Compile-Time Concurrent Marking Write Barrier Removal

V. Krishna Nandivada*
UCLA Computer Science Dept.
Boelter Hall
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 [24]. This section motivates 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 [17, 18, 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 an example of the *incremental update* form of mutator/collector interaction. That is, the mutator informs the 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. Objects allocated during marking, while implicitly marked, are not part of the snapshot object graph, and need be examined by the marker. In most programs, 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 system, pause times necessary to complete SATB marking are sometimes more than an order of magnitude smaller than corresponding incremental-update pauses. 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 [12], 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.

*Work done while at Sun Microsystems.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

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 writes 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 [23, 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 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 most recently at id, and $R_{id/B}$ summarizes all references to objects allocated at id previously in the method execution. The predicate $\mathrm{unique}(R_{id/A})$ is true (and false for all $R_{id/B}$ *Ref*s), 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 *Ref*s, 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:

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

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 $\{\mathit{GlobalRef}\}$. Further, all references reachable via *GlobalRef* are collapsed into *GlobalRef:* $\forall f : \sigma(\mathit{GlobalRef}, f) = \{\mathit{GlobalRef}\}$.

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 $\rho(y) = \{\mathit{GlobalRef}\}$. The operand stack stk is initialized to the empty stack. Constructors are treated in a similar fashion, except for the initial value of $\rho(\mathtt{this})$, 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 $< \rho, \sigma, \mathit{NL}, stk >$. We use the notation $[stk : \overline{v_i}]$ to denote pushing the vector $\overline{v_i} = v_0, ..., v_n$ onto the stack stk.

$$< \rho, \sigma, \mathit{NL}, stk > \quad \text{load}(x) \quad \Longrightarrow \quad < \rho, \sigma, \mathit{NL}, [stk : \rho(x)] >$$

$$< \rho, \sigma, \mathit{NL}, [stk : val] > \quad \text{store}(x) \quad \Longrightarrow \quad < \rho[x \leftarrow val], \sigma, \mathit{NL}, stk >$$

$$< \rho, \sigma, \mathit{NL}, stk > \quad \text{getstatic}(f) \quad \Longrightarrow \quad < \rho, \sigma, \mathit{NL}, [stk : \{\mathit{GlobalRef}\}] >$$

$$< \rho, \sigma, \mathit{NL}, [stk : val] > \quad \text{putstatic}(f) \quad \Longrightarrow \quad < \rho, \sigma, \text{AllNonTL}(\mathit{NL}, val, \sigma), stk >$$

$$< \rho, \sigma, \mathit{NL}, [stk : obj] > \quad \text{getfield}(f) \quad \Longrightarrow \quad < \rho, \sigma, \mathit{NL}, [stk : \textstyle\bigcup_{ot \in obj} \text{lookup}(\sigma, ot, \mathit{NL}, f)] >$$

$$\begin{array}{l}
< \rho, \sigma, \mathit{NL}, [stk : obj, val] > \quad \text{putfield}(f) \quad \Longrightarrow \\
\quad \text{if } obj = \{r\} \wedge \text{unique}(r) : \\
\qquad < \rho, \sigma[(r, f) \leftarrow val], \\
\qquad\quad \text{AllNonTLCond}(\mathit{NL}, obj, val, \sigma), stk > \\
\quad \text{else:} \\
\qquad < \rho, \sigma[\forall ot \in obj : (ot, f) \leftarrow \sigma(ot, f) \cup val], \\
\qquad\quad \text{AllNonTLCond}(\mathit{NL}, obj, val, \sigma), stk >
\end{array}$$

$$\begin{array}{l}
< \rho, \sigma, \mathit{NL}, [stk : \overline{v_i}] > \quad \text{invoke}(m(\overline{a_i}) : T) \quad \Longrightarrow \\
\quad \text{if } T \text{ is a reference type:} \\
\qquad < \rho, \sigma, \text{nAllNonTL}(\mathit{NL}, \overline{v_i}, \sigma), \\
\qquad\quad [stk : \{\mathit{GlobalRef}\}] > \\
\quad \text{else:} \\
\qquad < \rho, \sigma, \text{nAllNonTL}(\mathit{NL}, \overline{v_i}, \sigma), stk >
\end{array}$$

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 $\{\mathit{GlobalRef}\}$, 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, σ), which returns the result of extending the non-locality set *NL* with the *Ref* set RS and all *Ref*s (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($\sigma, r, \mathit{NL}, f$) returns $\{\mathit{GlobalRef}\}$ if r is in *NL* (is non-thread-local), and otherwise returns $\sigma(r, f)$.)

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, σ) 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 *Ref*s 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, σ) for all the v_i in $\overline{v_i}$.

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 interprocedural 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.)

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.

$$\boxed{\begin{array}{l} < \rho, \sigma, NL, [stk : arr, ind] > \qquad \text{aaload} \quad \Longrightarrow \\ \qquad < \rho, \sigma, NL, [stk : \bigcup_{at \in arr} \sigma(at, f_{elems})] > \end{array}}$$

$$\boxed{\begin{array}{l} < \rho, \sigma, NL, [stk : arr, ind, val] > \qquad \text{aastore} \quad \Longrightarrow \\ \quad < \rho, \sigma[\forall at \in arr : (at, f_{elems}) \leftarrow \\ \qquad\qquad\qquad \sigma(at, f_{elems}) \cup val], \\ \qquad \text{AllNonTLCond}(NL, arr, val, \sigma), stk > \end{array}}$$

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.

$$\boxed{\begin{array}{l} < \rho, \sigma, NL, stk > \qquad id : \text{newinstance}(cl) \quad \Longrightarrow \\ \quad < \text{rngSubst}(\rho, R_{id/A} \leftarrow R_{id/B}), \\ \qquad \text{tranfer}(\sigma, \mathrm{R_{id/A}}, \mathrm{R_{id/B}}), \\ \qquad \text{replS}(NL, R_{id/A} \leftarrow R_{id/B}), \\ \qquad [\text{rngSubst}(stk, R_{id/A} \leftarrow R_{id/B}) : \{R_{id/A}\}] > \end{array}}$$

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, and transfer to updating the heap state σ. These functions are defined formally in our technical report [16].

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 [23]. 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.* [20] 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 $< \rho, \sigma, NL, [stk : o] >$ 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 loop invariant:

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

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:

Value	:		*Refs* $\mid$ *IntVal* $\mid \bot$
ρ	$\subseteq$	*Var*	$\mapsto$ *Value*
σ	$\subseteq$	*Ref* $\times$ *FieldId*	$\mapsto$ *Value*
stk	$\subseteq$	Stack$<$ *Value* $>$	
Len	$\subseteq$	*Ref*	$\mapsto$ *IntVal*
NR	$\subseteq$	*Ref*	$\mapsto$ *IntRange*

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 *IntVal*s to have at most one term in a variable unknown, one constant term, and zero or more terms in constant unknowns: $au + k_0c_0 + ... + k_nc_n + b$. We perform symbolic arithmetic on *IntVal*s when it makes sense, but certain operations (for example, addition of *IntVal*s involving different variable unknowns) produce the top value $\top_{iv}$ as their result. (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 *IntVal*s: $[iv_1..iv_2]$. This is used only to represent an array's uninitialized indices immediately after the array's allocation. 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, denoted $[]$, 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 *IntRange*s 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.

$$\boxed{\begin{array}{l} <\rho, \sigma, NL, [stk : n], Len, NR> \\ \quad id : \text{newarray}(cl) \quad \Longrightarrow \\ \quad <..., Len[R_{id/A} \leftarrow n], \\ \qquad NR[R_{id/A} \leftarrow [0, n-1]]> \end{array}}$$

The `newarray` instruction is very similar to `newinstance`: it updates the first four state components (not shown) 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.

$$\boxed{\begin{array}{l} < \rho, \sigma, NL, [stk : arr, ind, val], Len, NR > \\ \quad\quad \text{aastore} \quad \Longrightarrow \\ \quad < ..., Len, \\ \quad\quad NR[\forall at \ \in arr : at \leftarrow \text{contract}(NR(at), ind)] > \end{array}}$$

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:

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

3.4 Initial conditions in array analysis

We now discuss initial conditions. 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 initial *Len* map: $Len(R_{arg(i)}) = c_i$.

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:

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

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:

$$< \cdots, \rho(i) = 1, \cdots, NR(R_{id/A}) = [1..] > \quad \text{and}$$

$$< \cdots, \rho(i) = 0, \cdots, NR(R_{id/A}) = [0, 2 * c_0 - 1] > \quad .$$

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 *IntVal*s 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 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 *IntVal*s are merged to accomplish this aim. The merge_intvals function takes three arguments:

$$\begin{array}{lcll} U & \subseteq & int & \mapsto VarUnknown \\ \mu_1 & \subseteq & VarUnknown & \mapsto IntVal \\ \mu_2 & \subseteq & VarUnknown & \mapsto IntVal \end{array}$$

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, mapping the variables to the values they represent in each input state; the existence of these mappings justifies the use of the variable in the merged state to express the values in both the input states.

In more detail, 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 *IntVal*s has a non-zero variable unknown term. If that variable has already been assigned a value, and substituting that value makes the merged *IntVal*s 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 *IntVal*s equal, using the match function described below. If this fails, we must merge to $\top_{iv}$.

The function $\text{match}(i_1, i_2)$ is called only when i_1 has a non-zero variable unknown term $a_1 v_1$, and succeeds only when i_2 has a non-zero variable unknown term $a_1 v_2$ with the same coefficient a_1, in which case it returns an *IntVal* that expresses v_1 as the sum of v_2 and a constant expression.

```
1  IntVal merge_intvals(i1: IntVal, i2: IntVal,
2                       U: int → VarUnknown,
3                       μ1: VarUnknown → IntVal,
4                       μ2: VarUnknown → IntVal) {
5      if (i1 = ⊤iv ∨ i2 = ⊤iv) return ⊤iv;
6      else if (i1 = i2)         return i1;
7      else {
8        if (var_term(i1) = 0)
9          i1, i2 ← i2, i1; μ1, μ2 ← μ2, μ1;
10       let δ = i2 − i1 in
11         if (int_const(δ) ∧ var_term(i1) = 0) {
12           if (U[δ] = null) {
13             let v = new VarUnknown in
14               U[δ] ← v; μ1[v] ← i1; μ2[v] ← i2;
15               return v;
16           } else {
17             let v = U[δ]; d = (i1 − μ1(v)) in
18               assert var_term(d) = 0;
19               return v + d;
20         }} else {
21           let (a1v1 = var_term(i1)) in
22             if (a1 ≠ 0) {
23               if (μ2[v1] ≠ null) {
24                 if (μ2[i1] = i2) { return i1; }
25                 else { return ⊤iv; };
26               } else {
27                 let s = match(i1, i2) in
28                   if s ≠ null {
29                     μ2[v1] ← s; return i1;
30                   } else return ⊤iv;
31             } else return ⊤iv;
32     }}
```

Figure 1. Procedure for merging integer state components

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 merged state.) 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. So we obtain:

$$< \cdots, \rho(i) = v, \cdots, NR(R_{id/A})) = [v..] >$$

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

$$< \cdots, \rho(i) = v+1, \cdots, NR(R_{id/A})) = [v+1..] >$$

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.

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, the validation iteration will merge such components to $\top_{iv}$, which will then allow a fixed point (with less information) to be reached.

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,

[1]To 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.

detail the effectiveness of the optimization, and explores the compile-time cost of the analysis. Finally we consider further techniques (not yet implemented) that might eliminate more barriers.

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.)

Table 4.2 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*. 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.

The percentages in Table 4.2 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.

In our technical report [16] we also show static 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).

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 4.2, 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 consideration suggests further work that would allow a significant number of additional SATB barriers to be eliminated. These techniques are detailed in our technical report [16], and some of them are briefly summarized here.

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 barrier. 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.hasMoreElements`:

```
Entry e = entry;
...
while (e == null && i > 0) {
    e = t[--i];
}
entry = e;  // Frequently executed
```

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

benchmark	Total $\times 10^6$	% elim	% Potential 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

The "frequently executed" store requires no barrier. 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.

Array rearrangements. Another class of optimizations suggested by examination of frequently-executed store sites concerns write barriers within blocks of code (usually loops) that rearrange the elements of 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 the writes. Of course, the arrangement does not actually occur atomically; to nevertheless get some benefits from this observation, we can restrict the direction in which the collector scans object arrays, or try to detect mutator/collector interference.

Let us give a concrete examples. 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 in a sorting routine, and are part of an an idiom that swaps two elements in an array. This swap is a permutation of the array elements. (And the outer loops in which these swaps occur are compositions of permutations, and so are also permutations.) Another set of examples occurs in the **jbb** benchmark, where some of the most frequently-executed store sites are in loops that delete a single element of an object array, by moving all higher elements down by one index. Taken as a whole, such a loop overwrites only one reference value: if it ran atomically with respect to the collector's scanning of the array, only the overwritten value would need to be logged.

If we inform the compiler of the order in which the collector scans object array elements, then one of the two barriers in the swap pattern may be eliminated. In the case of the move-down loop, barriers may be eliminated if the direction of collector array scanning agrees with the direction of object movement. Here we propose using bits in the object array header to allow the code generated for the mutator loop to inform the collector of the required scanning direction.

A more general approach is to use an optimistic concurrency control protocol, detecting mutator/collector interference dynamically and correcting for it as necessary. We would devote bits in the header of an object array to indicate the *tracing state* of the array, one of *untraced*, *tracing*, and *traced*. The concurrent marker would update this state information as it traces arrays. The compiler would recognize the copy-loop idiom of the example, and generate code to log the overwritten `a[index]` value and read the tracing state before and after the loop. 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 (perhaps with mutators stopped, to prevent livelock). This approach is more general because it applies to *any* array rearrangement code pattern. For example, we could eliminate both barriers in the swap idiom with this approach. We could amortize the cost of the check by hoisting them outside of the loop in which the swaps occur.

All of the arguments in this section consider mutator/collector concurrency, but not mutator/mutator concurrency. It turns out that (as far as we can tell) unsynchronized writes to the same array by multiple mutator threads invalidate both classes of optimization discussed above: they allow heap locations to be overwritten without being logged, which compromises the correctness of the SATB marking algorithm. So if we apply these techniques, we must do so only when no unsynchronized writes are possible: perhaps we can prove that the program obeys a locking discipline in accessing the array, or that the array is thread-local; or the compiler might generate code to obtain a special lock during loops that eliminate barriers (this approach raises contention and deadlock issues that require some care).

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 2 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.)

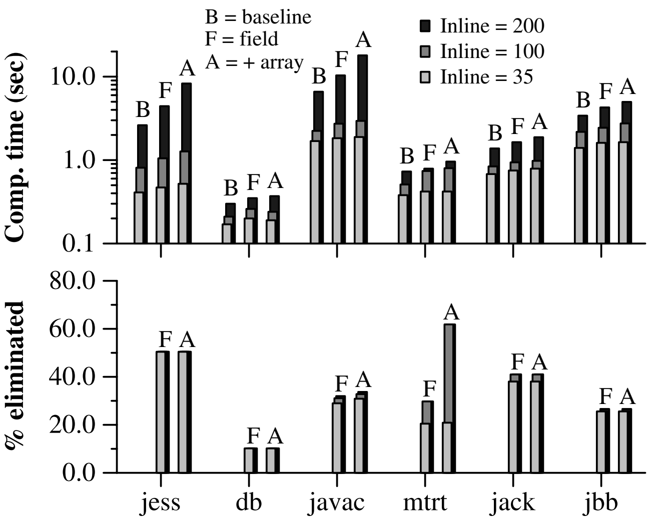

Figure 2. Compile-time cost and analysis effectiveness as a function of inlining level

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 4.2.

Figure 3 shows the effect of the two levels of analysis on compiled code size, at inlining level 100. 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.

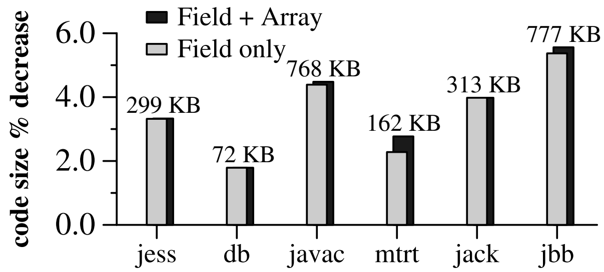

Figure 3. Analysis effect on code size, for various inlining levels

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.

Table 4.5 shows end-to-end throughput on the **jbb** benchmark for three different modes of operation.[2] 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 use 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.

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

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

Table 2. jbb end-to-end barrier cost

5 Related Work

Vecchev and Bacon [22] 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 to an object is often contained within the lifetime of another pointer to to the same object, allowing barriers associated with creation or deletion of the shorter-lived pointer to be elided. 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 [25] and Shuf *et al.* [21] present techniques for removing write barriers supporting remembered set maintenance in garbage collection. Hosking *et al.* [13] 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., [14, 7, 20] and escape analysis (see, e.g., [23, 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 [19, 15]. 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 infrastructure allowed us to identify the most-frequently executed individual store sites whose barriers were not eliminated. Our investigation of the corresponding source code in section 4.3 revealed several interesting paths for future work to improve these 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 with 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 Trademarks

Java and HotSpot are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

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.

[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/products/weblogic/server/jrockit_wp_052303_final.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] 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.

[13] Antony L. Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barrier for orthogonal persistence. In *Proceedings of the Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, August 1998.

[14] 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.

[15] 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.

[16] V. Krishna Nandivada and David Detlefs. Compile-time concurrent marking write barrier removal. Technical Report SML TR–2004–142, Sun Microsystems Laboratories, December 2004.

[17] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

[25] 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.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE