Effective Synchronization Removal for Java

Erik Ruf
Microsoft Research
Redmond, WA 98025
erikruf@microsoft.com

Abstract

We present a new technique for removing unnecessary synchronization operations from statically compiled Java programs. Our approach improves upon current efforts based on escape analysis, as it can eliminate synchronization operations even on objects that escape their allocating threads. It makes use of a compact, equivalence-class-based representation that eliminates the need for fixed point operations during the analysis.

We describe and evaluate the performance of an implementation in the Marmot native Java compiler. For the benchmark programs examined, the optimization removes 100% of the dynamic synchronization operations in single-threaded programs, and 0-99% in multi-threaded programs, at a low cost in additional compilation time and code growth.

1 Introduction

The Java^{TM} programming language [GJS96] provides synchronization constructs (`synchronized` methods and blocks) to permit safe use of concurrently-accessed data structures. These constructs are used pervasively in both the standard libraries and the runtime system. In many cases, a large number of these operations may be safely removed without compromising program semantics, thus improving performance. Removing these operations manually may be inconvenient, error-prone or even impossible.[1]

A dynamic synchronization operation in thread T on an object O is eliminable whenever no other thread T' attempts to synchronize O during the execution of the guarded code. Algorithms for automatic, static elimination of synchronization operations prove conservative approximations of this condition. Existing work falls broadly into two categories. One approach [BS96, FKR$^+$00] proves that the program spawns no threads, making contention impossible and all synchronization operations removable. This is both fast and effective, but has the disadvantages of having no effect on multithreaded programs and being unsound in the presence of notification operations.[2]

The second approach [ACSE99, Bla99, BH99, CGS$^+$99, WR99] proves that the object O cannot escape its creating thread, and thus cannot be subject to contention. This approach fares poorly on programs (even single-threaded ones) where synchronized data structures are stored in static variables; the median synchronization removal ratio for single-threaded programs in existing systems is below 55%. The compile-time costs of escape analysis can also be problematic: context-sensitive dependence-graph-based implementations such as [WR99] can take an hour or more to optimize programs of 10^4 statements [Rin99]. Some approaches improve performance at the cost of precision: [BH99] is context-insensitive and models only a single level of field dereferences, while [Bla99] blurs distinctions between sibling fields. [ACSE99] supplements escape analysis by eliminating synchronization operations that are always guarded by other synchronization operations. This promising extension is limited by cost/precision issues in an underlying pointer analysis, which does not scale up to programs such as `javacup` or `javac`. In general, it is difficult to assess the compile-time costs of escape-analysis-based implementations as only [Bla99] discloses analysis times.

We present a simple yet effective extension to the escape analysis approach, along with a high performance implementation technique. Our optimization achieves a superior degree of synchronization removal at a low cost in optimization time. It handles both `synchronized` methods and blocks, and preserves the Java synchronization, memory, and notification semantics. The distinguishing features of our approach are:

- **Explicit modeling of inter-thread object flow.** Instead of preserving all synchronization on escaping objects, our optimization finds cases where an object is synchronized only by a single thread (not necessarily its creating thread) during program execution, and eliminates synchronization for this case. This additional precision significantly improves the optimization in some cases, yet is obtained at little additional cost.

[1]Most Java-to-bytecode translators implement the string concatenation primitive '+' via a call to a synchronized library method. Short of reimplementing strings, users cannot avoid this behavior.

[2]Removing synchronization guarding a `wait`, `notify`, or `notifyAll` may cause the optimized program to throw a `IllegalMonitorStateException` not thrown in the original program. Such operations make little sense in single-threaded code, but may be present in code fragments or libraries also intended for multi-threaded use.

- **An equivalence class based representation with polymorphic summaries.** Our optimization models aliasing in a flow-insensitive manner by grouping potentially aliased expressions into equivalence classes, and models synchronization behavior as attributes of these equivalence classes. The representation is constructed in a single pass without fixed point operations, and enables context-sensitive analysis and specialization via a simple mapping function. It is sufficiently compact that large programs (10^5 statements) can be optimized without depth limiting or other explicit abstraction of nonrecursive field access paths.

This paper describes our optimization and evaluates its utility on both single- and multi-threaded programs, as well as its costs in terms of compilation time and code expansion.

2 Overview and motivation

2.1 Explicit thread modeling

In contrast with escape based techniques, which preserve synchronization on objects reachable from global state (and thus visible to multiple threads), our goal is to preserve synchronization only on objects potentially synchronized by more than one thread instance during program execution.

This relatively coarse abstraction of the problem yields benefits in several cases. It allows for the removal of global synchronization in singly-executing threads, which can arise when "helper" threads (e.g., asynchronous I/O, user interface code, or watchdog timers) are added to an otherwise single-threaded program having static instances of thread-safe data structures. Another common case involves library abstractions that safely share internal data structures (e.g., buffer pools or graphics resources) via a static lock object, but are then used only by a single thread. In the important degenerate case of purely single-threaded programs, this abstraction renders all synchronization operations removable.

Making this finer distinction requires two extensions to escape analysis. First, it is necessary to track value flow through global state, rather than merely marking globally-reachable values as "escaped." In other words, an alias analysis, rather than an escape analysis, is required. This is a simple extension, since escape analyses already model complex value flow, including aliasing, for local state. Second, the analysis must be able to identify the thread instances in which particular escaping values are synchronized. In our implementation, this is accomplished via a straightforward call-closure analysis that bounds the code executed by thread instances.

A primary limitation of this approach is that it only proves properties that hold for an object's entire lifetime. Thus, it fails to recognize cases where contention is limited to particular program scopes (e.g., fork-join parallelism) or lock scopes (e.g., enclosing synchronization).

2.2 Equivalence class based representation

Like many other systems [Bla99, CGS+99, CmH00, WR99], our optimization achieves context sensitivity by constructing reusable, polymorphic method summaries that can be independently applied at multiple call sites. In most existing work, aliasing is modeled via directed dependence arcs encoding "points-to" relationships, while escape properties are formulated as reachability queries over the resulting dependence graphs.

In an effort to minimize analysis time and space usage, our optimization relies on an equivalence based representation, in which potentially aliased values are forced to share common representative nodes. This choice enables single pass flow-insensitive analysis of any unit of code without the need for iteration to model flow across back arcs (e.g., loops and recursion) in the program's flow graph.

The convenience and efficiency of this representation come at a cost in precision. Within methods, the directionality of flow is lost, resulting in false aliasing between multiple values assigned into a single variable or field. A similar problem arises with recursion, where we give up context sensitivity within a recursive component to avoid iterating the analysis.

3 Algorithm

3.1 Preliminaries

The algorithm is implemented in the Marmot native compilation system for Java [FKR+00]. Marmot implements most Java 1.1 semantics and libraries, but does not support dynamic loading and limits the use of reflection. The resulting "closed-world" assumption enables a number of whole-program analyses, including the construction of a static call graph (using Rapid Type Analysis [BS96] and intraprocedural type propagation) used by our optimization.

For purposes of this exposition, the Marmot intermediate representation can be viewed as a statically-typed three-address format with local variables in static single assignment (SSA) form [CFRW91]. Control operations other than `return` and `throw` are irrelevant as the analysis is flow insensitive. Both `synchronized` blocks and methods are implemented with explicit `monitorEnter` and `monitorExit` primitives.

3.2 Phase 1: Computing thread properties

The first optimization phase identifies thread allocation sites (including those in library code, plus an artificial site for the main thread) and computes two attributes for each site:

- the set of methods potentially executed by the thread being allocated, and
- whether the allocation site (and thus the thread(s) it allocates) can be executed more than once at runtime.

At each thread allocation site, the corresponding `run` method(s) are derived from the thread's type and class hierarchy information. In the special case of a site `t = new Thread(r)`, where `r` is an instance of `Runnable`, we compute `r`'s type via intraprocedural type propagation. In many cases, `r` is allocated in the current method, enabling `r.run` to be precisely identified. A call graph closure analysis finds all methods reachable from the `run` method(s) and and associates them with the allocation site.

A thread allocation site is marked as multiply executed if it is in a loop, is reachable from a non-class-initialization method having multiple or multiply-executed call sites, or is reachable from a `run` method associated with a multiply executed thread allocation site. An annotation mechanism allows sites to be declared as singly-executed; we use this in library code for sites that allocate multiple threads known not to be simultaneously live.

3.3 Phase 2: Building method summaries

The second optimization phase computes

- for each global value (reference constant or static field) and its (transitive) fields and array elements, the set of allocation sites of threads potentially synchronizing the value, and
- for each method, the alias and synchronization effects of the method and its (transitive) callees.

Alias sets The optimization represents runtime values with instances of the *alias set* data structure:

$$aliasSet ::= \bot \mid \langle fieldMap,\ synchronized,\ syncThreads,\ global \rangle.$$

The $\bot$ case indicates a nonreference value, while the tuple case describes a reference value. The tuple elements define properties of the value:

- *fieldMap.* A mapping from fully qualified instance field names to alias sets for the corresponding field values; the distinguished fieldname `$ELT` denotes the contents of an array object.
- *synchronized.* A boolean, true if the value may be the target of a synchronization operation.
- *syncThreads.* For escaping values, a set of thread allocation sites representing the thread instances that may synchronize the value.
- *global.* A boolean, true if the value can be reached from a reference constant or static field (i.e., it escapes). If true, all alias sets reachable via *fieldMap* must also have *global=true*. This ensures that referents of an escaping object also escape.

Alias sets support a *unification* operation that merges two alias sets in place via a union-find data structure [ASU86]. The resulting alias set's attributes are the join of the input attributes under the function, boolean, set, and boolean lattices, respectively. In addition, joining the field maps causes alias sets corresponding to fieldnames present in the domains of both maps to be unified. The unifier is also responsible for noticing when a potentially synchronized value escapes. Thus, unifying a *global* alias set with a non-*global*, *synchronized* alias set causes the *syncThreads* attribute of the result to be augmented with the set of thread allocation sites associated with the current method.[3]

Another operation, *new instance creation*, allows the abstraction of the aliasing and synchronization properties of an alias set. New instance creation returns an alias set isomorphic to an existing one, in which only global alias sets are shared between the old and new instances.

Alias contexts The alias context data structure models the aliasing and synchronization behavior of parameter, normal result, and exception result values transmitted between call sites and methods. It is a tuple

$$aliasContext ::= \langle \langle f_0, ..., f_n \rangle,\ r,\ e \rangle$$

where f_i, r, and e are alias sets corresponding to the parameter, return, and exception values. Alias contexts are used to represent the information both for methods (in which case the f_i represent formal values received from the caller, and r and e represent values returned to the caller) and for call sites (in which case the f_i represent actual values transmitted to the callee, and r and e represent values returned by the callee). We call the former use a *method context* and the latter a *site context*.

Like alias sets, alias contexts support unification and new instance creation. Unification is the pointwise extension of alias set unification to tuples. The alias context returned by new instance creation preserves (recursively) all relationships between the original f_i, r, and e.

3.3.1 Interprocedural analysis

The interprocedural analysis associates each global value with an alias set and each method with a method context. It begins by binding each static field and object constant (e.g., string literal or statically allocated array) to a new alias set with *global=true*. It also constructs initial alias sets for compiler-generated runtime data structures whose initialization is not explicit in the intermediate code (e.g., class objects, interning and reflection tables, etc).

The analysis then partitions the static call graph into strongly connected components (SCCs) and traverses them in bottom-up topological order. Processing an SCC consists of creating an initial method context object for each method in the SCC, then applying the intraprocedural analysis to each of the SCC's methods individually.

3.3.2 Intraprocedural analysis

The intraprocedural analysis ensures that any aliasing or synchronization by the method and its callees is appropriately represented in the method's context and in global alias sets. It begins by associating each formal parameter variable with the corresponding formal alias set from the method context. It then walks the method's statements, unifying alias sets using the rules in Figure 1. Because local variables obey SSA invariants, our implementation saves time and space by binding locals to new alias sets lazily upon use, and implements assignments to unbound locals by updating the binding table instead of performing unification.

Only statements that modify reference variables or values are processed. Primitive operations that induce aliasing cause the alias sets of potentially aliased expressions to be unified. For example, the assignment `x.f = y` (where `x` and `y` are local variables) causes the analysis to unify `y`'s alias set with the alias set returned by $x.fieldmap(f)$, where x is the alias set for `x`. Similarly, analyzing `throw z` unifies `z`'s alias set with those of all relevant handlers (including the returned-exception value e of the method context if `z` could be uncaught by the method).[4]

The synchronization operations `monitorEnter` and `monitorExit` set the *synchronized* property of their argument alias set. In addition, if the argument alias set is *global*, all thread allocation sites reaching the current method are added to the argument alias set's *syncThreads* property.

At method invocations, the analysis constructs a site context S whose formal, return, and exception alias sets

[3]In effect, the optimization records the fact that a thread executes a synchronization operation on a value V at the point where V escapes, not at the point where V is synchronized. Doing so improves precision because V may not escape all threads that synchronize it.

[4]We can safely ignore implicit exceptions from primitives, as these are always newly constructed, unaliased objects without reference fields.

Domains

$v \in V$	local variables
$g \in G$	global values (constants, static fields)
$f \in F$	field names
$a, r, e \in A$	alias sets
$mc, sc \in C$	method, site contexts
$m, p \in M$	methods
$s \in S$	thread creation sites
$t \in T$	types

Analysis State

$GAS : G \rightarrow A$	alias set lookup for globals
$AS : V \rightarrow A$	alias set lookup for locals
$MC : M \rightarrow C$	method context lookup
$CALLEES : M \times V \rightarrow 2^M$	method target lookup
$SCC : M \rightarrow 2^M$	SCC lookup
$TC : M \rightarrow 2^S$	thread creation site lookup

Analysis Rules

statement	action
$v_0 = v_1$ $v_0 = (t)v_1$	$unify(AS(v_0), AS(v_1))$
$v = g$ $g = v$	$unify(AS(v), GAS(g))$
$v_0 = v_1.f$ $v_1.f = v_0$	$unify(AS(v_0), AS(v_1).fieldmap(f))$
$v_0 = v_1[]$ $v_1[] = v_0$	$unify(AS(v_0), AS(v_1).fieldmap(\$\texttt{ELT}))$
$v = \phi(v_0, ..., v_n)$	$\forall v_i \ unify(AS(v), AS(v_i))$
$v =$ `new` T	*no action*
`return` v	$unify(AS(v), r)$
`throw` v	$unify(AS(v), e)$

```
monitorEnter v     $AS(v).synchronized = true$
monitorExit v      $if\ AS(v).global$
                       $AS(v).syncThreads =$
                               $AS(v).syncThreads \cup TC(m)$

$v = p(v_0, ..., v_n)$   $let\ sc = \langle\langle AS(v_0), ..., AS(v_n)\rangle\rangle, AS(v), e\rangle$
                      $\forall p_i \in CALLEES(p, v_0)$
                          $let\ mc = MC(p_i)$
                              $if\ SCC(m) \neq SCC(p_i)$
                                  $let\ mc' = newInstance(mc)$
                                      $unify(sc, mc')$
                              $else$
                                  $unify(sc, mc)$
```

Figure 1: Intraprocedural analysis rules for relevant statement types. The rules assume that the statements being analyzed belong to a method m with method context $\langle\langle a_0, ..., a_n\rangle, \ r, \ e\rangle$, where the AS relation maps formal variables to the corresponding a_i. This description slightly oversimplifies the handling of exceptions and assignments to locals (see text).

correspond to the actual, result, and relevant exception alias sets at the call site. It then iterates over the methods invoked by the call site, performing one of the following two operations:

1. *Nonrecursive target.* The analysis computes a new instance M' of the method context M and unifies it with the site context S.[5] This has the combined effect of (1) reflecting callee-side aliases to the call site, and (2) propagating callee-side properties to the call site. Creating a new instance each time a method is applied prevents the accumulation of call-site-specific information in the method context, allowing context-sensitive analysis.

2. *Recursive target.* In this case, the analysis unifies the method context M and site context S. While this introduces context insensitivity at recursive call sites, it has a large performance benefit in that the analysis does not need to iterate over the entire SCC until a fixed point is reached.[6]

After a method has been analyzed, the analysis drops the reference to the local variable mapping, allowing all alias sets not escaping the method's stack frame to be reclaimed. Subsequent phases requiring information about local variables reconstitute it by reexecuting the intraprocedural analysis.[7]

3.3.3 Example

Figure 2 shows part of a toy vector class and three of its clients immediately prior to synchronization optimization. We use a Java-like syntax for the intermediate code, in which virtual calls have been statically bound, and each statement executes a single operation. In addition, explicit `monitorEnter`, `monitorExit`, and `catch` operations are used to implement the synchronized method `SimpleVector.elementAt` and the synchronized block encircling the ellipsis in method `test3`. The results of the first analysis phase are shown as comments: we will assume that both `T1` and `T2` represent single-instance thread allocation sites.

The second phase begins by assigning a new alias set $\alpha_0 = \langle\{\}, false, \{\}, true\rangle$ to the static variable `SimpleVector.v`, and computes the bottom-up schedule `<init>`, `elementAt`, `test0`, `test1`, `test2`. The method context constructed for `<init>` is $\langle\langle\alpha_1\rangle, \perp, \alpha_3\rangle$, where $\alpha_1 = \langle\{\texttt{elements} \rightarrow \alpha_2\}, false, \{\}, false\rangle$ and α_2 and α_3 have default attributes ($\langle\{\}, false, \{\}, false\rangle$). This context indicates that the formal parameter may have a field `elements` described by α_2, and there is no return value. Neither the formal, any value reachable from it, nor any thrown exception can be synchronized by `<init>`.

[5]Our implementation folds these operations into a single, parallel traversal of M and S.

[6]Given the relative imprecision of the RTA based call graph, SCCs can sometimes be quite large (e.g., most `toString` methods end up in a single SCC). Iteration is further complicated by the size of the alias context data structures, and because convergence is not guaranteed (e.g., the "add to head of linked list" method will grow its list argument on each iteration). We experimented with an adaptive, iteration-based scheme that could degenerate into the direct-unification scheme described above. In most cases, the space bounds were violated before convergence was achieved, so little to no additional precision was obtained.

[7]Because all callee method contexts, even for recursive callees, are complete at reconstitution time, the nonrecursive strategy (item 1 above) is always used.

```
class SimpleVector {
  Object[] elements;
  static SimpleVector v;

  /* invoked by T1, T2 */
  static void <init>(SimpleVector this1) {
    Object[] temp = new Object[10];
    this1.elements = temp;
  }

  /* invoked by T1, T2 */
  static Object elementAt(SimpleVector this2,
                          int index) {
    monitorEnter(this2)
    try {
      Object[] elts = this2.elements;
      Object elt = elts[index];
      monitorExit(this2);
      return elt;
    }
    catch (Throwable t) {
      monitorExit(this2);
      throw t;
    }
  }
}



/* invoked by T1 */
static void test1() {
  SimpleVector v1 = new SimpleVector;
  SimpleVector.<init>(v1);
  Object o1 = SimpleVector.elementAt(v1, 0);
}

/* invoked by T1 */
static void test2() {
  SimpleVector v2 = new SimpleVector;
  SimpleVector.<init>(v2);
  Object o2 = SimpleVector.elementAt(v2, 0);
  SimpleVector.v = v2;
}

/* invoked by T2 */
static void test3() {
  SimpleVector v3 = SimpleVector.v;
  Object o3 = SimpleVector.elementAt(v3, 0);
  monitorEnter(o3);
  try {
    ...
    monitorExit(o3);
    return;
  }
  catch (Throwable t) {
    monitorExit(o3);
    throw t;
  }
}
```

Figure 2: Example program fragments

The method context for `elementAt` is $\langle\langle\alpha_4, \bot\rangle, \alpha_6, \alpha_7\rangle$, where $\alpha_4 = \langle\{\texttt{elements} \to \alpha_5\}, \mathit{true}, \{\}, \mathit{false}\rangle$, $\alpha_5 = \langle\{\texttt{\$ELT} \to \alpha_6\}, \mathit{false}, \{\}, \mathit{false}\rangle$, and α_6 and α_7 have default attributes. In this case, the first parameter may be synchronized, and the contents of its `elements` array may be returned.

The intraprocedural analysis on `test1` finds that the value of variable `v1` may be synchronized, but does not escape either into either `test1`'s method context or a global alias set. Analyzing the first three statements of `test2` yield a similar configuration of locals, with `v2` bound to $\alpha_8 = \langle\{\texttt{elements} \to \alpha_9\}, \mathit{true}, \{\}, \mathit{false}\rangle$, $\alpha_9 = \langle\{\texttt{\$ELT} \to \alpha_{10}\}, \mathit{false}, \{\}, \mathit{false}\rangle$, and `o2` bound to α_{10}, where α_{10} has default attributes. The assignment `SimpleVector.v = v2` unifies α_8 with α_0, producing (due to the unification of global and a nonglobal alias sets) the alias set $\alpha_0 = \alpha_8 = \langle\{\texttt{elements} \to \alpha_9\}, \mathit{true}, \{\texttt{T1}\}, \mathit{true}\rangle$ where $\alpha_9 = \langle\{\texttt{\$ELT} \to \alpha_{10}\}, \mathit{false}, \{\}, \mathit{true}\rangle$ and $\alpha_{10} = \langle\{\}, \mathit{false}, \{\}, \mathit{true}\rangle$. At this point, we know that `v` and `v2` may be aliases holding a value that escapes and is synchronized by a thread allocated at site `T1`, and that the value in `o2` escapes but is not synchronized.

The analysis of `test3` binds v3 to α_0. The application of `elementAt` marks α_0 as synchronized under the thread allocated at `T2` and binds the variable `o3` to α_{10}. The synchronization of `o3` causes α_{10} to be marked as synchronized, but only by `T2`. At the end of phase 2, the method contexts for `<init>` and `elementAt` are as given above, while the alias set for `SimpleVector.v`, `v2`, and `v3` is $\alpha_0 = \langle\{\texttt{elements} \to \alpha_9\}, \mathit{true}, \{\texttt{T1}, \texttt{T2}\}, \mathit{true}\rangle$, where $\alpha_9 = \langle\{\texttt{\$ELT} \to \alpha_{10}\}, \mathit{false}, \{\}, \mathit{true}\rangle$, and $\alpha_{10} = \langle\{\}, \mathit{true}, \{\texttt{T2}\}, \mathit{true}\rangle$.

3.4 Phase 3: Specialization and transformation

The third optimization phase propagates synchronization information from call sites to callees, and uses this information to remove or simplify synchronization operations in callees. It also constructs specialized versions of methods where different call sites allow distinct simplifications.

3.4.1 Interprocedural analysis

The interprocedural analysis processes SCCs in a top-down topological order while maintaining per-SCC queues of specialization requests (in the form of $\langle \mathit{method}, \mathit{methodContext}\rangle$ pairs). The analysis iteratively executes the intraprocedural analysis over all specialization requests for methods in a given SCC until all have been satisfied.

3.4.2 Intraprocedural analysis

The intraprocedural analysis both optimizes the method body (removing or simplifying synchronization operations and redirecting calls to specialized targets) and requests the creation of specialized method bodies. Given a $\langle \mathit{method}, \mathit{methodContext}\rangle$ pair, the analysis begins by executing the intraprocedural analysis of Section 3.3.2, associating each local variable with an alias set. It then walks the method's statements, rewriting synchronization operations and calls as follows.

- **Synchronization operations.** An alias set is said to be *contention free* if its *syncThreads* set is empty or contains a single thread allocation site that executes at most once. Given a statement of the form `monitorEnter(o)` or `monitorExit(o)`, where `o` has alias set o, the analysis checks to see if o is contention free. If so, it removes the statement and, if the program is multi-threaded (i.e., the analysis found a non-artificial thread allocation site), inserts a memory barrier primitive so that later optimizations will obey the Java memory semantics at this point.

- **Call sites.** Given a `call` statement, the analysis constructs a site summary S from the actual, return, and reachable exception handler alias sets. For each target method with method context M, it constructs a new instance M' of M and then walks M' and S in parallel; for each alias set m' in M' that is *synchronized*, the *syncThreadSet* attribute of the corresponding alias set s is added to the *syncThreadSet* attribute of m'.[8] The updated M' is then compared with both M and the method contexts of all existing or pending specializations of the target method, under the condition that two alias sets match if their contention free status is the same. If no match is found, the method is cloned and a request to specialize the cloned method on M' is enqueued. If M' does not match M, the call is rewritten to invoke the appropriate specialized method.[9]

Marmot's intermediate representation is constructed from the Java bytecode, which uses explicit synchronization operations to implement `synchronized` blocks. Because bytecode verification does not prove any invariants about the use of these operations, it is up to the optimizer to find correlated groups of `monitorEnter` and `monitorExit` operations to remove.

Enter/exit correspondences that do not span method boundaries are easily handled by our optimization. Within a method, all potentially aliased objects have identical *syncThreads* attributes, ensuring that all synchronization operations on a particular object will be preserved or eliminated as a whole. All of our benchmarks (and, presumably, all bytecode generated by reasonable Java front ends) have only intra-method enter/exit correspondences.

Correspondences that span multiple procedures are more difficult, as removing or preserving a synchronization operation in one method may require the removal or preservation of one or more corresponding operations in another method. Our specialization strategy handles this by aggressively specializing callees with respect to the contention status of values at call sites, ensuring that caller and (specialized) callee methods will always agree on the removal/preservation choice for any given runtime value. Less aggressive specialization strategies (in which contexts inducing differing contention properties can share a common specialization) must place additional restrictions on synchronization removal.

[8]There is no need to transfer aliasing information from caller to callee, (e.g., by unifying site and method contexts) since all potentially-aliased caller-side expressions will have identical alias sets.

[9]Indirect calls require additional effort, as the call must invoke the specialized clone only for a subset of the receiver objects arriving at runtime. To handle this, new selectors (method names) are introduced at the appropriate points in the class hierarchy; these tail-call the appropriate clones with identical arguments. Such "trampolines" allow specializations to be shared at the cost of additional direct call operations. This overhead is later eliminated by the Marmot code generator, which "inlines" the tail calls into the dispatch tables and removes the trampoline method bodies.

3.4.3 Example

We continue the example of Section 3.3.3 into the final transformation of the optimization. This phase makes no changes to `test1`, as the *syncThreads* attribute of `v1`'s alias set (and its elements) matches that of `this1` and `this2`'s alias sets. The same is true for the invocation of `<init>` in `test2`. Since the *syncThreads* attribute of `v2`'s alias set denotes multiple threads and the corresponding alias set in `elementAt`'s context does not, `test2`'s call to `elementAt` is rebound to a clone, `elementAt2`, with context $\langle\langle\alpha_{11},\bot\rangle,\alpha_{13},\alpha_{14}\rangle$, where $\alpha_{11} = \langle\{\texttt{elements}\rightarrow\alpha_{12}\}, \textit{true}, \{\texttt{T1},\texttt{T2}\}, \textit{false}\rangle$, $\alpha_{12} = \langle\{\texttt{\$ELT}\rightarrow\alpha_{13}\}, \textit{false}, \{\}, \textit{false}\rangle$, and α_{13} and $\alpha 14$ have default attributes. In other words, `elementAt2` is a specialization of `elementAt` that preserves synchronization behavior on the formal parameter `this2`.

The call to `elementAt` in `test3` is also retargeted to `elementAt2`. Local `o3` is found to have the alias set $\alpha_{10} = \langle\{\}, \textit{true}, \{\texttt{T2}\}, \textit{true}\rangle$, which is synchronized, but only by a singleton thread. This means that all three synchronization operations on `o3` are eliminable, so they are replaced by memory barrier primitives.

The `<init>` method is not processed because it has neither synchronization operations nor callees. Processing of `elementAt` finds that `this2` cannot be synchronized (recall that both invocations that passed synchronized arguments were redirected to `elementAt2`), and successfully replaces the synchronization operations on `this2` with barriers. The alias set α_{11} in the context for `elementAt2` is synchronized by two threads, causing all three synchronization operations to be preserved.

3.5 Other issues

3.5.1 Complexity

The worst-case time/space complexity of the optimization is at least exponential in program size. A method m_1 returning a new pair, both of whose arms point to the methods's argument, will have a return alias set with field map $\{\texttt{left}\rightarrow\alpha, \texttt{right}\rightarrow\alpha\}$ where α is the formal alias set. A method m_2 containing a cascade of k calls to m_1 can construct a formal alias set with a field map of size 2^k.

That said, few programs construct large recursive data structures without the use of iteration or recursion. Given that the analysis does not explore recursive paths in control flow graphs or the call graph, exponential cascades of the sort described above are rare. The method-local nature of many objects also limits duplication, as such objects do not contribute to method summaries. In practice, optimization costs are greater than linear in program size but remain manageable ($>$ 7500 stmts/sec) even for our largest benchmarks.

3.5.2 Event notification operations

The Java threading model supports event notification via the `Object.wait`, `Object.notify`, and `Object.notifyAll` methods, all of which require that their `this` argument be locked (otherwise an exception is thrown). Preserving this behavior in the face of synchronization elimination requires some additional effort.

When a notification method is invoked on an object, a boolean *notified* attribute in the object's alias set is set to true. When the analysis finds an otherwise removable synchronization operation whose alias set has *notified*=*true*, it replaces the operation with a specialized version that performs enough bookkeeping to satisfy the notification methods, without actually performing any machine-level synchronization operations.[10]

3.5.3 Object cloning

The method `Java.lang.Object.clone` returns a new object whose reference fields are aliased to the corresponding fields in the original. Representing this using the scheme described above is difficult because the analysis may add fields to the argument object long after the application of `clone` has been processed. Explicitly constructing aliases for all possible fields would be impractical.

Instead, we move the *fieldMap* attribute of the alias set data structure into a separate *contents* object that supports unification and new instance creation. Field and array element operations on alias sets are delegated to the contents object, while unify/new instance operations are performed recursively on the contents object. The `Object.clone` method can then be given a special method context in which the contents objects of the argument and return values are aliased, but the values themselves are not. This avoids false aliasing of the base and clone objects, but is still imprecise on field values that are immediately, strongly updated by a subclass's `clone` method.

3.5.4 Indirect synchronization removal

For a restricted case, our optimization is able to remove synchronization operations on objects subject to contention by multiple threads. In the Marmot runtime, an object's lock and hashcode data are stored in a corresponding, dynamically created extension object. The object extension operation must synchronize on a global lock, rather than on the object being extended, as the object's lock is not yet created.

The analysis described above does not eliminate extension synchronization in multithreaded programs because the object being synchronized (the global lock) is indeed synchronized by multiple threads. We extend the analysis by adding the alias set attributes *extended* and *extThreads*, which mirror *synchronized* and *syncThreads*, but track extension events rather than synchronization events. Extension operations on objects with contention-free *extThreads* sets are redirected to a version that does not perform synchronization.

3.5.5 Single-threaded programs

The thread-allocation-site analysis described in Section 3.2 declares a program single-threaded when it is unable to locate any thread construction sites other than that for the main thread. This knowledge allows our algorithm to avoid the insertion of memory barrier operations. It also enables the use of a garbage collector and runtime system customized for the single-threaded case.

3.5.6 Performance improvements

We lower the optimization's compile time costs by avoiding work that cannot enable the removal of synchronization operations. During the second phase, we identify methods that cannot (transitively) execute synchronization operations. Such methods will never require removal of synchronization operations or retargeting of call sites, and thus can be ignored in the transformation phase. This optimization reduces costs by as much as 50%.

Another optimization lowers memory usage and reduces unification, comparison, and new instance costs by compressing method contexts. An alias set can be removed from a context if (1) it is not synchronized, (2) it is not global, (3) it only appears once in the context, and (4) all of its fields are removable. Restrictions (2) and (3) ensure that aliases are propagated from callees to callers. While the additional context traversal required by compression can increase costs on our smaller benchmarks, it reduces optimization times by as much as 30% on larger ones.

[10]The Jalapeno system [CGS+99] performs a similar optimization dynamically by predicating machine-level synchronization primitives on a bit in the lock object.

```
// a.  original implementation
void f(Object obj) {
  if (obj == null) {
    obj = ⟨default⟩;
  }
  ...
}


// b.  modified implementation
void f(Object obj2) {
  if (obj2 == null) {
    f2(⟨default⟩);
  } else {
    f2(obj2);
  }
}

void f2(obj) {
  ...
}
```

Figure 3: Rewriting a method to avoid aliasing the parameter `obj` with the global-valued expression $\langle default \rangle$.

3.5.7 Avoiding false aliasing

Figure 3(a) shows source code for a common Java idiom in which a null formal parameter value is replaced with a default value prior to the execution of a method body. Our optimization assigns a common alias set to the variable `obj` and the expression $\langle default \rangle$. If $\langle default \rangle$ denotes a global value, the method signature for `f` will be marked as *global*. Since globals are modeled monomorphically, the alias sets of the actual parameters at all of `f`'s call sites will be unified even though `f` induces no callee-side aliasing. In this case, the (otherwise convenient) bidirectional nature of unification-based flow is problematic.

If the identity of the default value doesn't matter, the programmer can avoid this problem by constructing new default values (e.g., via `new` or cloning) as necessary. If identity does matter, or construction is too expensive, one can use the strategy of Figure 3(b). Binding `obj` via parameter passing instead of assignment keeps $\langle default \rangle$'s alias set out of the contexts of both `f` and `f2`, avoiding undesirable aliasing at call sites invoking `f`. For the dual case in which a global value is returned, only the `new`/clone approach can be used. The Marmot library uses these approaches in meth-

name	methods	stmts	dyn syncs	sync ovhd	description
`javac`	1,877	40,758	1.693E+7	15.62%	`javac` compiling `jlex` 4 times
`javacup`	859	21,657	5.926E+5	5.19%	`javacup` generating Java parser
`jess`	1,339	26,172	4.797E+6	5.97%	expert system shell
`jlex100`	536	15,698	1.665E+8	57.66%	`jlex` generating lexer for `sample.lex`, 100 times
`marmot`	8,193	211,332	1.172E+8	10.33%	compile `javac` to native code
`mtrt`	716	16,500	7.486E+5	1.49%	multithreaded ray tracer
`multimarmot`	8,225	212,160	1.183E+8	9.99%	multithreaded compile of `javac`
`plasma`	1,038	17,857	4.159E+4	0.01%	constrained plasma field simulation/visualization
`slice`	1,059	18,697	1.388E+4	0.02%	viewer for 2D slices of 3D radiology data
`volano`	741	13,085	4.623E+7	5.52%	chat room simulator

Figure 4: Benchmark programs. Method and statement counts were performed on the intermediate form just prior to application of the synchronization optimization algorithm.

name	sync operations		
	original	opt	
		complete	partial
javac	1.693E+7	0	3,740
javacup	5.926E+5	0	0
jess	4.797E+6	0	0
jlex100	1.665E+8	0	0
marmot	1.172E+8	0	0
mtrt	7.486E+5	948	0
multimarmot	1.183E+8	7.810E+7	0
plasma	4.159E+4	3,188	0
slice	1.388E+4	8,664	0
volano	4.623E+7	4.610E+5	0

Figure 5: Dynamic synchronization measurements.

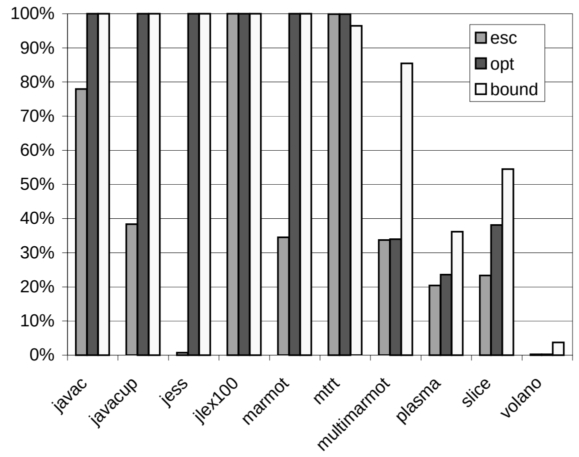

Figure 6: Fraction of synchronization operations removed

ods of the `String` and `StringBuffer` classes when the null value is replaced by the string `"null"`. It also returns clones of string literals in some contexts where returning a single value causes undesirable aliasing.

4 Results

4.1 Benchmark programs

We tested our algorithm on five single-threaded and five multi-threaded programs, described in Figure 4. Most of these programs are well known. `Marmot` is the bytecode-to-native-code compiler described in [FKR+00], while `multiMarmot` is a version of `marmot` reconfigured to perform per-method optimizations (amounting to approximately 25% of total compilation time) in two parallel threads. `Plasma` and `slice` are modified versions of public-domain applet code.[11] `Volano` is the VolanoMark™ 1.0 networking benchmark; we optimized both the client and server but report results only for the client.

The method and statement counts were performed after unreachable methods (in both the benchmark program and the libraries) were removed by a "treeshake" pass. The "synchronization overhead" column approximates the fraction of execution time spent performing synchronization operations in the unoptimized program. We computed this value by measuring the average cost of an executing an empty synchronized block (7.5E-8 seconds, or 58 machine cycles), multiplying it by the number of dynamic synchronization operations, and dividing by the unoptimized execution time.[12] Interestingly, the single-threaded programs execute far more synchronization operations as a function of running time than the multi-threaded programs do.

Testing was performed on a dual-processor 770Mhz Intel Pentium III workstation with 512MB of memory under Windows 2000 Professional. All results are the mean of multiple executions; standard deviations were nominal.

4.2 Synchronization removal

Figure 5 shows dynamic synchronization counts for the original and optimized versions of the benchmark applications. The "partial" category refers to operations that were only partially removed to preserve the notification semantics (c.f., section 3.5.2); only `javac` had removals of this sort.

Figure 6 presents the fraction of synchronizations removed in each of three scenarios. The leftmost column of each bar represents our optimization with all methods treated as executing in all threads, restricting removals to those enabled by escape analysis. As an escape analysis, our system is roughly comparable to existing work, except on `javac`, where it does much better, and `jess`, where it fails almost completely due to an imprecision in the call graph causing false aliasing with a static. The central column rep-

[11] Available from the author.

[12] This figure overestimates the cost of recursive synchronization (no machine level lock is required) and underestimates the cost of initial synchronization (a lock object must be allocated) and contention (queue operations are required). The estimate does not account for secondary effects due to caches, missed optimizations, etc.

name	execution time		
	original	opt	gcopt
javac	8.13	6.44	5.97
javacup	0.86	0.76	0.66
jess	6.03	5.63	5.12
jlex100	21.66	8.37	8.09
marmot	85.10	71.35	61.83
mtrt	3.78	3.73	3.73
multimarmot	88.85	80.67	80.67
plasma	22.41	22.51	22.51
slice	4.64	4.64	4.64
volano	6.29	6.29	6.29

Figure 7: Execution time measurements (user+kernel time in seconds).

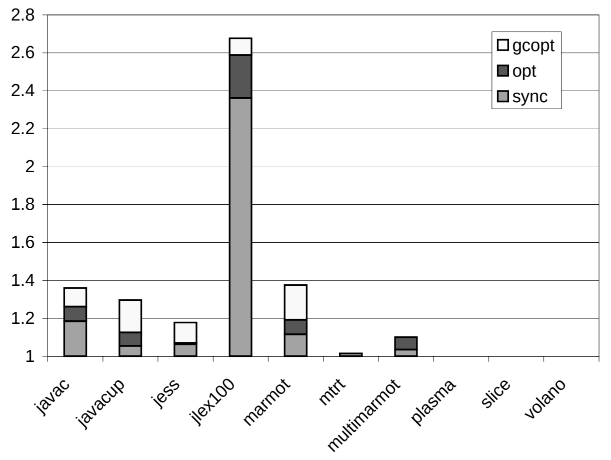

Figure 8: Speedup.

resents the optimization with thread information enabled. This version achieved 100% elimination in single threaded code and improvements over our escape analysis in `plasma` and `slice`.

The rightmost column represents a rough upper bound on the degree of synchronization removal possible using techniques that prove an object to be synchronizable by at most one thread during the object's lifetime. We computed this value by instrumenting the library to count the number of objects synchronized by more than one thread during execution, and assuming that all other synchronizations were removable. In `mtrt`, the optimization improved upon the bound because some synchronization operations referencing multiply-synchronized objects were found to be removable (c.f., section 3.5.4). In `multimarmot`, worker threads performing per-method optimizations never contend for per-method data, but since that data is reached from a shared symbol table, a large number of unnecessary synchronization operations are preserved. All three scenarios fare poorly on `volano`, where 98% of the synchronization takes place on `BufferedInputStream` objects that are synchronized by multiple threads.[13]

[13] An analysis tracking relationships between locks may be able to remove these synchronizations, which appear to be guarded by an escaping, but less frequently synchronized, `DataInputStream` object.

4.3 Execution time

Figure 7 presents execution times for unoptimized and optimized versions of the benchmark programs. For programs found to be single-threaded by our analysis, we examined two strategies. The first performs synchronization elimination only, while the second passes a threading flag to the code generator and runtime system, enabling the use of memory allocation and collection primitives specialized for the single-threaded case.

In Figure 8, each speedup result is divided into three segments. The lower segment represents an estimated speedup computed from the measured synchronization counts and the average-case synchronization cost described in Section 4.1. Together, the lower two segments represent the speedup measured when our optimization is applied. This value exceeds the estimate because synchronization removal enables additional optimization.[14] The sum of all three segments is the speedup measured when the optimization and the single-threaded flag are enabled. In cases where nontrivial speedup is achieved, the additional optimizations account for a significant fraction of the improvement (the majority of the improvement in 4 of 6 cases).

Other than `multimarmot`, which improved by 10%, the multi-threaded benchmarks did not become faster as a result of synchronization removal. `Mtrt` performs all of its synchronization as part of loading its data file, which represents a small fraction of the overall computation. All of the synchronization in `plasma` and `slice` occurs in the AWT libraries; the inner loops of the applets are floating point computations that do not perform synchronization. No performance improvement was obtained on `volano`, as very few synchronization operations were removed.

4.4 Static costs

Figure 9 presents various static measures of our optimization. The absolute costs of the synchronization analysis were quite low (seconds), and represented only a small fraction of overall compilation time.[15] At the same time, by shrinking method sizes (removing synchronization code) and increasing method counts (generating specialized methods), the optimization significantly altered the costs of subsequent phases of the Marmot optimizing compiler. Overall compilation times fell by 79% in `javacup`, but rose by 20% in `multimarmot`. With the exception of `javac`, which contains notification operations, the single-threaded programs did not require specialization. For multithreaded programs, the average number of specializations per method ranged from .08 (`mtrt`) to .20 (`volano`).

The optimization's effect on the amount of code generated[16] varied greatly. In some cases, the removal of synchronization code (which Marmot always inlines) more than compensated for the addition of specialized methods and any

[14] In the single-threaded case, memory barriers are eliminated, enabling a small amount of additional load caching. Most of the benefit comes from additional inlining made possible (under Marmot's size-based heuristics) by reductions in method sizes when synchronization code is removed.

[15] It is worth noting that the analysis allocates a large amount of storage while analyzing a method, much of which becomes dead when the method summary is constructed. Not surprisingly, the optimization performs better under Marmot's generational garbage collector than under its copying collector.

[16] The "code growth" column in Figure 9 refers only to executable code generated for the user program and Java libraries. It does not include C or assembly runtime code, static data, or static metadata.

name	opt time (sec)	frac of comp time	comp time change	specs	tramps	code growth
javac	4.17	6.75%	3.45%	3	0	9.88%
javacup	1.01	2.75%	-79.15%	0	0	-21.76%
jess	1.59	4.55%	-14.05%	0	0	6.82%
jlex100	0.56	3.28%	-8.79%	0	0	-1.03%
marmot	22.00	5.41%	12.99%	0	0	20.35%
mtrt	0.86	3.85%	2.73%	58	49	-0.93%
multimarmot	28.03	6.46%	20.38%	1198	775	4.32%
plasma	1.11	4.04%	10.59%	132	124	8.73%
slice	1.16	3.90%	18.35%	152	124	12.43%
volano	0.73	3.91%	10.69%	148	86	7.8%

Figure 9: Static statistics. Optimization time includes the cost of call graph construction.

additional inlining enabled by method size decreases. For the single-threaded programs, almost all of the code size increase is attributable to the inlining of allocation operations under the single-threaded storage management regime.

5 Related work

This section describes work not addressed in the introduction or in the text.

5.1 Synchronization optimizations

[DR96, DR97] describe schemes for aggregating multiple critical regions guarded by the same lock into a single, larger critical region, and for replacing multiple lock objects with a single lock that guards all of the subobjects' operations. These techniques reduce the number of lock operations performed at the risk of reducing parallelism due to coarser lock granularity. [Tse95] automatically restructures parallel programs to replace barrier synchronization with less expensive operations, or to remove it entirely. In both cases, the transformations were developed for a particular style of thread synchronization produced by a parallelizing compiler, so it is not clear that they are sound for general monitor synchronization as in Java.

Another way to reduce the runtime cost of synchronization operations is to implement them more efficiently. The IBM "thin locks" work [BKMS98] and the Marmot lock implementation [FKR$^+$00] are examples of fast locking mechanisms.

5.2 Related analyses

The construction of abstract summary functions for use in interprocedural analysis dates back at least to the "functional approach" of [SP81]. Alias analysis based on equivalence classes and unification was introduced in [Ste96b, Ste96a]. Recent work in the context of summary-based pointer analysis includes [CRL99, CmH00] and the summary-based escape analyses [Bla99, CGS$^+$99, WR99] discussed in the introduction. [FRD00] explores a combination of equivalence-class-based analysis and procedure summaries that supports higher-order procedures.

6 Future work

While our optimization did very well on single-threaded benchmarks and had some success in the multithreaded case, there is much work to be done for multithreaded programs. Our optimization treats all threads as though they run for the duration of program execution, while many programs (including `multimarmot`) use fork/join strategies in which thread lifetimes are far shorter. This suggests the pursuit of more temporally sensitive strategies. Another open issue is the treatment of threads themselves; all existing analysis, including ours, treat all data reachable from thread objects as escaping. More powerful techniques are required to show that some state held in instance variables of threads remains unaliased.

Our flow analysis is fragile in the presence of cycles in the call graph (e.g., `jess`). Possible improvements include enhancing the Marmot static call graph analysis via context sensitive techniques or modeling of polymorphic data structures. Alternatively, we could avoid the use of a static call graph by encoding method dispatch into types and using the instantiation constraint based flow analysis technique of [FRD00]. A third option would allow limited use of sets of *aliasSet* to represent values in cases where unification yields false aliases [SH97].

The high speed of our analysis opens several opportunities. One possibility is to use the analysis as a preprocessing phase to reduce the cost of a more precise model. Another is an iterative, pessimistic call graph optimization in the style of [HH98], in which the analysis is used to model class sets, which are used to improve the call graph, enabling reanalysis, etc., until convergence is achieved.

We plan to apply equivalence-class-based summarization techniques to other interprocedural problems such as stack allocation, memory disambiguation, and type propagation.

Finally, we believe it is important to continue the search for and the development of additional, more realistic multithreaded programs for use in the design and testing of optimizations.

7 Conclusion

We have described an effective, efficient technique for statically removing unnecessary synchronization operations from Java programs. The distinguishing features of this approach are

- the use of thread closure and alias analyses rather than escape analysis, enabling more precise modeling of value flow in the face of global variables and multiple threads, and

- the use of equivalence class based method summaries, enabling simple, fast, non-fixed-point, context-sensitive analysis and transformation.

Our optimization handles both `synchronized` methods and blocks, and preserves the Java synchronization, memory, and notification semantics. Our experiments, performed in the context of an optimizing compiler, demonstrate improvements in dynamic synchronization counts and execution time in both single- and multi-threaded programs, at a reasonable cost in compilation time and code growth.

Acknowledgments

We thank the members of the Advanced Programming Languages group at Microsoft Research for their support of the Marmot compiler infrastructure, Bjarne Steensgaard for finding unifier bugs, and the anonymous referees for their suggestions.

References

[ACSE99] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *SAS'99*, LNCS. Springer-Verlag, September 1999.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, USA, 1986.

[BH99] J. Bogda and U. Hölzle. Removing unnecessary synchronizations in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[BKMS98] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[Bla99] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[BS96] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 324–341, October 1996. Published as Proceedings OOPSLA '96, ACM SIGPLAN Notices, volume 31, number 10.

[CFRW91] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CGS+99] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[CmH00] B.-C. Cheng and W. mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[CRL99] R. Chatterjee, B. G. Rynder, and W. A. Landi. Relevant context inference. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, January 1999.

[DR96] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, LNCS 1239, pages 285–299, August 1996.

[DR97] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 187–200, 1997.

[FKR+00] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[HH98] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, June 1998.

[Rin99] M. Rinard. Personal communication. 1999.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–284. Prentice-Hall, 1981.

[Ste96a] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150, April 1996.

[Ste96b] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Tse95] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 144–155, July 1995.

[WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.