

Compile-Time Concurrent Marking Write Barrier Removal

David Detlefs and V. Krishna Nandivada

Compile-Time Concurrent Marking Write Barrier Removal

David Detlefs and V. Krishna Nandivada*

SMLI TR-2004-142

December 2004

Abstract:

Garbage collectors incorporating concurrent marking to cope with large live data sets and stringent pause time constraints have become common in recent years. The *snapshot-at-the-beginning* style of concurrent marking has several advantages over the *incremental update* alternative, but one main advantage: it requires the mutator to execute a significantly more expensive write barrier. This paper demonstrates that a large fraction of these write barriers are unnecessary, and may be eliminated by static analysis.

*Work done while at Sun Microsystems.



Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
david.detlefs@sun.com
nvk@cs.ucla.edu

© 2004 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java, JVM, and Java HotSpot are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>. All technical reports are available online on our Website, <http://research.sun.com/techrep/>.

Compile-Time Concurrent Marking Write Barrier Removal

V. Krishna Nandivada*
UCLA Computer Science Dept.
Los Angeles, CA 90095-1596
nvk@cs.ucla.edu

David Detlefs
Sun Microsystems Inc.
1 Network Dr.
Burlington, MA 01803-2756
david.detlefs@sun.com

Abstract

Garbage collectors incorporating concurrent marking to cope with large live data sets and stringent pause time constraints have become common in recent years. The *snapshot-at-the-beginning* style of concurrent marking has several advantages over the *incremental update* alternative, but one main disadvantage: it requires the mutator to execute a significantly more expensive write barrier. This paper demonstrates that a large fraction of these write barriers are unnecessary, and may be eliminated by static analysis.

1 Introduction

This paper presents several static analysis techniques that allow elimination of many write barriers supporting the *snapshot at the beginning* (henceforth, *SATB*) style of concurrent garbage collection [27]. This section explains why we were motivated to study this problem, and gives a quick summary of results.

The Java™ programming language is the first garbage-collected language to achieve widespread commercial popularity. This has presented extremely challenging requirements to garbage-collection implementors: some applications have multiple gigabytes of live data and must run continuously for months or years with maximum garbage collection pauses measured in milliseconds. A common solution to this problem is to employ *concurrent* collection, in which garbage collection and the user program (the *mutator*, in GC parlance) execute simultaneously [19, 20, 5]. This approach allows the live data in large heaps to be traced without long pauses.

In any concurrent garbage collector, the mutator must inform the collector of pointer updates performed during collection. The three collectors cited above are all based on the *mostly-parallel* technique of Boehm, Demers, and Shenker [6], which is classified by Wilson [26] as an example of the *incremental update* form of mutator/collector interaction. That is, the mutator informs the

*Work done while at Sun Microsystems.

collector of locations modified, and the collector re-examines them. In the alternate SATB style of concurrent marking, the collector marks the objects reachable in a logical snapshot of the object graph taken at the start of marking. The mutator helps to track this logical snapshot by noting actions that might unlink subgraphs of the original snapshot graph.

SATB marking has a major advantage over incremental-update marking. All objects allocated during marking are considered live, so only modifications to objects allocated at the start of marking need cause collection activity. In contrast, incremental-update marking must examine all modified marked objects, since any such object could hold a pointer to a subgraph that had been otherwise “hidden” from the concurrent marking process. Most programs are “mostly-functional”: a large proportion of modifications are initializations of newly-allocated objects. SATB marking can safely ignore such modifications, while incremental-update collectors must examine them. In our experimental context, pause times necessary to complete incremental-update collection are sometimes measured in hundreds of milliseconds, while SATB marking pauses for the same programs are at least an order of magnitude smaller. A number of recent concurrent collectors use the SATB style [3, 2, 10].

On the other hand, SATB marking also has a significant drawback with respect to the incremental update style: the cost of the collector/mutator interaction. In the incremental update style, a common approach is to use a *card-marking* write barrier [14], which can cost as few as two extra instructions per pointer write. SATB barriers are more expensive. In the Garbage-First collector [10] we used for our experiments, the “inline” portion of the barrier first checks whether marking is in progress. If so, it reads the pre-write value of the field, and checks whether that value is non-null; if so, it calls an out-of-line routine to add the value to a thread-local buffer. Logged values are read and processed by a concurrent marking thread. These steps require between 9 and 12 RISC instructions for each barrier. The dynamic cost depends on what fraction of the time marking is in progress, and what fraction of overwritten values are null. Section 4.5 shows measurements of the cost of the SATB barriers.

The goal of this paper is to minimize the effects of this drawback, thus allowing the strengths of SATB marking to show through. In particular, we introduce two static analyses, based on abstract interpretation, that together prove that a significant fraction of object reference writes require no SATB-related write barriers. Both identify *pre-null* writes, writes to heap locations guaranteed to contain null before the write. The first does so for fields of objects, the second for elements of (object reference) arrays. These analyses are performed after inlining.

The rest of this paper is organized as follows. Section 2 presents an analysis that identifies pre-null writes to object fields. Section 3 describes an extension that identifies pre-null writes to array elements. Section 4 presents our experiments on the efficacy of these optimizations: in some benchmarks, as many as 1/2 of all dynamically executed barriers may be eliminated. Section 4 also speculates on techniques that might eliminate further write barriers. Finally, section 5 discusses related work, and section 6 presents conclusions.

2 Eliminating Barriers for Object Field Writes

The first analysis eliminates barriers for some writes to object fields, in particular, for pre-null writes that can be proven to overwrite `null`. In the overwhelming majority of such cases, the field is null because the object has been recently allocated, and the allocator zeros fields. The write is therefore an *initializing* write.

This may seem at first glance to be a relatively simple problem. However, tracking values of fields of heap objects with precision requires tracking aliasing via a pointer analysis. Further, since we are working in a multi-threaded language, we must be aware of the possibility that objects accessible by multiple threads might be modified asynchronously: Therefore, we also perform an *escape analysis*, in the style of [25, 8], to determine when locally-allocated objects become accessible to other threads. Actually, it turns out that the write barrier problem requires greater precision than escape analysis. Generally, escape analysis wishes simply to determine whether allocation sites produce objects that *ever* escape after allocation. In contrast, we can eliminate a write barrier to an eventually-escaping object if the write occurs before the object has escaped. Tracking object escapedness at each program point requires a more precise analysis. If a compiler already implements sophisticated pointer and/or escape analysis, our techniques may be incorporated as an extension to the existing analysis.

The analysis is a flow-sensitive, intra-procedural iterative dataflow analysis (*aka* an abstract interpretation) that computes the possible reference values that might flow into fields of objects before those fields are modified by putfield instructions. In standard fashion, this pass analyzes basic blocks with modified start states, propagating changes to successor blocks, until a fixed point is reached. The analysis result allows us to identify pre-null pointer writes, and eliminate their write barriers. For clarity, we present the analysis over the well-known Java Virtual Machine (JVM) bytecode instruction set, though our actual implementation processes an internal just-in-time compiler intermediate form.

2.1 Abstract value space

In this section we describe the value space over which the analysis is defined. A *Ref* is an abstract object reference. When analyzing a method, we create two *Ref* values for each allocation site *id*. $R_{id/A}$ represents a reference to the object allocated most recently at *id*, and $R_{id/B}$ summarizes all references to objects allocated at *id* previously in the method execution. The predicate $\text{unique}(R_{id/A})$ is true (and false for all $R_{id/B}$ *Refs*), indicating that $R_{id/A}$ denotes a single concrete reference. As we shall see in section 2.4, we allow *strong update* for stores to fields of unique references. We also track the *thread-locality* of *Refs*, whether a *Ref* may be accessible to threads other than the one that allocated it.

We also create abstract reference values to denote initial values of arguments of reference types: $R_{arg(i)}$ is the initial value of argument *i*. We assume that such argument values are non-unique, because of possible aliasing with other arguments, and non-thread-local. There is an exception for

a constructor: the implicit `this` argument (i.e., $R_{arg(0)}$) of a constructor is considered unique and thread-local in the initial state, as discussed further in section 2.3.

Finally, we create a single abstract reference value $GlobalRef$ to denote all objects allocated outside the analyzed method, and not passed to it as arguments.

The analysis computes a $RefVal$, a set of possible Ref values or a bottom/uninitialized element \perp_{rv} , for each variable at each program point. A variable known to contain (only) `null` is mapped to the empty set of Ref .

Given these types, we define a *program state* as a 4-tuple of an abstract environment ρ , an operand stack stk , a non-local reference set NL , and an abstract store σ :

$$\begin{array}{llll} \rho & \subseteq & Var & \mapsto RefVal \\ stk & \subseteq & Stack < RefVal > & \\ NL & \subseteq & Ref & \\ \sigma & \subseteq & Ref \times FieldId & \mapsto RefVal \end{array}$$

The environment ρ maps local variables to the sets of possible reference values they contain. Similarly, the stk tracks the state of the JVM operand stack. Together, we will refer to ρ and stk as the *local state*. The set NL is the set of reference values known to be non-thread-local. The store σ maps reference-value/object-field identifier pairs to the set of possible reference values that the field (of the given object) may contain.

2.2 Merging states

When we process a basic block, we merge the final program state of the block into the initial state of each of the block's successors (marking them as changed if necessary). When we merge two program states, we compute the elementwise merge of the tuples. The set of local variables in the method under analysis is fixed, and bytecode verification ensures that operand stacks agree at join points, so two parts of the local state may be merged elementwise. The non-local reference set NL is merged by set union. The set of reference values and field identifiers is fixed and finite, so the range of the σ map can be merged at all points in the domain. The $RefVal$ type, which is the range of ρ , stk , and σ , forms a lattice. For any two non-bottom elements in the lattice, the meet operation is defined as the union of the two sets. The meet of any element x with \perp_{rv} is x . Non-bottom elements are ordered by the subset relation.

2.3 Initial states

As is normal in a dataflow analysis, the initial states of the maps at all program points (except for the method entry point) map everything to the bottom element of the map's range. The NL set is initialized to the singleton set $\{GlobalRef\}$. Further, all references reachable via $GlobalRef$ are collapsed into $GlobalRef$: $\forall f : \sigma(GlobalRef, f) = \{GlobalRef\}$.

It remains to define the initial state at method entry. This initial state differs in constructors and non-constructors. In non-constructors, for each non-argument local variable x of reference type, $\rho(x) = \perp_{rv}$. For each argument reference type local variable y (including the `this` variable), we set $\rho(y) = \{GlobalRef\}$. The operand stack stk is initialized to the empty stack. Constructors are treated in a similar fashion, except for the initial value of $\rho(\text{this})$, which is set to $\{R_{arg(0)}\}$. The bytecode verifier enforces the constraint that fields of a newly-allocated object may not be accessed before some constructor for the object has been executed (if some such constructor exists); further, the reference itself cannot have been stored in a global location. So on entry to a constructor for class T , we assume that $R_{arg(0)}$ is not a member of NL , and that the fields defined in type T are null for the object being constructed: $\forall f : \sigma(R_{arg(0)}, f) = \{\}$.

2.4 Effects of operations

Below we show the effect of several relevant bytecode instructions, by showing the state tuple resulting from the start state $\langle \rho, \sigma, NL, stk \rangle$. We use the notation $[stk : \overline{v_i}]$ to denote pushing the vector $\overline{v_i} = v_0, \dots, v_n$ onto the stack stk .

The `load` and `store` instructions access and update local variables. The `getstatic` and `putstatic` instructions perform corresponding operations on static fields. Reference values stored into static variables “escape.” The `getstatic` instruction always returns $\{GlobalRef\}$, and storing a reference value via `putstatic` causes it (and any reference reachable from it) to escape. This is the purpose of $AllNonTL(NL, RS, \sigma)$, which returns the result of extending the non-locality set NL with the *Ref* set RS and all *Refs* (transitively) reachable (in σ) via fields of references in RS .

The `getField` and `putfield` instructions operate on fields of heap objects. The `getField` bytecode retrieves the value of a field — note that both the *obj* reference and the result of a field lookup are sets, so the result is the union of the lookup result for each member of *obj*. The function $lookup(\sigma, r, NL, f)$ is defined by

$$\text{if } r \in NL \text{ then } \{GlobalRef\} \text{ else } \sigma(r, f) \quad .$$

The interpretation of `putfield` is complicated by the distinction between *strong* and *weak* update semantics. If the `putfield` updates a field of a single abstract reference value that is unique (denotes a single runtime value), then strong update can be used. Otherwise, the new *val* is merged into the previous contents via set union. Storing a reference value into a heap location via `putfield` may also cause the value to escape, if the object into which the value is stored is itself possibly non-thread-local. The function $AllNonTLCond(NL, RS, val, \sigma)$ is used to update the non-locality set appropriately; this returns NL if the intersection of RS and NL is empty, and otherwise returns the result of extending NL with *Refs* transitively reachable from *vals* via σ .

Somewhat similarly, passing a reference value as an argument to a method (via the `invoke` instruction) may cause it to escape, so our handling of `invoke` updates the non-locality set via the function $nAllNonTL(NL, \overline{v_i}, \sigma)$, which is simply the union of $AllNonTL(NL, v_i, \sigma)$ for all the v_i in $\overline{v_i}$.

$\langle \rho, \sigma, NL, stk \rangle$ $\text{load}(x) \implies$ $\langle \rho, \sigma, NL, [stk : \rho(x)] \rangle$
$\langle \rho, \sigma, NL, [stk : val] \rangle$ $\text{store}(x) \implies$ $\langle \rho[x \leftarrow val], \sigma, NL, stk \rangle$
$\langle \rho, \sigma, NL, stk \rangle$ $\text{getstatic}(f) \implies$ $\langle \rho, \sigma, NL, [stk : \{GlobalRef\}] \rangle$
$\langle \rho, \sigma, NL, [stk : val] \rangle$ $\text{putstatic}(f) \implies$ $\langle \rho, \sigma, \text{AllNonTL}(NL, val, \sigma), stk \rangle$
$\langle \rho, \sigma, NL, [stk : obj] \rangle$ $\text{getfield}(f) \implies$ $\langle \rho, \sigma, NL, [stk : \bigcup_{ot \in obj} \text{lookup}(\sigma, ot, NL, f)] \rangle$
$\langle \rho, \sigma, NL, [stk : obj, val] \rangle$ $\text{putfield}(f) \implies$ if $obj = \{r\} \wedge \text{unique}(r)$: $\langle \rho, \sigma[(r, f) \leftarrow val], \text{AllNonTLCond}(NL, obj, val, \sigma), stk \rangle$ else: $\langle \rho, \sigma[\forall ot \in obj : (ot, f) \leftarrow \sigma(ot, f) \cup val], \text{AllNonTLCond}(NL, obj, val, \sigma), stk \rangle$
$\langle \rho, \sigma, NL, [stk : \bar{v}_i] \rangle$ $\text{invoke}(m(\bar{a}_i) : T) \implies$ if T is a reference type: $\langle \rho, \sigma, \text{nAllNonTL}(NL, \bar{v}_i, \sigma), [stk : \{GlobalRef\}] \rangle$ else: $\langle \rho, \sigma, \text{nAllNonTL}(NL, \bar{v}_i, \sigma), stk \rangle$

The functions `AllNonTL`, `AllNonTLCond`, and `nAllNonTL` are defined more formally in appendix A.

Our analysis is performed after inlined method bodies are expanded, since this conservative treatment of arguments of non-inlined methods (and our current lack of inter-procedural techniques) is detrimental to the precision of the analysis. For example, every allocation is followed by the invocation of a constructor on the allocation's result: if the constructor is not inlined, the allocated object is considered to escape immediately. (Fortunately, constructors are usually simple enough to be inlined, and we have not found this to be a practical problem.)

Processing an `invoke` instruction also pushes $\{GlobalRef\}$ on the operand stack if the method has a reference return type.

We must also consider reads and writes to object arrays, since these provide another mechanism by

which references may escape a thread. For this purpose, we treat an object array as an object with a single field f_{elems} that “collapses” all elements of the array. Because of this, all array updates are treated as weak updates.

$\begin{aligned} &< \rho, \sigma, NL, [stk : arr, ind] > \\ &\quad aaload \implies \\ &< \rho, \sigma, NL, [stk : \bigcup_{at \in arr} \sigma(at, f_{elems})] > \end{aligned}$
$\begin{aligned} &< \rho, \sigma, NL, [stk : arr, ind, val] > \\ &\quad aastore \implies \\ &< \rho, \sigma[\forall at \in arr : (at, f_{elems}) \leftarrow \sigma(at, f_{elems}) \cup val], \\ &\quad AllNonTLCCond(NL, arr, val, \sigma), stk > \end{aligned}$

Next we consider the effects of the `newinstance` instruction, which allocates a new object. Recall that we associate two abstract reference values with an allocation site at instruction id : one ($R_{id/A}$) to denote the most recently allocated object, and another ($R_{id/B}$) to denote all previously allocated objects. An allocation at the site will associate $R_{id/A}$ with a newly allocated object, but first it must merge attributes previously associated with $R_{id/A}$ into $R_{id/B}$, since it becomes part of the set of previously allocated objects.

$\begin{aligned} &< \rho, \sigma, NL, stk > \\ &\quad id : newinstance(cl) \implies \\ &< rngSubst(\rho, R_{id/A} \leftarrow R_{id/B}), \\ &\quad transfer(\sigma, R_{id/A}, R_{id/B}), \\ &\quad replS(NL, R_{id/A} \leftarrow R_{id/B}), \\ &\quad [rngSubst(stk, R_{id/A} \leftarrow R_{id/B}) : \{R_{id/A}\}] > \end{aligned}$
--

We accomplish the removal of $R_{id/A}$ as follows. To update NL , we use `replS`, which returns a set like NL but with $R_{id/A}$ removed, and with $R_{id/B}$ added if $R_{id/A}$ had originally been a member. We use `rngSubst` to perform similar replacement in the ranges of the local state maps ρ and stk (treating stk as a map from integers to values). Updating the heap state σ is somewhat more subtle. The expression `transfer`(σ, r_1, r_2) returns an abstract store that is like σ , except that r_1 has been removed from the domain, any entries for fields of r_1 have been merged into the corresponding fields of r_2 , and r_2 has been substituted for r_1 in all elements of the range of store. Appendix A gives formal definitions for these functions.

Finally, $R_{id/A}$ is pushed on the operand stack as the result of the allocation instruction.

This use of two reference values for each allocation site is the way in which our analysis is more precise than “traditional” escape analysis. To motivate the need for this extra precision, consider the following example (`p1` and `p2` are arbitrary predicates):

```

class Foo { public String s; }
Foo f1 = null; Foo f2 = null;
while (p1) {
    f1 = new Foo();           // F1
    f1.s = "hi1";             // W1
    if (p2) f2 = new Foo();   // F2
    f2.s = "hi2";             // W2
}

```

If we denote all values produced at an allocation site with a single name, then we must use weak update: if we used strong update, we'd improperly “prove” that no barrier is necessary at W2. If we use weak update, however, we will not be able to prove that the barrier at W1 is unnecessary. Reserving a name for the most-recently-allocated object allows assignments to fields of that object to use strong update. The use of two names per allocation site is suggested, but not explored, by Whaley and Rinard [25]. Corbett [9] also uses this technique, but only for allocation sites that occur within loops. Most-recently-allocated nodes are merged into summary nodes at the end of loops. In contrast, our method of merging as a side effect of allocation avoids any need to identify loops. Finally, the shape analysis of Sagiv *et al.* [22] gets similar precision via different means.

As is normal in an abstract interpretation, we iterate until there are no statements whose input states have changed. Any change is by virtue of a merge; since we are working over finite lattices (there are a finite number of abstract references, bound by the program size), it is possible to bound the worst-case execution time as $O(n^5)$. Tighter bounds may be possible, and in practice, performance is much better than this bound might suggest: this bound calculation considers the number of local variables, maximum operand stack size, and number of distinct *Ref* values all as $O(n)$, when they are typically small fractions of n . Section 4.4 gives data on the variation of analysis time with code size.

When we process `putfield(f)` instructions, we also note whether the instruction requires a write barrier: if the pre-instruction state $\langle \rho, \sigma, NL, [stk : o] \rangle$ has the property that $\forall ot \in o : ot \notin NL \wedge \sigma(ot, f) = \{\}$, then the SATB write barrier may be omitted. The last such judgment (at the fixed point of the analysis) is correct.

3 Eliminating Barriers for Array Element Writes

The Java language specification states that a newly allocated array of an object type has all elements set to null. Therefore, just as with object field writes, the first (initializing) writes to such array elements do not require SATB barriers. However, proving that an array element write is initializing is somewhat more involved than proving the corresponding property for field writes.

The rest of this section describes an extension to the previous analysis that proves that writes to array elements are pre-null. The analysis uses abstract interpretation to infer linear relationships between integer state components.

3.1 Motivating array example

We first show a simple motivating example. Consider the following method:

```
public static T[] expand(T[] ta) {
    T[] new_ta = new T[ta.length*2];
    for (int i = 0; i < ta.length; i++)
        new_ta[i] = ta[i];
    return new_ta;
}
```

All the writes to the array variable `new_ta` in the `for` loop of the above example are initializing writes. In the benchmarks we studied, a significant number of array writes are initializing writes in loops similar to this. Therefore, the aspirations of the analysis we describe in this section do not extend far beyond such simple examples.

Eliding the SATB barriers from the stores in the `for` loop above requires inference of the following loop invariant:

$$\forall j : i \leq j < \text{new_ta.length} : \text{new_ta}[j] = \text{null} \quad .$$

We accomplish this inference by tracking the uninitialized portion of each array, and also by tracking the values of integer local variables. The next section shows how we do this formally.

3.2 Analysis extensions for arrays

This section details how we extend the previous analysis to track object array stores. We modify some of the previous state components, and add some new ones:

<i>Value</i>	:	<i>Refs</i> <i>IntVal</i> \perp
ρ	\subseteq <i>Var</i>	\mapsto <i>Value</i>
σ	\subseteq <i>Ref</i> \times <i>FieldId</i>	\mapsto <i>Value</i>
<i>stk</i>	\subseteq <i>Stack</i> < <i>Value</i> >	
<i>Len</i>	\subseteq <i>Ref</i>	\mapsto <i>IntVal</i>
<i>NR</i>	\subseteq <i>Ref</i>	\mapsto <i>IntRange</i>

We extend the abstract state by tracking integer, as well as reference, values; this is reflected in the redefinition of the ranges of ρ , σ , and the *stk* stack.

An *IntVal* is a linear combination of integer terms. These terms may be integer constants or the product of an integer coefficient and an integer *unknown*. An unknown may be *constant* (have the same value in all states; denoted c_i), or *variable* (may represent different values in different states; denoted v_i). We allow *IntVals* to have at most one term in a variable unknown, one constant term, and zero or more terms in constant unknowns: $au + k_0c_0 + \dots + k_nc_n + b$. We perform

symbolic arithmetic on *IntVals* when it makes sense, but certain operations (for example, addition of *IntVals* involving different variable unknowns) produce the top value \top_{iv} as their result. Obviously, operations where one of the elements is \top_{iv} yield \top_{iv} . Method calls with integer return types return \top_{iv} , since different invocations might return different values. An *IntRange* represents a subsequence of the sequence of valid indices of an array. There are several kinds of *IntRange*. A *full IntRange* is a (closed) integer interval, bounded by two *IntVals*: $[iv_1..iv_2]$. This is used only to represent an array's uninitialized indices immediately after the array's allocation: it is the smallest non-bottom element of the range lattice for the given array. There are two varieties of *half-open* ranges: $[iv..]$ denotes the sequence of indices i of r such that $i \geq iv$, and a $[..iv]$ denotes the sequence of indices i of r such that $i \leq iv$. The lattice ordering on ranges obeys the following rules: $[i..j]$ is below $[i..]$ and $[..j]$; $[i..]$ is below $[j..]$ if $i < j$ and $[..i]$ is below $[..j]$ if $j < i$ (smaller ranges are larger in the lattice). The *empty* range is denoted $[]$, and is the top element of the range lattice.

The new map *Len* maps references to arrays to their lengths. The map *NR* (for *null range*) maps object array references to *IntRanges* representing subranges of their valid indices known to be `null`.

As discussed, the various ranges here all form lattices; it should be fairly straightforward to see how these lattices merge. The overall merge operation on states is somewhat more complicated than a simple merge of all the state components, however; this is detailed later, in section 3.5.

3.3 Effects of operations in array analysis

We now show the effect of some operations on the new state components.

$$\begin{array}{l}
\langle \rho, \sigma, NL, [stk : n], Len, NR \rangle \\
id : \text{newarray}(cl) \implies \\
\langle \text{rngSubst}(\rho, R_{id/A} \leftarrow R_{id/B}), \\
\text{transfer}(\sigma, R_{id/A}, R_{id/B}), \\
\text{replS}(NL, R_{id/A} \leftarrow R_{id/B}), \\
[\text{rngSubst}(stk, R_{id/A} \leftarrow R_{id/B}) : \{R_{id/A}\}], \\
Len[R_{id/A} \leftarrow n], \\
NR[R_{id/A} \leftarrow [0, n - 1]] \rangle
\end{array}$$

The `newarray` instruction is very similar to `newinstance`: it updates the first four state components in the same way. In addition, it records the length of the newly allocated array, and notes that the entire range of valid indices currently maps to `null`.

Next we reconsider the `aastore` instruction, which writes to an element of an object array.

$$\begin{array}{l}
\langle \rho, \sigma, NL, [stk : arr, ind, val], Len, NR \rangle \\
aastore \implies \\
\langle \rho, \sigma[\forall at \in arr : (at, f_{elems}) \leftarrow \sigma(at, f_{elems}) \cup val], \\
\text{AllNonTLCond}(NL, arr, val, \sigma), stk, Len, \\
NR[\forall at \in arr : at \leftarrow \text{contract}(NR(at), ind)] \rangle
\end{array}$$

Again, the first four state components are updated as before. Additionally, the NR map is updated to reflect that the `astore` may have written into the null range of the array, causing it to “contract.” The contract function embodies a set of simple heuristics, essentially recognizing stores at either end of the uninitialized range:

$$\begin{aligned} \text{contract}([au + b..x], au + b) &= [au + (b + 1)..] \\ \text{contract}([x..au + b], au + b) &= [..au + (b - 1)] \\ \text{otherwise : } \text{contract}([x..y], i) &= [] \end{aligned}$$

(In the rules above, $[x..y]$ is also intended to match the half-open ranges $[z..]$ and $[..z]$.)

3.4 Initial conditions in array analysis

We now discuss initial conditions. We have previously mentioned (constant and variable) integer unknowns, without defining what gives rise to such unknowns. We create a constant unknown c_i for each integer input parameter i , and set $\rho(i) = c_i$ in the initial state. We also create a distinct constant unknown c_i to represent the length of every input parameter of array type, and record this equality in the Len map: $Len(R_{arg(i)}) = c_i$. As mentioned previously, method calls with integer return types do *not* create constant unknowns (since they might be called several times in a concrete execution, returning different results); instead, they are modeled as returning \top_{iv} . We sometimes create new variable unknowns at merge points, according to rules described in the next section. The number of such variables is limited to the number of merge points times the number of local state components (though in practice it is usually much smaller).

3.5 Merging in array analysis

Now we describe the special rules for merging states: these rules are the core technique that allows us to infer the required invariant. Before doing so, let us consider the operation of the analysis described so far on our simple motivating example. The allocation of the new array

```
T[] new_ta = new T[ta.length*2];
```

updates the program state as shown:

$$\langle \dots, Len(R_{arg(k)}) = c_0, NR(R_{id/A}) = [0..2 * c_0 - 1] \rangle$$

where c_0 is the length of `ta` (the 0^{th} argument of `expand`), and $R_{id/A}$ represents the result of the allocation. Next we enter the `for` loop:

```
for (int i = 0; i < ta.length; ++i)
    new_ta[i] = ta[i];
```

In the preamble of the loop, we store 0 in local variable `i`, recording this in ρ . The assignment to `new_ta[i]` corresponds to an `aastore` bytecode, and causes the uninitialized range of `new_ta` to contract to `[1..]`.

Now control flow takes us back to the loop head. We need to merge the current program state into the program state recorded when we first visited the loop head; these two states are:

$$\begin{aligned} & \langle \dots, \rho(i) = 1, \dots, NR(R_{id/A}) = [1..] \rangle \quad \text{and} \\ & \langle \dots, \rho(i) = 0, \dots, NR(R_{id/A}) = [0, 2 * c_0 - 1] \rangle \quad . \end{aligned}$$

Before going further, we define the concept of *integer state components*. These components include the integer-valued elements of ρ and stk , as well as *IntVals* that appear as bounds of uninitialized ranges. When we merge two states S_1 and S_2 , we merge these values component-wise. If an integer component has different values i_1 and i_2 in the two states, we can always merge them to \top_{iv} , but we can also choose to express the value of the component in the merged state as a function of a new variable unknown v .

This becomes useful when different state components can be expressed as functions of the same variable unknown. In our example, we wish to discover that the lower bound of the uninitialized range of `new_ta` varies with the same stride as the loop variable `i`.

Figure 1 shows how *IntVals* are merged to accomplish this aim. The `merge_intvals` function takes three arguments:

$$\begin{aligned} U & \subseteq \text{int} && \mapsto \text{VarUnknown} \\ \mu_1 & \subseteq \text{VarUnknown} && \mapsto \text{IntVal} \\ \mu_2 & \subseteq \text{VarUnknown} && \mapsto \text{IntVal} \end{aligned}$$

These maps are initially empty, but are modified by `merge_intvals`. U maps integer constant “strides” to generated variable unknowns that vary with the given stride. The μ maps are substitutions for variable unknowns. These substitutions will together construct a merged state S_3 where for each integer state component c :

$$S_3[c] = \top_{iv} \vee (\mu_1(S_3[c]) = S_1[c] \wedge \mu_2(S_3[c]) = S_2[c]) \quad .$$

Thus, $S_3[c]$ will abstract both $S_1[c]$ and $S_2[c]$, as a proper merged state must. When $S_1[c]$ and $S_2[c]$ are distinct constants, $S_3[c]$ is merged to a generated variable unknown for their difference, and the μ substitutions map this variable back to values in each input state. On later iterations the values being merged will already be variable terms; one will be chosen for the merged state, and the μ substitutions will transform that value to the other.

Figure 1 shows in more detail how *IntVals* are merged to accomplish this aim. Briefly, when merging two distinct constants, we create (or look up) a variable unknown that is assumed to vary by the appropriate difference in consecutive executions, remembering what value the variable represents in each of the merged states. Otherwise, one of the merged *IntVals* has a non-zero variable unknown

term. If that variable has already been assigned a value, and substituting that value makes the merged *IntVals* equal, then the merged value is unchanged; if the substituted value is not equal, then we must merge to \top_{iv} . If the variable is not yet mapped by the substitution, then we attempt to extend the substitution with a mapping for the variable that makes the merged *IntVals* equal, using the match function described below. If this fails, we must merge to \top_{iv} .

The match function works as follows. The arguments are two *IntVals*. Representing these in general form (giving each *IntVal* a term for every constant unknown c_i , possibly with $k_1 = 0$), our problem is to solve the following equation below for v :

$$av + k_0c_0\dots + k_nc_n + b = a'v' + k'_0c'_0 + \dots + k'_nc'_n + b' \quad .$$

We give up unless $a = a'$. If this holds, then we obtain

$$v = v' + \frac{k'_0 - k_0}{a}c'_0 + \dots + \frac{k'_n - k_n}{a}c'_n + \frac{b' - b}{a} \quad .$$

While we could imagine allowing *IntVals* with rational coefficients, we do not currently support that, and return this result only if all the divisions in the equation above have zero remainder (returning null otherwise).

We now return to our example. When we merge the end-of-loop state back into the loop head state, we discover that the $\rho(i)$ components differ by the constant 1. We therefore create a new variable unknown v , record this as $U[1]$, and set $\mu_1(v) = 0$ (the value of $\rho(i)$ in the first state being merged.) Later (though the order does not matter), we merge the uninitialized ranges for $R_{id/A}$. Since the new range is a half-open range and the other range is full, we merge to the half-open range. The left bound is determined by merging the corresponding integer components 0 and 1. These vary with a delta (1) for which a variable v has already been recorded, so we substitute v for this component in the merge state (no constant is added since the value of this component in S_1 is 0, which is the same value already recorded in as $\mu(v)$). So we obtain:

$$\langle \dots, \rho(i) = v, \dots, NR(R_{id/A}) \rangle = [v..] \rangle$$

Now we iterate the loop again. At the end of the loop body, the state is

$$\langle \dots, \rho(i) = v + 1, \dots, NR(R_{id/A}) \rangle = [v + 1..] \rangle$$

which we merge into the previous state of the loop head. Now when we merge the values of $\rho(i)$, we find that while these terms have a constant difference 1, they have non-zero variable terms, and are therefore led to line 27, where the match function computes the substitution $\mu_2[v] = v + 1$, which justifies returning $i_1 = v$ as the result of the merge. Later, we merge v and $v + 1$ again, this time as the low bounds of the uninitialized range for $R_{id/A}$. This time we find that there is already a substitution for v , and go to line 24. Fortunately, the substitution makes $\mu_2[i_1] = i_2$, and we can again return v as the result. We have correctly inferred that the low bound of the uninitialized range and the value of the loop variable i are the same.

```

1  IntVal merge_intvals( $i_1$ : IntVal,  $i_2$ : IntVal,
2                       $U$ : int  $\rightarrow$  VarUnknown,
3                       $\mu_1$ : VarUnknown  $\rightarrow$  IntVal,
4                       $\mu_2$ : VarUnknown  $\rightarrow$  IntVal) {
5  if ( $i_1 = \top_{iv} \vee i_2 = \top_{iv}$ ) return  $\top_{iv}$ ;
6  else if ( $i_1 = i_2$ ) return  $i_1$ ;
7  else {
8    if (var_term( $i_1$ ) = 0)
9       $i_1, i_2 \leftarrow i_2, i_1; \mu_1, \mu_2 \leftarrow \mu_2, \mu_1$ ;
10   let  $\delta = i_2 - i_1$  in
11     if (int_const( $\delta$ )  $\wedge$  var_term( $i_1$ ) = 0) {
12       if ( $U[\delta] = \text{null}$ ) {
13         let  $v = \text{new VarUnknown}$  in
14            $U[\delta] \leftarrow v; \mu_1[v] \leftarrow i_1; \mu_2[v] \leftarrow i_2$ ;
15         return  $v$ ;
16       } else {
17         let  $v = U[\delta]; d = (i_1 - \mu_1(v))$  in
18         assert var_term( $d$ ) = 0;
19         return  $v + d$ ;
20       } else {
21         let ( $a_1 v_1 = \text{var\_term}(i_1)$ ) in
22         if ( $a_1 \neq 0$ ) {
23           if ( $\mu_2[v_1] \neq \text{null}$ ) {
24             if ( $\mu_2[i_1] = i_2$ ) { return  $i_1$ ; }
25             else { return  $\top_{iv}$ ; };
26           } else {
27             let  $s = \text{match}(i_1, i_2)$  in
28             if  $s \neq \text{null}$  {
29                $\mu_2[v_1] \leftarrow s$ ; return  $i_1$ ;
30             } else return  $\top_{iv}$ ;
31           } else return  $\top_{iv}$ ;
32     }
  }

```

Figure 1: Procedure for merging integer state components

Our technique finds all integer state components that vary with the same “stride” in the first iteration of the loop, and makes the provisional assumption that they vary with that stride in all iterations. Such assumptions may of course be incorrect. When they are, the technique is still safe, since it validates that the assumptions lead to a fixed point of the analysis. When the fixed-stride assumption is erroneous, however, the validation iteration will merge such components to \top_{iv} , which will then allow a fixed point (with less information) to be reached. Consider, for example:

```
for (int j=1, int i=0; i < 10; i++)
  { ... ; j *= 2 ; ... }
```

In the first iteration of this loop, both i and j increase by one. Our method introduces a common variable for this delta; merging after the first iteration might produce $\rho(i) = v \wedge \rho(j) = v + 1$. In the previous case, another iteration showed that introducing the variable led to a fixed point. In this example, however, another iteration detects that the initial assumption was incorrect. We attempt to merge $v + 1$ and $2v + 2$; we have no mechanism for doing so, and therefore merge to $\rho(j) = \top_{iv}$.

3.6 Overflow

Any analysis that applies reasoning about abstract mathematical integers to concrete fixed-width machine integers must always worry about overflow. For example, in our case, we might worry that integer overflow might allow an index to “wrap around” and update a previously-initialized array element without a barrier. This turns out not to be a problem in our case, because of safety properties of the target language. The conservative definition of contract disables optimization unless the array elements are initialized in order: if element i was last initialized, contract loses all information unless $i + 1$ or $i - 1$ is the next element initialized. The uninitialized range of an array must contract in the same way on all paths leading to a control flow merge point (such as a loop head), or else it will merge to the empty range at that point. Therefore, an array store site whose barrier has been eliminated must be executed with a negative index, producing an array bounds exception, before it can possibly be executed with an index value that has wrapped around to a positive value.¹

4 Evaluation

In this section, we evaluate the effectiveness of these analyses. We discuss the benchmarks we use for evaluation, detail the effectiveness of the optimization, and explore the compile-time cost of the analysis. Finally we consider further techniques (not yet implemented) that might eliminate more barriers.

¹To get this completely correct, the analysis should handle the rare methods that catch array bounds exceptions specially, for example, by not eliminating array barriers at all in them.

4.1 Benchmarks

We present results from 5 of the 7 programs in the SPECjvm98 benchmark suite (omitting two benchmarks with very little heap or pointer manipulation). These are:

- **jess**, an expert-systems shell,
- **db**, a small database program,
- **javac**, a compiler,
- **mtrt**, a multi-threaded ray-tracer, and
- **jack**, a compiler-compiler.

We also present results for **jbb**, the SPECjbb2000 benchmark, run with 8 warehouses for a two-minute timing interval.

While the SPECjvm benchmarks are not very large programs, and use relatively small heaps, they do represent a range of programming styles, and we believe that they are useful test cases for this evaluation.

4.2 Results

This section gives our results. We have done this work in the “client” just-in-time (JIT) compiler of the Java HotSpot™ Virtual Machine, and perform measurements on a machine with eight 750Mhz UltraSPARC™ III processors. (The experimental platform will be relevant for later measurements of compile times.)

Tables 1 and 2 shows the effectiveness of the analyses at eliminating write barriers. Table 1 shows the number of barriers executed dynamically in JIT-compiled code, the percentage of those executions that can be eliminated by analysis, the breakdown of the compiled barrier executions into field and array stores, and the percentage of executions of each kind of barrier that can be eliminated. In our instrumentation of the code generated for a pointer store, we also counted, for each compiled store, the number of associated barrier executions in which the pre-value of the updated location was `null`. We call a store site whose pre-value is never (dynamically) non-null *potentially pre-null*.² Counting potentially pre-null sites is both a useful correctness check (our analysis should only eliminate barriers at potentially pre-null store sites!) and also provides an upper bound on the possible effectiveness of the pre-null technique. The last column lists the percentage of compiled barrier executions that are for potentially pre-null stores.

²This definition includes store sites that are compiled but never dynamically executed. Some such sites may be proved pre-null by our static analysis, so without this interpretation, the number of sites proved pre-null might exceed the potential.

The percentages in Table 1 are of total JIT-compiled barrier executions; in all cases, non-compiled barrier executions (because not all methods are JIT-compiled) comprise fewer than 3% of executed barriers, and in most cases considerably fewer.

Table 2 corresponds to table 1, but shows static rather than dynamic counts of eliminated barriers. The dynamic results are obviously more important in determining the effect of the analysis on the running time of the programs, but static results are also important, since they determine the effect of the analysis on compiled code space (which may also indirectly improve running time via instruction cache effects).

benchmark	Total $\times 10^6$	% elim	% Potentially pre-null	Field/ Array	Field % elim	Array % elim
jess	7.9	50.5	75.0	51/49	99.7	0.0
db	30.1	10.2	28.2	10/90	99.4	0.0
javac	19.9	32.8	38.5	92/ 8	33.9	20.5
mtrt	3.0	61.9	91.6	41/59	72.0	54.7
jack	10.7	41.0	54.0	74/26	55.5	0.0
jbb	297.8	25.6	53.4	69/31	37.0	0.0

Table 1: Analysis results: dynamic

In general, our results show that our analyses can eliminate a significant fraction, though not a consistent majority, of barriers – roughly between 1/4 and 2/3, with the exception of **db**'s dynamic result. Our results are significantly better for object fields than for arrays: in two of five cases, we are unable to prove any array stores to be initializing (and in a third, those proven initializing account for a negligible fraction of total executed array stores.) However, the optimization is not in vain: in **mtrt**, for example, the majority of eliminated barrier executions are for array stores.

When we compare the actual percentage of barriers eliminated with the upper bound of the potential sites, we find we are obtaining a fairly large fraction of the benefits achievable by these techniques. In the the dynamic results of table 1, we eliminate at least 2/3 of all executions of potentially pre-null barriers in all cases except **db** and **jbb**, and eliminate almost 1/2 for **jbb**.

Comparing our static and dynamic results, we found that (as one might expect) the percentage of stores executed dynamically that are array stores is usually higher, sometimes considerably, than the corresponding static percentage. Therefore, since we have less success in finding initializing array stores, our dynamic elimination rate is generally lower than the static elimination rate.

4.3 Detailed evaluation

We further analyzed our results for individual store sites. For each benchmark, we sorted the results, and considered the most frequently-executed store sites whose barriers were not eliminated. This

benchmark	Total	% elim	% Potentially pre-null	Field/Array	Field % elim	Array % elim
jess	370	67.3	93.5	81/19	82.4	1.4
db	77	42.9	89.6	69/31	62.3	0.0
javac	1684	52.0	81.5	93/ 7	54.5	17.0
mtrt	305	42.6	93.4	47/53	75.4	14.1
jack	869	35.8	95.6	43/57	83.6	0.0
jbb	1137	38.3	90.0	57/43	64.7	2.9

Table 2: Analysis results: static

consideration suggests further work that would allow more SATB barriers to be eliminated; we discuss these techniques below.

4.3.1 Null-or-same analysis.

We noticed that several frequently-executed store sites, while not pre-null, had a related property that should allow elimination of their associated write barriers. For these sites, we can prove (currently by inspection, not via automated tools) that the write either overwrites null, or else *writes the value the field already contains*. Obviously, no SATB barrier is required in either case. An important example occurs in `Hashtable.Enumerator`'s `hasMoreElements` method, shown in figure 2.

```

1| public boolean hasMoreElements() {
2|     Entry e = entry;
3|     int i = index;
4|     Entry t[] = table;
5|     /* Use locals in loop body */
6|     while (e == null && i > 0) {
7|         e = t[--i];
8|     }
9|     entry = e; // Frequently executed
10|    index = i;
11|    return e != null;
12| }

```

Figure 2: Code for `Hashtable.Enumerator.hasMoreElements`

The “frequently executed” store requires no barrier. To see why, consider an execution of this method. Assume that the `Enumerator` `this` is thread-local, so values of its fields change only as a result of actions by the current thread. The first line reads the value of `this.entry`; call

this R_0 . The **while** loop tests whether R_0 is null. If R_0 is null, then we have determined that the pre-value of `this.entry` at line 9 is null: the barrier need not be executed in this case. If, on the other hand, R_0 is not null, then the loop body is not executed, so $e = R_0$ when we reach line 9. The store therefore does not change the value of `this.entry`, and no barrier is necessary. So no barrier is necessary in either case.

Stores of this form account for 15% of the write barriers executed in **javac**, 14% for **jack**, and 4% in **jbb**. We are currently considering how best to incorporate this observation into our analysis. In any case, we would require further escape analysis infrastructure to clone a version of `hasMoreElements` compiled under the assumption that its `this` argument is thread-local, which is a necessary condition of the optimization.

4.3.2 Array swap idiom

This section and the next discuss optimizations suggested by examination of frequently-executed array store sites occurring within loops. Some of these loops perform *array rearrangements*, moving elements within an object array, possibly overwriting some elements. If such a rearrangement occurred atomically with respect to the collector's tracing of the array, then only the overwritten elements (if any) would need to be logged. These updates might be a small fraction of all the writes performed within the loop. Of course, the arrangement does not actually occur atomically; to nevertheless get some benefit from this observation, we can restrict the direction in which the collector scans object arrays, or try to detect mutator/collector interference.

The array swap optimization is suggested by the non-eliminated stores of the **db** benchmark. This benchmark represented our worst results, eliminating only about 10% of dynamically executed barriers. The top two stores in **db**, together accounting for more than 70% of stores, seem initially unpromising, since neither is potentially pre-null; in fact, *no* executions observe a null previous value. But these stores occur within a loop next that performs an array rearrangement, as part of a (suboptimal) sorting algorithm:

```
1| for (...; gap > 0; ...)
2|     for (j = ...)
3|         ...
4|             e = index[j];
5|             index[j] = index[j+gap];
6|             index[j+gap] = e;
```

Lines 4-6 swap two elements of the array, and are the only writes to the array in the loop nest. Since a swap is a permutation, and the composition of permutations is another permutation, the loop nest permutes the array without losing any elements. This is enticing, since the array references the same objects before and after each swap; if the sorting loop occurred atomically with respect to the marking thread, it would not change the set of object references in the array, and therefore the array stores in last two lines would not require barriers.

Unfortunately, these statements will be executed concurrently with the marking thread, and various interleavings could cause either of the swapped values to be missed in the absence of write barriers. However, if we intertwine the compiler and garbage collector somewhat, committing the concurrent marking thread to scan object arrays in (say) ascending index order, then we can eliminate one of the barriers when we detect the idiom of a swap within the same array. To do this, we must reverse the swap, so that the higher index is written first:

```
4|         e = index[j+gap];
5|         index[j+gap] = index[j];
6|         index[j] = e;
```

Since $gap > 0$, the concurrent marking thread will execute

```
a|         t1 = index[j];
|         ...
b|         tn = index[j+gap];
```

On a sequentially consistent machine, no barrier is necessary for the store at line 6: either the read (a) of `index[j]` by the concurrent marker precedes the store at 6, and thus observes the pre-value, or it follows the store at 6, in which case both reads follow the completed swap operation, which preserves the object graph.

This argument does not take mutator/mutator concurrency into account; see section 4.3.5 for more discussion of this issue.

4.3.3 Slide-down idiom

The second array rearrangement optimization is suggested by **jbb**. Two of the three most frequently-executed non-eliminated store sites, together accounting for about 12% of stores, suggest another way to take advantage of loops that perform array rearrangements.

These sites both have the form:

```
1|         for (j = index+1; j < count; j++) {
2|             a[j-1] = a[j];
3|         }
```

To remove an element from a collection represented by an array, it “slides elements down” to replace the deleted element at `index`. If the entire loop occurred atomically with respect to the concurrent marking thread, then only the overwritten element `a[index]` would need to be logged. However, we must take mutator/collector concurrency into account. It turns out that if the collector scans the array in the same direction as object movement, no elements can be lost, but if the directions differ, then the collector may fail to observe elements copied by the loop.

We propose two possible schemes to take advantage of this observation. The first is more general, and applies whenever a compiler can identify a block of code (usually a loop) that moves elements within an object array in such a way that the element set at the end of the code block is a subset of the original elements. In this situation, we propose an optimistic concurrency control scheme. We would devote bits in the header of object arrays to indicate the *tracing states* of the array, one of *untraced*, *tracing*, and *traced*. The concurrent marker would update this state information as it traces arrays. The compiler would precede the element movement code block with code to log the “lost” elements, and also add code to read the tracing state before and after the code block. If the states indicate that the marker may have done any tracing of the array concurrently with the loop, then the mutator places the entire array on a special *retrace list*, requiring the collector to trace it again.

This technique will be advantageous if the number of elements “slid” per element removal is relatively large, and if the the probability of collector/mutator interference on an individual array is small. Both conditions obtain for **jbb**. This technique could also be applied to eliminate both barriers in the array swap idiom that dominates **db**, but in that case the mutator spends so much time in the loop that the probability of retrace is relatively high.

The second scheme is more specific to the current example, and takes advantage of the observation that the collector observes all the copied elements when it scans in the same direction as the object movement. This scheme devotes bits in the object array header that allow the compiled code to force the collector to scan object arrays in the required direction: a downward-moving loop like the example would be preceded by code setting the bits to inform the collector that the array must be scanned downwards. More bits are necessary to allow the collector to inform the mutator code of its scanning: if a downwards-moving loop operates on an array that the collector is already scanning upwards, it must leave its bit set at the end of the loop, so the collector knows that it must rescan the loop in the proper direction. (This opens the possibility of collector/mutator livelock, if the mutator repeatedly moves elements in alternating directions. Another bit pattern could actually “lock” the array from such loops, allowing the collector to obtain a consistent view.)

We have not yet considered barriers required by the standard library method `System.arraycopy`, but many uses in these benchmarks have the same form as the loop described in this section; the same techniques could be used to eliminate barriers in those cases.

4.3.4 Miscellaneous

In this section we complete our examination of the non-eliminated stores in our benchmark, suggesting some other methods of eliminating SATB barriers.

The **jess** benchmark has a non-eliminated array store site that ties for the most frequently-executed (accounting for 23% of all stores), yet is potentially pre-null.

Further examination reveals that the store is in fact unnecessary, but that the analysis would have to be extended in several ways to eliminate it. We would have to:

- track abstract values of integer fields of thread-local object references;
- decide conditionals statically when the analysis determines the outcome of the test; and
- extend the analysis to understand the effect of `System.arraycopy` on array uninitialized ranges.

The third-most-frequently-executed pointer store sites of the **db** is potentially pre-null, and accounts for about 17% of the stores. The store is at line 4:

```

1|  index = new Entry[n];
2|  i = 0;
3|  while ( e.hasMoreElements() )
4|      index[i++] = (Entry)e.nextElement();

```

The problem is that `index` is an object field (of `this`) rather than a local variable. Since `this` might be globally visible, the array might be updated by other threads (or even via the calls to `hasMoreElements` or `nextElement`) between iterations.

If it could be determined (most obviously by inlining) that neither iterator method accesses the field `Database.index`, then we could transform this loop to store the allocated array into a local variable, perform the initialization, and only then write the result back into the field. (This is always a possible execution of the original code under the memory model.) This would enable the barrier to be eliminated.

Though we do fairly well on **mtrt**, eliminating 62% of dynamically-executed barriers, possibilities remain: the third- through sixth-most-frequently-executed stores are non-eliminated potentially pre-null array stores, all in the same method, `OctNode.Copy`. An `OctNode` has three fields that reference object arrays, and `Copy` copies the contents of the arrays from an `OctNode` argument to the corresponding arrays in `this`. `Copy` is called by only one method, where it is invoked on a newly-allocated `OctNode`. Making these facts apparent to the analysis would require more aggressive inlining: in this case, letting the fact that a method is called at only one invocation site allow the inlining of even large methods would work. Eliminating these barriers for **mtrt** would also require handling an oddity of its code: the `OctNode` constructor contains the following:

```

Adjacent = new OctNode[6];
Child = new OctNode[8];
for (i = 0; i < 6; i++) Adjacent[i] = null;
for (i = 0; i < 8; i++) Child[i] = null;

```

Our array analysis could easily be used to identify these loops as redundant, allowing them to be eliminated entirely. In any case, we must recognize that they do not change the uninitialized ranges of the arrays they update.

The most frequently-executed pointer store in the **jack** benchmark is in `Vector.addElement`; this non-eliminated store is potentially pre-null, and accounts for 13% of all pointer stores. This method is not inlined, because it is synchronized. If our analysis were extended in directions similar to those mentioned in our discussion of **jess**, it would be possible to prove, at least at some call sites, that the `Vector` argument was thread-local, and did not require locking. This would enable inlining, which would enable the barrier elimination. In particular, we would need to track the values of integer fields of thread-local heap objects, and decide control flow branches statically when possible.

4.3.5 Mutator concurrency issues

The (very) informal correctness arguments for the optimizations of sections 4.3.2 and 4.3.3 considered concurrency between the collector and a single mutator threads, but did not consider multiple mutator threads. It turns out that even very simple mutator/mutator concurrency can have bad effects that invalidate these optimizations. Consider the swap idiom of section 4.3.2, which claimed that if the collector thread scans arrays in ascending order, the barrier for the write at line 5 may be elided:

```
0|          // Assume i < j
1|          t1 = a[i];
2|          t2 = a[j];
3|          log t2;      // logging barrier.
4|          a[j] = t1;
5|          a[i] = t2;  // No logging.
```

Now consider another mutator thread that writes a value z to `a[j]`. It will perform a write barrier, so it will do:

```
1|          t = a[j];
2|          log t;
3|          a[j] = z;
```

These may be interleaved as follows:

Initially: $a[i] = x, a[j] = y$	
Thread 1	Thread 2
$t1 = a[i];$ // reads x	
$t2 = a[j];$ // reads y	
$\log t2;$ // logs y	
	$t = a[j];$ // reads y
	$\log t;$ // logs y
$a[j] = t1;$ // writes x	
$a[i] = t2;$ // writes y	
	$a[j] = z;$ // writes z

At the end of this, $a[i] = y$ and $a[j] = z$. The value x was in the array at the start, and therefore should be observed by the concurrent marker, but has been lost without having been logged. A similar argument can be constructed to show that the optimization eliminating write barriers in the “copy-down” loop of section 4.3.3 may also be invalidated by mutator concurrency.

It is interesting to note that while the values observed by the mutator threads in these examples are perfectly permissible results of programs with data races under the Java Memory Model [12, 18], the concurrent collector must be held to a higher standard: it must preserve correctness under all possible interleavings. When some of the collector implementation (i.e, write barriers) is provided by compilers, then we must be aware of this distinction, and be careful to avoid optimizations that might be appropriate for mutator code but inappropriate for collector code.

As far as we can determine, optimizations such as those in sections sections 4.3.2 and 4.3.3, which elide barriers on the basis of arguments about permissible interleavings of mutator and collector execution, may only be applied if mutator/mutator concurrency is not possible. This may be ensured in several ways:

- The target array may be proven to be thread-local.
- The program may be proven to obey a consistent locking discipline in accessing the array.
- The generated code may be modified to obtain a lock during loops that elide barriers. This approach raises contention and deadlock issues that require some care.
- Future processors might support inexpensive *transactional memory* [13]. Performing array operations as part of transactions (encompassing code blocks with elided barriers) would allow detection of concurrent interference that would otherwise invalidate the barrier elision.

4.3.6 Summary

In this section we have suggested a number of further techniques for eliminating further barriers. Table 3 shows the estimates of dynamic barrier elimination rates if all the techniques described in

this section are successfully implemented.³ The numbers in the last column are greater in several cases than the potentially pre-null column of table 1, since several of the speculative techniques eliminate barriers for non-pre-null store sites.

benchmark	Dynamic % elim	
	current	+ further work
jess	51%	74%
db	10%	62%
javac	33%	48%
mtrt	62%	75%
jack	41%	62%
jbb	26%	42%

Table 3: Estimated increases from further work

If this future work is successful, we would closely approach or exceed elimination of the majority of SATB barriers for all our benchmarks. The required techniques vary. Some are generally useful compilation techniques, some require extension of our analysis, and others propose closer cooperation between mutator and collector. None are individually daunting.

4.4 Inlining level and compilation time

Inlining exposes more information to static analyses like the ones we describe in this paper. An “inline limit” parameter of this compiler determines the maximum bytecode size of an inlined method. Of course, more aggressive inlining increases compile time as well.

Figure 3 shows the effect of the inline limit on analysis effectiveness and compilation time, without our analysis (B), with the field analysis only (F), and with both the field and array analysis (A). (Note that the compilation time scale is logarithmic in figure 3.) Table 4 shows in full detail the numbers on which figure 3 is based.

³Actually these estimates are lower bounds, since we assume only that the specific frequently-executed barriers considered are eliminated; the techniques may also eliminate other less frequently-executed barriers not considered.

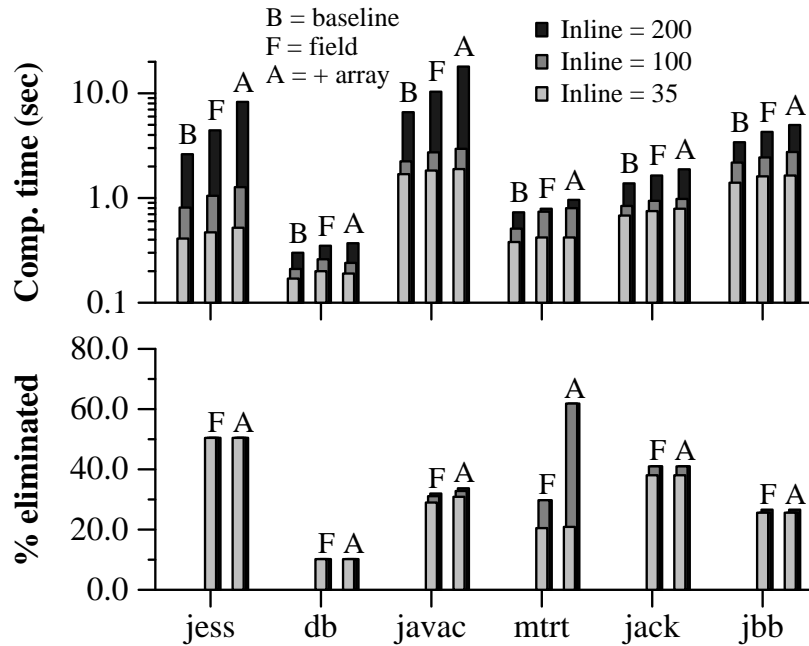


Figure 3: Compile-time cost and analysis effectiveness as a function of inlining level

While the analysis time becomes comparable to the rest of compilation time for the larger examples, note that this compilation time is for a “client” JIT compiler intended to produce medium-quality code quickly. The “server” compiler produces high-quality code, but is approximately an order of magnitude slower; this analysis would fit well into the goals and compilation budget of such a compiler. The 100-bytecode inlining level gains essentially all the analysis results, and adds much less compilation time than the 200-bytecode inlining level. This is the inlining level used for the results in Table 1.

In the implementation of the analysis, one technique stood out as important to achieving acceptable performance on large problems. During performance debugging, we found that the fixed-point calculation was taking many iterations to converge. We determined that changes to the abstract store were being propagated to all (transitive) successors, even when the new information was irrelevant to those successors. We solved this problem by performing an *abstract garbage collection* at the end of each basic block, deleting information about thread-local abstract references that are unreachable in the block’s post-state. This approach is suggested by Jones and Muchnick [16], whose *cleanup* operation performs such a garbage collection (as well as operations associated with limiting the size of the representation of the heap state). Jones and Muchnick have the goal of classifying nodes as amenable to reference counting or requiring other forms of garbage collection that can collect garbage cycles. It is somewhat surprising that they did not suggest an obvious extension of this technique: if this garbage collection finds unreachable abstract nodes, then the corresponding concrete nodes may be collected.

Figure 4 shows the effect of the two levels of analysis on compiled code size, at inlining level 100.

bench- mark	inline limit	comp time w/o analysis	Field analysis only		Field + array analysis	
			% elim	Δ comp time	% elim	Δ comp time
jess	35	0.41 s	50.4%	0.06 s (15.2%)	50.4%	0.11 s (27.3%)
jess	100	0.81 s	50.5%	0.24 s (30.2%)	50.5%	0.46 s (57.1%)
jess	200	2.62 s	50.5%	1.81 s (69.2%)	50.5%	5.66 s (216.4%)
db	35	0.17 s	10.2%	0.03 s (16.1%)	10.2%	0.03 s (15.5%)
db	100	0.21 s	10.2%	0.05 s (23.8%)	10.2%	0.04 s (17.5%)
db	200	0.30 s	10.2%	0.05 s (17.5%)	10.2%	0.08 s (25.3%)
javac	35	1.69 s	29.0%	0.14 s (8.4%)	30.9%	0.20 s (11.8%)
javac	100	2.24 s	31.1%	0.49 s (21.8%)	32.8%	0.71 s (31.6%)
javac	200	6.60 s	32.0%	3.75 s (56.9%)	33.7%	11.39 s (172.5%)
mtrt	35	0.38 s	20.5%	0.04 s (10.3%)	20.9%	0.04 s (10.8%)
mtrt	100	0.51 s	29.8%	0.23 s (45.3%)	61.9%	0.29 s (57.9%)
mtrt	200	0.73 s	29.8%	0.05 s (7.2%)	61.8%	0.23 s (31.2%)
jack	35	0.68 s	38.0%	0.07 s (10.3%)	38.0%	0.11 s (15.9%)
jack	100	0.84 s	41.0%	0.10 s (12.0%)	41.0%	0.14 s (16.4%)
jack	200	1.38 s	41.0%	0.26 s (18.9%)	41.0%	0.51 s (36.9%)
jbb	35	1.40 s	25.6%	0.21 s (15.3%)	25.6%	0.24 s (17.3%)
jbb	100	2.18 s	25.6%	0.26 s (12.0%)	25.6%	0.57 s (26.0%)
jbb	200	3.41 s	26.6%	0.87 s (25.6%)	26.6%	1.57 s (46.2%)

Table 4: Compile-time cost and analysis effectiveness as a function of inlining level

Each column is labeled with the compiled code size before the optimization. To ensure comparability across runs with different optimization modes, code sizes were summed only for methods compiled in all runs for a benchmark. This excluded at most 1% of compiled methods.

Write barrier elimination decreases compiled code size by between 2 and 6%. Array analysis has smaller impact than it does on dynamic elimination rates, since array barriers usually occur in loops, which magnifies their dynamic impact.

4.5 End-to-end performance

In our current system, the effect of all this work is rather small, because the SATB barrier is expensive only when marking is in progress, which is a small fraction of total running time. However, in the future we plan to incrementalize marking, spreading it out over longer periods. To see why, consider an application running on a two-processor machine. Currently marking might run for 5 consecutive seconds every 25 seconds, requiring designers planning for worst cases to assume that they get only one processor for considerable periods. The alternative approach might perform the marking in 50 msec increments distributed evenly over these 25 seconds. The incremental approach allows designers to assume the machine has a constant 1.8 processors available.

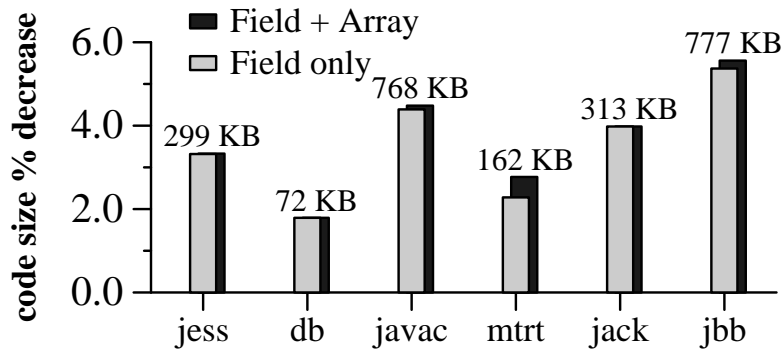


Figure 4: Analysis effect on code size, for various inlining levels

Table 5 shows end-to-end throughput on the **jbb** benchmark for three different modes of operation.⁴ The **no-barrier** mode eliminates *all* SATB barriers (we run all tests with a heap sufficiently large to require no marking). The **always-log** mode simulates the future work described above, by eliding the check for whether marking is in progress and always logging non-null pre-values. Write barrier elimination is disabled. Finally, the **always-log-elim** mode is like **always-log**, but enables write barrier elimination. Each result is the average of 5 runs; higher is better. In this mode, SATB barriers cost about 2.5% in end-to-end performance; this percentage would be greater if we used the more-optimizing compiler. Eliminating 25% of the barriers in **jbb** gets back approximately that fraction of this cost.

Note again that for reasons of implementation simplicity this work has been done with the “client” JIT compiler of the Java HotSpot VM, which is aimed at producing code quickly. Using a more highly-optimizing compiler, such as the Java HotSpot VM’s “server” compiler, would magnify the cost of SATB barriers, and increase the importance of eliminating them.

Barrier mode	Throughput	Relative to no-barrier
no-barrier	29968	1.000
always-log	29218	0.975
always-log-elim	29503	0.984

Table 5: **jbb** end-to-end barrier cost

5 Related Work

Vecchev and Bacon [24] present a dynamic limit study suggesting the potential for elimination of write barriers supporting concurrent marking. Their main observation is that the lifetime of a pointer

⁴We use **jbb** because its relatively longer running times make it easier to reliably detect small performance differences.

to an object is often contained within the lifetime of another pointer to the same object, allowing barriers associated with creation or deletion of the shorter-lived pointer to be elided. They also note that a large fraction of SATB barrier executions are pre-null, and propose, as we do, that allowing the compiler to assume a known scanning order might allow further barrier elision. While suggestive, this work does not provide an analysis to prove that the observed dynamic properties hold on all executions.

Barth [4] presents a static analysis to eliminate write barriers in a reference-counting collector. This includes the observation that a reference-count decrement is unnecessary for an initializing write to a newly allocated object, where the overwritten field is known to contain null. However, no algorithm is given for taking advantage of this observation, and the algorithms that are given assume programs with no procedure calls in single-threaded systems.

Zee and Rinard [28] and Shuf *et al.* [23] present techniques for removing write barriers supporting remembered set maintenance in garbage collection. Hosking *et al.* [15] describe techniques for reducing the number of write barriers required for tracking updates to a persistent store. In all cases, the purpose and form of the removed write barriers are very different from ours, as are the analysis techniques.

There is a large literature on pointer and shape analysis (see, e.g., [16, 7, 22] and escape analysis (see, e.g., [25, 8]). No previous work that we know of has considered the relationship between shape or escape analysis and elimination of SATB write barriers. It is interesting to compare the analysis we present with escape analyses. As discussed in section 2, our analysis must be more precise to eliminate barriers for stores to objects that are currently thread-local but later escape. Section 2.4 discussed previous work related to our use of two abstract reference values per allocation site.

There has been considerable previous work on bounding ranges of integer variables [21, 17]. This is not sufficient for our purposes, since we need to know relationships between values in particular states. This analysis could be accomplished by identifying loop *induction variables* [1, p. 644][11] and their strides, and the values of the induction variables on entry to the loop, but it is interesting to note that our method requires no identification of loop structure. We know of no previous work that identifies uninitialized subranges of arrays, nor that accomplishes the inference of integer state components with common strides as part of an abstract interpretation.

6 Conclusions and Future Work

Concurrent techniques are attractive for preventing garbage collection from excessively impacting applications with large live data working sets and (soft) real-time requirements. Experience has shown that SATB concurrent marking requires considerably shorter pauses than incremental update techniques. One barrier to the acceptance of SATB techniques has been the greater cost, in execution time and compiled code space, of the write barriers that must be executed at each pointer store. We have presented static analyses that prove these barriers unnecessary in many cases.

These analyses gave moderately good results for our benchmarks, but our measurement infrastruc-

ture allowed us to identify the most frequently-executed individual store sites whose barriers were not eliminated. Section 4.3 performed a detailed evaluation of important store sites whose barriers the analysis did not eliminate, revealing several interesting paths for future work to improve our results. Some of these are more detailed static analyses. Others suggest ways in which the collector/mutator interaction could be modified to allow more barriers to be eliminated.

Finally, one perhaps-reasonable reaction to this work is that the amount of effort required is out of proportion to the specific benefit provided. However, there are many problems within compilation that can make use of analyses that reveal this much information about a compiled method: these analyses could augment alias analysis; determination of exact types for, e.g., devirtualization of virtual calls; discovery of array indexing properties for bounds check removal; escape analysis for stack allocation and/or lock elision — the list is long. So our view is that these analyses should be part of an integrated static analysis framework that provides a variety of information to inform subsequent compilation steps, of which SATB write barrier removal is just one.

7 Acknowledgements

We would like to thank Jens Palsberg, Victor Luchango, Mark Moir, Alex Garthwaite, David Chase, Jan-Willem Massen, Tony Printezis, and Miriam Kadansky for helpful and informative discussions and readings, and anonymous referees for useful feedback.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.
- [2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on Sliding views. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, Anaheim, CA, November 2003.
- [3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.
- [4] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.
- [5] BEA. BEA WebLogic JRockit. The Server JVM. URL: http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf.

- [6] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, ON, Canada, June 1991. ACM Press.
- [7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 296–310, 1990.
- [8] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, October 1999. ACM Press.
- [9] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Trans. Softw. Eng. Methodol.*, 9(1):51–93, 2000.
- [10] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 2004 International Symposium on Memory Management*. ACM Press, 2004.
- [11] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.
- [12] JSR-133 Expert Group. JSR 133: The java™ memory model and thread specification revision. URL. <http://jcp.org/en/jsr/detail?id=133>.
- [13] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300. ACM Press, 1993.
- [14] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.
- [15] Antony L. Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahmamath. Optimizing the read and write barrier for orthogonal persistence. In *Proceedings of the Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, August 1998.
- [16] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [17] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

- [18] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391. ACM Press, 2005.
- [19] Yoav Ossia, Ori Ben-Yitzhak, Irit Gofit, Elliot K. Kolodner, Victor Leikehman, and Avi Oshankov. A parallel, incremental and concurrent gc for servers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 129–140. ACM Press, 2002.
- [20] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.
- [21] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, volume 35.5 of *ACM Sigplan Notices*, pages 182–195, N.Y., June 18–21 2000. ACM Press.
- [22] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
- [23] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. Exploiting prolific types for memory management and optimizations. In *Conference Record of the Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 2002.
- [24] Martin T. Vechev and David F. Bacon. Write barrier elision for concurrent garbage collectors. In *Proceedings of the 4th international symposium on Memory management*, pages 13–24. ACM Press, 2004.
- [25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, October 1999. ACM Press.
- [26] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in *Lecture Notes in Computer Science*, Saint-Malo (France), September 1992. Springer-Verlag. <ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps>.
- [27] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.
- [28] Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 191–210. ACM Press, 2002.

A Formal Definitions of Support Functions

This appendix gives formal definitions of functions defined informally in section 2.4. First we define the functions $\text{AllNonTL}()$, $\text{AllNonTLCond}()$, and $\text{nAllNonTL}()$, used in section 2.4 to describe which reference values become non-thread-local because of stores. In the definitions below, NL is a non-locality set, RS is an element of Refs , and σ is an abstract store. We denote sequences of reference values using stack notation.

$$\begin{aligned}
 \text{AllNonTL}(NL, \{\}, \sigma) &= NL \\
 \text{AllNonTL}(NL, \{r\} \cup RS, \sigma) &= \\
 &\quad \text{AllNonTL}(NL \cup \bigcup_{\forall f} \sigma(r, f), RS, \sigma) \\
 \text{AllNonTLCond}(NL, RS, val, \sigma) &= \\
 &\quad \text{if } RS \cap NL \neq \emptyset \text{ then } \text{AllNonTL}(NL, val, \sigma) \text{ else } NL \\
 \text{nAllNonTL}(NL, \overline{v_i}, \sigma) &= \bigcup_{\forall i} \text{AllNonTL}(NL, v_i, \sigma)
 \end{aligned}$$

Now we give definitions for $\text{rngSubst}()$, $\text{replS}()$, and transfer , used in section 2.4 to merge attributes of an abstract reference value $R_{id/A}$ denoting the object most recently allocated at allocation site id into the summary reference $R_{id/B}$ for that site in a program state.

$$\begin{aligned}
 \text{replS}(s, v_1 \leftarrow v_2) &= \\
 &\quad (s - \{v_1\}) \cup (\text{if } v_1 \in s \text{ then } \{v_2\} \text{ else } \{\}) \\
 \text{rngSubst}(m, v_1 \leftarrow v_2) &= \\
 &\quad \forall x \in \text{dom}(m) \ m[x \leftarrow \text{replS}(m[x], v_1 \leftarrow v_2)] \\
 \text{transfer}(\sigma, r_1, r_2) &= \\
 &\quad (\sigma - r_1)[\forall (r, f) \in (\sigma - r_1) : (r, f) \leftarrow \\
 &\quad \quad (\text{if } r = r_2 \text{ then} \\
 &\quad \quad \quad \text{replS}(\sigma(r_1, f), r_1 \leftarrow r_2) \\
 &\quad \quad \quad \cup \text{replS}(\sigma(r_2, f), r_1 \leftarrow r_2) \\
 &\quad \quad \text{else } \text{replS}(\sigma(r, f), r_1 \leftarrow r_2))]
 \end{aligned}$$

About the Authors



David Detlefs is a Senior Staff Engineer at Sun Microsystems Laboratories, and works on a variety of advanced language implementation techniques, especially garbage collection and just-in-time compilation. He worked previously at the Digital Equipment Corporation's Systems Research Center. He has an S.B. degree from the Massachusetts Institute of Technology, and a Ph.D. from Carnegie Mellon University, both in Computer Science.



V. Krishna Nandivada is a Ph.D. student at University of California at Los Angeles. His interests include program optimization and program verification. He has a Bachelors degree from National Institute of Technology (Rourkela), and a Masters degree from Indian Institute of Science. He worked previously at Hewlett Packard's Compilers and Tools group.