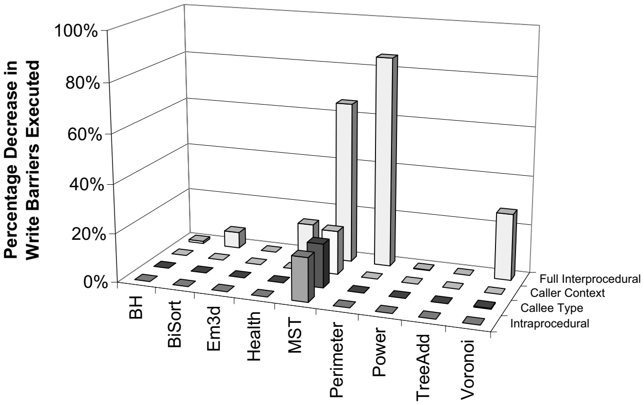

Figure 20: Percentage Decrease in Write Barriers Executed without Allocation Order Transformations

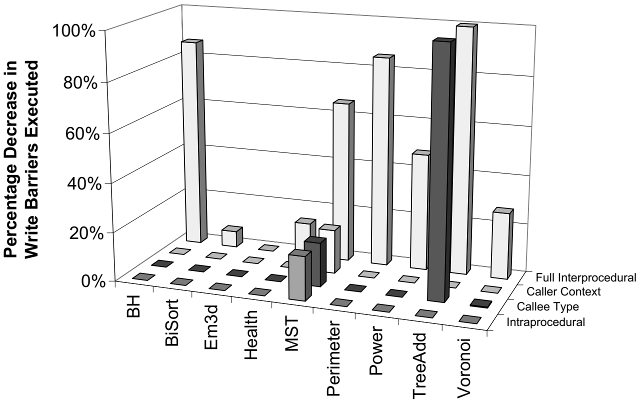

Figure 21: Percentage Decrease in Write Barriers Executed with Allocation Order Transformations

gardless of the input, the statement 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 statement 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.

Figures 22 and 23 present the results of these experiments with and without allocation order transformations, respectively. In each figure, we present one bar for each application and divide each bar into three categories:

- **Removed Write Barriers:** The proportion of executed write barriers that the Full Interprocedural analysis removes.
- **Unremovable Write Barriers:** The proportion of executed write barriers from statements that create a reference from an older object to a younger object.
- **Potentially Removable:** The rest of the write barriers; i.e., the proportion of executed write barriers that the Full Interprocedural analysis failed to remove, but are from statements that never create a reference from an older object to a younger object when run on our input set.

For all but three of our applications, our analysis is almost optimal in the sense that it removes almost all of the write barriers that can be removed by any age-based write barrier removal scheme that removes write barriers associated with statements that always create references from younger to older objects. Sections 7.2.1 through 7.2.6 discuss the reasons for these results in more detail.

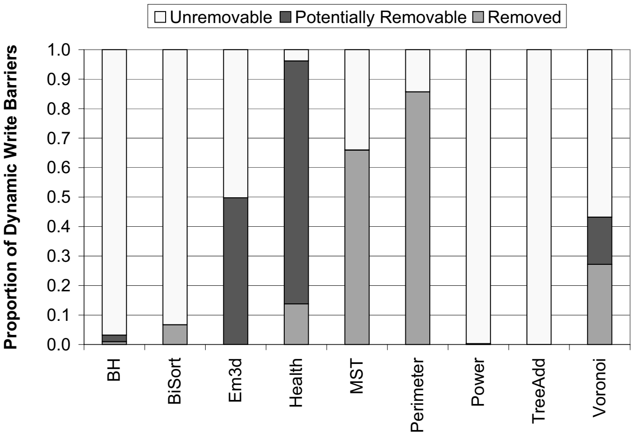

Figure 22: Write Barrier Characterization without Allocation Order Transformations

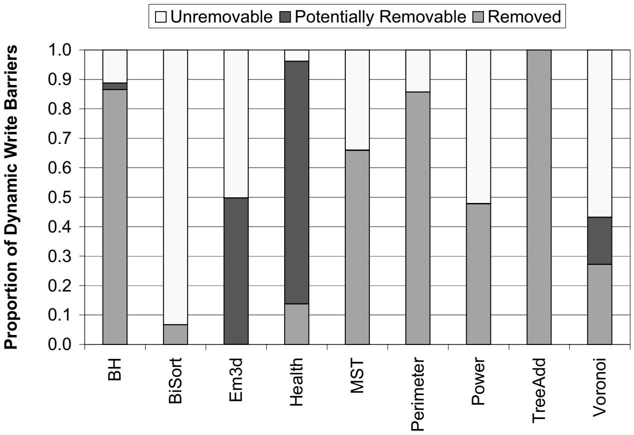

Figure 23: Write Barrier Characterization with Allocation Order Transformations

7.2.1 BH and Power

In BH and Power, the vast majority of the removed write barriers occur in constructors that execute statements of the form `this.f = new C()` to initialize fields of the object under construction to refer to newly allocated objects. Our first transformation (see Section 5) is able to lift the allocations out of the constructor and place them before the allocation of the object under construction. After this transformation, the assignments in the constructor always create references from the most recently allocated object to older objects; our algorithm is able to recognize this property and remove the corresponding write barriers. Without this transformation, virtually all of the write barriers are unremovable by any age-based scheme.

For BH, the transformation enables our analysis to remove the vast majority of the write barriers. But even after transformation, many of the write barriers in Power remain unremovable. Almost all of these unremovable write barriers occur in code that first allocates an empty array of references to objects, then executes a loop that fills in the array. All of the assignments in the array initialization loop create a reference from the array to a newly allocated object, with a write barrier clearly required. And because the array is referenced through a field of the containing object, an analysis would need to track references in the heap before it could recognize that the initialized and newly allocated arrays are in fact the same object.

7.2.2 BiSort

BiSort first builds a tree of numbers, then repeatedly restructures parts of the tree to sort the numbers. Most of the write barriers occur during the tree restructuring. Because the restructuring assignments have no correlation with the allocation order of the nodes in the program, the algorithm is unable to remove these write barriers.

7.2.3 Em3d

Almost all of the write barriers in Em3d occur in a part of the code that builds a bipartite graph. This graph is represented as a set of nodes, each of which has two arrays of references to other nodes. The vast majority of the write barriers in the Baseline version are associated with statements that fill in these arrays. As the algorithm allocates each node, it also allocates the first array for that node. After it has allocated all of the nodes, it goes back and fills in all of the first arrays. The write barriers in this section of the code are unremovable since there is no correlation between the direction of the created references (from the arrays to the nodes) and the object allocation order.

When the application finishes filling in the first arrays, it then goes back and allocates and fills in all of the second arrays. All of the write barriers in this part of the code are in principle removable since all of the corresponding statements create references from the (younger) second arrays to older graph node objects. But our technique is not designed to remove these kinds of write barriers—it reasons about write barrier removal at the granularity of individual objects, not groups of objects as this application requires.

7.2.4 Health

In Health, almost all of the write barriers occur in loops that use enumerations to traverse elements of collection data structures. The write barriers occur at statements that move the current element reference in the enumeration to the next element in the collection. Although the enumeration object is always younger than the objects in the collection, the loop using the enumeration also allocates (in the loop body) objects of the same type as those in the collection. The algorithm recognizes that the newly allocated objects are of the same type as those assigned to the current element reference of the enumeration. Since the algorithm does not currently differentiate between the newly allocated objects and the older objects of the same type in the collection being enumerated, it is unable to remove these write barriers.

7.2.5 MST, Perimeter, and Voronoi

In MST, Perimeter, and Voronoi, almost all of the removed write barriers occur in constructors that use statements of the form `this.f = p` to initialize fields of the object under construction to refer to objects passed by reference into the constructor. At all call sites, the object under construction is the most recently allocated object; the Full Interprocedural analysis recognizes this fact and removes the write barriers for the field initialization statements.

In MST, the majority of the unremovable write barriers are caused by insertions of new objects into an existing hash table. In principle, the allocation order could be changed to allocate the hash table after the objects it contains, transforming unremovable write barriers into removable ones. In practice, the transformed code would need temporary locations in which to store all the references to the objects to be put into the hash table. As hash tables usually contain many objects, this approach is not practical.

For Perimeter, the majority of the unremovable write barriers are caused by assignments that create back references from child nodes in a tree to their parent node. Note that changing the allocation order of the tree nodes would not improve the results—it would simply move the unremovable write barriers from statements that create back references to statements that create forward references from parent nodes to child nodes.

In Voronoi, most of the unremoved write barriers occur in a method that creates a reference between two edge objects. The write barriers associated with this part of the code are unremovable because the method is used in many contexts; there is no correlation between the direction of the created references and the relative ages of the edge objects involved.

The remaining unremoved write barriers in Voronoi occur in code that creates a quad edge data structure. The application first creates an array of edges and allocates edge objects with which to fill the array. The write barriers in this section of the code are unremovable because the assignments create references from the older array object to newer edge objects. The application then indexes into the edge array, and creates references between the edge objects in it. Some of the write barriers in this part of the code are in principle removable; the corresponding statements always create references from younger edge objects to older edge objects. But to identify these write barriers, an analysis must know the relative ages of objects stored in an array. Once again, our analysis is not designed to remove these kinds of write barriers—it reasons about write barrier removal at the granularity of individual objects, not groups of objects.

7.2.6 TreeAdd

In the Baseline version of TreeAdd, almost all of the write barriers occur in code that recursively builds a tree. The second transformation (see Section 5) replaces a top-down tree construction algorithm with a bottom-up tree construction algorithm, enabling the removal of virtually all of the write barriers.

7.2.7 Discussion

The majority of all write barriers in the applications occur during data structure creation, rather than during the modification of an existing data structure. Exceptions include the tree restructuring computation in BiSort and the hash table insertions in MST. Our techniques are able to remove most of the write barriers that occur during data structure construction. Exceptions include some of the write barriers associated with the construction of cyclic data structures such as the tree in Perimeter and write barriers in computations such as the graph construction in Em3d that first create, then link together, groups of objects.

Java programs have several typical patterns that any effective write barrier removal technique must successfully handle. The first pattern is an allocation of an object followed by the invocation of its constructor and the initialization of its reference fields inside the constructor. Consider an analysis that reasons about the relative ages of objects to remove write barriers associated with statements that only create references from younger objects to older objects. For such an analysis to remove the write barriers associated with object initialization, it must propagate information across procedure boundaries to recognize that the object under construction is younger than the objects to which it is initialized to refer. In some cases it may also need to transform the program to allocate the object under construction after the objects to which it will refer.

The second pattern is that Java constructors call the superclass constructor on method entry; the superclass constructor may then create new objects. Once again, some interprocedural analysis is required to recognize when these potential new object creations do not interfere with the removal of write barriers at subsequent initialization statements in the currently analyzed method. Note that for successful static write barrier removal, information must flow both from callers to callees (to recognize that the initialization of the object under construction creates references from a younger object to older objects) and from callees to callers (to recognize when objects potentially allocated by called methods do not interfere with write barrier removal).

7.3 Execution Times

We ran each version of each application (without instrumentation) five times, measuring the execution time of each run. The times were reproducible; see Figures 31 and 32 for the mean execution time data and the standard deviations. Figures 24 and 25 present the mean execution time for each version of each application normalized to the mean execution time of the Baseline version, with and without allocation order transformations, respectively. In general, the benefits are rather modest, with the optimization producing overall performance improvements of up to 6%. 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. The increased execution time for BH in Figure 25 is due to the allocation order transformation; the allocation order transformation increases the number of parameters in the transformed method, resulting in performance degradation. Although the performance improves due to write barrier removal, in this case the improvement is not large enough to overcome the overhead of the transformation.

Figure 26 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 removal, two things must occur: 1) the Baseline version of the application must have a high write barrier density, and 2) the analysis must remove most of the write barriers.

7.4 Overhead of Optimistic Extension

We have implemented the optimistic extension described in Section 6. We applied this extension to programs analyzed with the Full Interprocedural analysis, both with and without allocation order transformations. Figure 27 presents the measured increase in running times due to the extension as a percentage of the execution times. As these numbers show, the overhead is small. Negative numbers in the figure represent improvements in performance after applying the optimistic extension; we attribute these improvements to memory system effects.

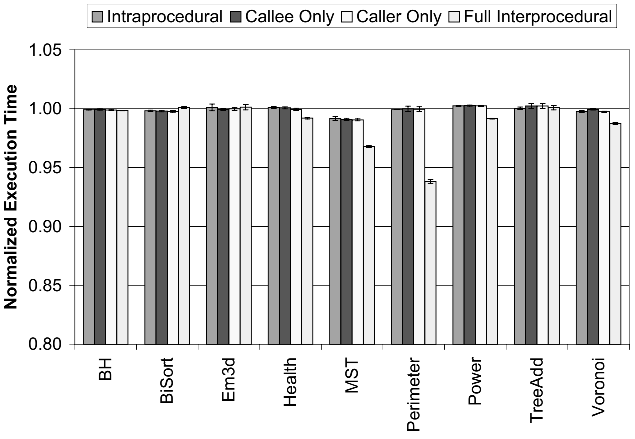

Figure 24: Normalized Execution Times without Allocation Order Transformations

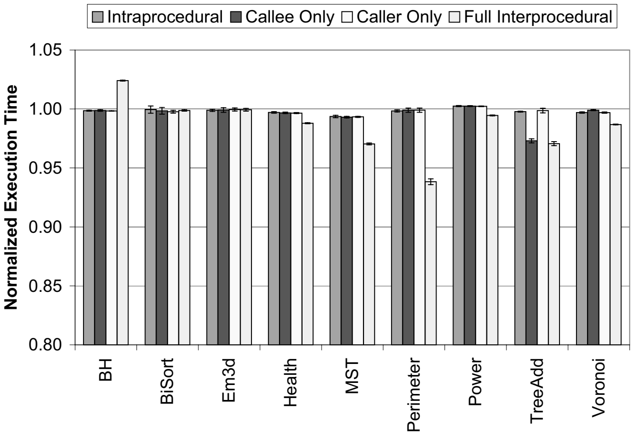

Figure 25: Normalized Execution Times with Allocation Order Transformations

7.5 Analysis Times

Figure 28 presents the analysis times for the different applications and versions of the analysis. In general, the analysis is acceptably efficient—even the Full Interprocedural analysis always completes in less than a minute.

8. RELATED WORK

There is a vast body of literature on different approaches to write barriers for generational garbage collection [33, 36, 21, 23]. Several researchers have investigated implementation techniques for efficient write barriers [11, 18, 20]; 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 re-

	Analysis Time (s)			
Benchmark	Intraprocedural	Callee Only	Caller Only	Full Interprocedural
BH	15	27	31	38
BiSort	16	27	25	34
Em3d	16	27	28	38
Health	16	24	31	38
MST	16	21	28	35
Perimeter	15	23	27	35
Power	15	26	32	38
TreeAdd	12	26	29	37
Voronoi	14	29	31	42

Figure 28: Analysis Times for Different Analysis Versions

Benchmark	Write Barrier Density (write barriers/s)
BH	913871
BiSort	5041184
Em3d	898145
Health	2452119
MST	1361658
Perimeter	2586117
Power	3720
TreeAdd	786157
Voronoi	1747947

Figure 26: Write Barrier Densities of the Baseline Version of the Benchmark Programs

	% Increase in Execution Time	
Benchmark	Without Allocation Order Transformations	With Allocation Order Transformations
BH	0.36 %	-0.95 %
BiSort	-0.33 %	-0.05 %
Em3d	0.02 %	0.17 %
Health	-0.08 %	0.37 %
MST	0.55 %	0.19 %
Perimeter	0.37 %	0.37 %
Power	0.23 %	-0.12 %
TreeAdd	0.06 %	0.22 %
Voronoi	0.09 %	-0.83 %

Figure 27: Overhead for Optimistic Extension

move them from the program. To the best of our knowledge, our algorithms are the first to use static program analysis to remove these unnecessary write barriers.

Other researchers have also identified unnecessary write barriers as a potential target for program optimization; they apply write barrier removal in the context of a type-based memory management system [31]. The basic idea is to identify "prolific" types, then allocate all instances of these types in one region of the heap, with all other objects allocated in a separate region. The collector never moves objects between regions. A minor collection involves only the prolific region, while a major collection involves the entire heap. The memory management system uses write barriers to identify all references from the non-prolific region into the prolific region; it uses these references as roots of the minor collections. With this scheme, write barriers are required only for statements that create references from non-prolific objects to prolific objects. The implemented system uses offline profiling to determine which types are prolific; it uses a type test to identify statements that create references from non-prolific objects to prolific objects.

It is possible to detect writes to objects stored in older generations by using operating systems primitives to write-protect the pages that contain objects in older generations [30, 13, 35, 2]. The advantage of this approach is that it completely eliminates the write barrier instructions (and the corresponding overhead) in the generated code. The disadvantages are the signal handling overhead associated with every write to an object stored in an older generation and the need to interact with the operating system's page protection mechanism.

The term "write barrier" has also been used in persistent object systems. In this context, a write barrier checks if a persistent object that the program intends to write is resident in memory. If it is not resident, the write barrier reads the object into memory, enabling the application to perform the write. Researchers have developed a technique that identifies regions of code that repeatedly write the same persistent object [19]. This technique inserts a write barrier at the beginning of each such region and removes all other write barriers within the region. It relies on some mechanism to ensure that the object is not written back out to stable storage while execution is within such a region.

9. 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, remove most of the write barriers in the program. The key idea is 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 remove virtually all of the unnecessary write barriers. And our optimistic extension enables the application of our techniques to generational collectors with arbitrary promotion policies.

10. ACKNOWLEDGEMENTS

We are grateful to C. Scott Ananian, who designed and built the Flex compiler infrastructure on which the analyses were implemented. Many thanks also to Alexandru Sălcianu for his help in formalizing the write barrier removal analyses, and to Michael Bedford Taylor, for his insights into the analyses comprising the optimistic extension.

11. REFERENCES

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice & Experience*, 19(2):171–183, 1989.

[2] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, 1991.

[3] J. E. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[4] Jon Louis Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of Algorithms*, 1(1):51–59, 1980.

[5] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[6] Bruno Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[7] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 35–46, 1999.

[8] Chandrasekhar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, 2001.

[9] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, 2001.

[10] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, 1995.

[11] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Languages*. PhD thesis, Stanford University, 1992.

[12] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 1999.

[13] Brian Cook. Four garbage collectors for Oberon. Undergraduate thesis, Princeton University, 1989.

[14] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 262–273, 1993.

[15] Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 345–357, 2000.

[16] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 274–284, 2000.

[17] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[18] Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA '93 Workshop on Garbage Collection and Memory Management*, 1993.

[19] Antony Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barriers for orthogonal persistence. In *Proceedings of the 8th International Workshop on Persistent Object Systems*, pages 149–159, 1998.

[20] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *OOPSLA '93 Workshop on Garbage Collection and Memory Management*, 1993.

[21] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the 6th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 92–109, 1992.

[22] Richard Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*, pages 388–403, 1992.

[23] Richard Jones and Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[24] Richard M. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224, 1977.

[25] Butler W. Lampson. Hints for computer system design. *Operating Systems Review*, 15(5):33–48, 1983.

[26] G. Lomow, J. Cleary, B. Unger, and D. West. A performance study of Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 50–55, 1988.

[27] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 240–249, 1993.

[28] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[29] Hanan Samet. Computing perimeters of regions in images represented by quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1981.

[30] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, 1987.

[31] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, pages 295–306, 2002.

[32] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[33] David Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[34] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[35] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, November 1989.

[36] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, 1990.

	Number of Times Write Barrier Executed				
					Full
Benchmark	Baseline	Intraprocedural	Callee Only	Caller Only	Interprocedural
BH	8589477	8589477	8589477	8589477	8507442
BiSort	3911959	3911959	3911959	3911959	3649764
Em3d	836173	836173	836173	836173	836102
Health	11971243	11971243	11971243	11971243	10322779
MST	6544130	5373698	5373698	5373698	2229965
Perimeter	3170579	3170579	3170579	3170579	453024
Power	23556	23556	23556	23556	23503
TreeAdd	1048733	1048733	1048733	1048733	1048680
Voronoi	8426852	8426852	8394084	8426852	6135739

Figure 29: Dynamic Write Barrier Counts without Allocation Order Transformations

	Number of Times Write Barrier Executed			
				Full
Benchmark	Intraprocedural	Callee Only	Caller Only	Interprocedural
BH	8589477	8589477	8589477	1154767
BiSort	3911959	3911959	3911959	3649764
Em3d	836173	836173	836173	836102
Health	11971243	11971243	11971243	10322097
MST	5373698	5373698	5373698	2229965
Perimeter	3170579	3170579	3170579	453024
Power	23556	23556	23556	12302
TreeAdd	1048733	159	1048733	106
Voronoi	8426852	8394084	8426852	6135739

Figure 30: Dynamic Write Barrier Counts with Allocation Order Transformations

	Average Execution Time (s) ± Standard Deviation (s)				
					Full
Benchmark	Baseline	Intraprocedural	Callee Only	Caller Only	Interprocedural
BH	9.39 ± 0.02	9.385 ± 0.002	9.387 ± 0.005	9.383 ± 0.006	9.379 ± 0.002
BiSort	0.777 ± 0.003	0.7758 ± 0.0004	0.776 ± 0.001	0.7754 ± 0.0005	0.778 ± 0.0007
Em3d	0.9314 ± 0.0005	0.932 ± 0.003	0.931 ± 0.001	0.931 ± 0.001	0.933 ± 0.002
Health	4.830 ± 0.002	4.835 ± 0.004	4.833 ± 0.004	4.827 ± 0.005	4.791 ± 0.004
MST	4.804 ± 0.005	4.765 ± 0.008	4.760 ± 0.005	4.758 ± 0.004	4.651 ± 0.004
Perimeter	1.222 ± 0.002	1.221 ± 0.000	1.222 ± 0.003	1.222 ± 0.003	1.146 ± 0.002
Power	6.349 ± 0.003	6.364 ± 0.003	6.366 ± 0.003	6.364 ± 0.003	6.296 ± 0.002
TreeAdd	1.3334 ± 0.0009	1.334 ± 0.001	1.336 ± 0.003	1.336 ± 0.003	1.335 ± 0.003
Voronoi	4.841 ± 0.003	4.829 ± 0.004	4.838 ± 0.003	4.828 ± 0.003	4.781 ± 0.004

Figure 31: Average Execution Times of Benchmark Programs without Allocation Order Transformations

	Average Execution Time (s) ± Standard Deviation (s)			
				Full
Benchmark	Intraprocedural	Callee Only	Caller Only	Interprocedural
BH	9.380 ± 0.004	9.382 ± 0.006	9.3784 ± 0.0005	9.620 ± 0.004
BiSort	0.777 ± 0.002	0.776 ± 0.002	0.7754 ± 0.0009	0.7762 ± 0.0004
Em3d	0.9304 ± 0.0009	0.931 ± 0.002	0.931 ± 0.001	0.931 ± 0.001
Health	4.816 ± 0.003	4.814 ± 0.003	4.813 ± 0.002	4.771 ± 0.002
MST	4.774 ± 0.005	4.771 ± 0.003	4.772 ± 0.002	4.662 ± 0.004
Perimeter	1.220 ± 0.001	1.221 ± 0.002	1.221 ± 0.002	1.147 ± 0.003
Power	6.365 ± 0.003	6.365 ± 0.002	6.364 ± 0.002	6.315 ± 0.003
TreeAdd	1.3304 ± 0.0005	1.297 ± 0.002	1.332 ± 0.003	1.294 ± 0.002
Voronoi	4.827 ± 0.003	4.836 ± 0.002	4.826 ± 0.003	4.778 ± 0.002

Figure 32: Average Execution Times of Benchmark Programs with Allocation Order Transformations