

Removing Unnecessary Synchronization in Java

Jeff Bogda and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{bogda,urs}@cs.ucsb.edu

Abstract

Java programs perform many synchronization operations on data structures. Some of these synchronizations are unnecessary; in particular, if an object is reachable only by a single thread, concurrent access is impossible and no synchronization is needed. We describe an interprocedural, flow- and context-insensitive data-flow analysis that finds such situations. A global optimizing transformation then eliminates synchronizations on these objects. For every program in our suite of ten Java benchmarks consisting of SPECjvm98 and others, our system optimizes over 90% of the alias sets containing at least one synchronized object. As a result, the dynamic frequency of synchronizations is reduced by up to 99%. For two benchmarks that perform synchronizations very frequently, this optimization leads to speedups of 36% and 20%.

1. Introduction

Java provides synchronization constructs to allow multiple threads to access shared data structures safely. The standard Java Development Kit (JDK) library uses these constructs wherever possible, making all its data types thread-safe. As a result, typical Java programs often execute many synchronization operations per second. These synchronizations incur significant overhead, prompting researchers to focus on efficient implementations of the Java synchronization primitives [4]. In contrast to this research, we focus on the complementary goal of completely eliminating synchronization operations where possible, thereby reducing the overhead to zero for these situations.

Our optimization rests on a simple observation: an object reachable from a single thread does not need to be synchronized. Consider the example in Figure 1, which depicts a Java program in which two threads access objects in a shared heap. Both Thread 1 and Thread 2 can access object C, but only Thread 1 can access object A or B. Assume that all objects are instances of the class `Hashtable`, which has a synchronized `put` method. Clearly, when invoking `put` on object C, synchronization is essential since both threads may simultaneously try to insert an element into the hash table; for this reason, the `put` method must be declared synchronized.

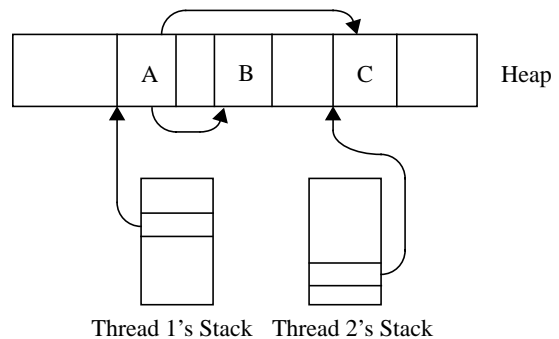


Figure 1. A multi-threaded Java program

On the other hand, invoking `put` on A or B is safe even without synchronization since only Thread 1 can access these objects. Nonetheless, the program will still perform a synchronization every time it invokes `put` because all hash tables are of the same class, and they therefore share code. How can we avoid synchronizing in the latter case and still provide synchronization for the former case?

Current systems shift the burden of optimization to the programmer by requiring him to recognize such situations and to change the code manually to use alternative unsynchronized versions of the appropriate methods or classes. Since reachability is a global property, manually verifying that an object is only reachable by one thread is tedious and error-prone, and subsequent program changes may invalidate such an optimization. In addition, by creating unsynchronized versions of data structures, the programmer duplicates code, creating undesirable redundancies in interfaces and implementations.

We have developed a program optimization that automatically detects situations where synchronization can be safely suppressed and that rewrites the affected program parts to eliminate the unnecessary synchronizations. It achieves the benefits of a manual optimization, while concealing the details and relieving the programmer of any responsibility. On a suite of ten Java programs, including the programs from the SPECjvm98 suite, our system optimizes over 90% of all candidate situations for every program, leading to reductions in the run-time frequency of synchronizations of up to 99%.

In short, Java synchronization and especially the need to optimize synchronizations (reviewed in section 2) has caused us to develop a program optimization (described in section 3) that lessens the synchronization overhead. Consisting of an analysis step (described in section 4) and a transformation step (described in section 5), our optimization is effective when applied to our benchmark suite (described in section 6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '99 Denver, CO USA

2. Motivation

One of Java's strengths lies in its support for multi-threaded programming at both the language and the library level. Each Java thread has its own stack but shares a common heap that houses all objects. By adding the keyword `synchronized` to a method's signature, a programmer can prevent multiple threads from simultaneously invoking that method on an object. In order to enforce this synchronization, the Java virtual machine (JVM) acquires a monitor lock on entry to each synchronized method and releases it on exit. The monitor locks are reentrant, which means the same thread can lock an object multiple times without blocking. For a complete description of synchronization in Java, we refer the reader to the Java and JVM definitions [15][24].

In addition to the language level, Java supports multi-threaded programming in its libraries. The libraries are thread-safe, which means programs can safely use them in a multi-threaded setting. Examples of thread-safe library classes include all container classes (e.g., lists, vectors, and hash tables), files and streams (for concurrent I/O from a shared file or network connection), and windowing classes (for concurrent screen updating).

Locking, while desirable from a software engineering point of view, exacts a run-time overhead. Common applications—even if single-threaded—can execute millions of synchronized methods, with each invocation costing dozens or hundreds of cycles. Krall *et al.* report the cost of synchronization as between 0.4 and 40 microseconds, depending on the virtual machine and the application [23]. Our own measurements of the Solaris JDK 1.2 Production VM, which employs a very efficient synchronization scheme, reveal a cost of 0.14-0.19 microseconds per synchronized call on a 400 MHz processor. In Marmot [12], a research compiler system arguably comparable to Microsoft Visual J++, five single-threaded medium-sized applications spend between 26% and 60% of their execution time in synchronization. In summary, it is evident that synchronization incurs a non-negligible overhead in current Java implementations. Heydon and Najork acknowledge the slow-down caused by excessive synchronization in the core libraries and program around it, paying close attention to the thread-safe `StringBuffer`, `InetAddress`, and stream classes [16]. The next section provides an overview of an optimization targeted at automatically removing as much of this overhead as possible.

3. Overview of Synchronization Elimination

Objects that are only accessed by a single thread do not have to be synchronized.¹ For single-threaded programs this condition encompasses every object, and for multi-threaded applications it includes data structures reachable from exactly one thread, as observed above. In general, since it is impossible to know at compile-time a program's flow of control, it is also impossible to know the precise set of objects reachable by exactly one thread.

With the help of a global data-flow analysis, however, a compiler can prove that certain objects will be thread-local. Our optimization is based on such an analysis and draws from simple notions of escape and shape analysis. It works by conservatively identifying those objects that may *not* be thread-local and then optimizing the remaining ones. Before going into a detailed technical description of the analysis, we give an intuitive overview

and discuss the conditions under which an object ceases to be thread-local.

The simplest and most conservative condition asserts that any object whose reference is stored into the heap can be accessed by multiple threads; we term such objects *s-escaping*, or stack-escaping. Conversely, an object reachable only from a local variable on the stack is *s-local*. Object A in Figure 1 is s-local, while objects B and C are s-escaping. After initially being placed on a thread's private stack, an object reference can escape into the heap by being assigned to a class variable or to a field of an object. For example, if `r` is a reference to a local object (e.g., after the statement `r = new Object`), then the assignment `p.x = r` makes the object s-escaping. The first stage of our analysis uses this notion of s-escaping to find thread-local objects.

However, the above definition fails to find some provably thread-local objects. Consider object B in Figure 1. Since its reference is stored into object A, our first approximation deems it s-escaping and not optimizable; however, since A is s-local, the store of B into A is not an "escaping" store—B is still only reachable from Thread 1's stack. In other words, since A is s-local, B should still be optimizable. This observation leads to the second, more precise analysis algorithm.

Recognizing that an object stored into a field of another object may still be optimizable, we extend our first algorithm by asserting that objects reachable only from s-local objects are still thread-local. Ideally, an analysis would recognize all objects that are transitively reachable only from an s-local object. Unfortunately, in the presence of recursion and recursive data structures, this is a difficult reachability problem. Our goal is to keep the analysis simple while adequately detecting thread-local objects.

To simplify the problem, we assume that any object reachable via more than one dereference from an s-local object is not optimizable. This assumption both allows the analysis to ignore recursive data types and keeps our flow-insensitive analysis from rapidly losing precision that stems from imprecise alias information. Thus, we deem any object stored into a class variable or into a field of an s-escaping object not optimizable. We call such objects *f-escaping* (for field-escaping, since the object escapes through a field of another object). The second variant of our analysis detects f-escaping objects.

4. Analysis

Our system statically detects thread-local objects with an interprocedural, flow- and context-insensitive, constraint-based whole-program analysis. As discussed in the previous section, it executes in two stages: the first stage detects s-escaping objects (objects whose references appear in the heap), and the second stage detects f-escaping objects (objects reachable from global objects or by more than one level of indirection from s-local objects). In other words, we determine first when a reference escapes the stack and second when a reference escapes local objects.

We present the two stages as constraint problems. Since the analysis concerns itself only with instructions involving reference variables, we ignore all other operations. Furthermore, to simplify

¹ We say that an object is synchronized if its class has at least one synchronized instance method.

the analysis we assume without loss of generality that the program consists solely of instructions of the following forms:

$x = y$	assignment
$x.f = y$	field assignment
$y = x.f$	field reference
$C.f = y$	assignment into static (class) variable
$y = C.f$	reference to static (class) variable
$x = \text{new } T$	object creation
$\text{foo}(a_0, \dots, a_n)$	method or static call ¹

Here x, y, a_0, \dots, a_n are stack variables, C is a class name, f is a field name, foo is a method name, and T is a class type. By introducing temporary variables, our implementation transforms stack-based Java bytecodes into this canonical form. For example, in the conversion

$\text{aload } x$	$t_0 = x$
$\text{getfield } f$	$t_1 = t_0.f$
$\text{astore } y$	$y = t_1$

the temporary variables t_0 and t_1 capture the stack assignments of the `aload` and `getfield` instructions. We refer the reader to [21] or [12] for more information regarding similar conversions.

Some Java constructs do not map trivially to this form and require special attention. To accommodate exception handling, we treat `throw` and `catch` statements as static field accesses. Assuming the class `Exception` has a static field named `escapes`, we view statements of the form `throw x` as `Exception.escapes = x` and statements of the form `catch x` as `x = Exception.escapes`. Similarly, since the compiler cannot generally distinguish two array elements, we view all array cell accesses as accesses to the single fictitious array instance field. The statement `a[i] = x` thus transforms to `a.array = x`.

We also handle parameter passing and method return values specially. First, virtual methods take their receiver as the first argument. Second, to achieve the same effect as `return x`, we assume that each method contains a local reference variable named `return`. Conceptually, a method does not return an object but merely stores the object into the method's `return` variable. Therefore, letting `returnm` denote the `return` variable of method m , we transform the statement `return x` in m to `returnm = x`. Third, we model the destination of a method call as an additional argument to the call. For example, these rules change the call `o = hashtable.put(x,y)` into `put(hashtable,x,y,o)`. From the other end of the call site, the method prototype looks like `void put(Hashtable this, Object key, Object value, Object returnput)`. Any action involving actual and formal parameters also applies to the destination and `return` variables. The fact that the destination parameter behaves differently than the other parameters (namely, that it is passed by reference) is handled explicitly in the constraints. In all constraints, a_i denotes the variable used as the i^{th} actual argument, and p_i denotes the corresponding formal argument.

Because of dynamic dispatch we do not generally know a program's exact static call graph. The analysis conservatively approximates the call graph by connecting each call site to `methods-invoked(m)`, the set of all methods that are legal targets.

¹ Note that in our canonical program form actual arguments must be variables, not expressions.

Java's method invocation rules (section 15.11 in [15]) and the program's complete class hierarchy determine the elements of `methods-invoked(m)`.

We base the analysis on our notion of an alias set. The *alias set* of a variable x within method m , $AS(x)$, is the set of all variables appearing in m that can alias x . We say that variable y *aliases* x if at some time t_1 y references an object referenced by x at some time t_2 . Intuitively, an alias set represents objects that flow through a method and consists of the set of local variables, including formal parameters, that may refer to those objects. The constraint $AS(x) \supseteq \{x\}$ must hold. Since our rules impose set equivalence on alias sets, two alias sets $AS(x)$ and $AS(y)$ will either be disjoint or identical. The predicate `connected(x,y)` denotes the latter situation.

4.1 Detecting s-escaping objects

The first phase of the analysis detects s-escaping objects, or alias sets. For a local variable x , we define the set *s-escape*(x) either to be empty (representing false) or to contain true (T). Phrasing this boolean property as a set allows us to express all rules as set constraints. Initially, the set is empty, and as the analysis proceeds, it becomes $\{T\}$ if x is aliased with a variable stored into the heap or loaded from the heap. No set ever changes from $\{T\}$ back to empty because all constraints are inclusion constraints.

Table 1 lists the rules to construct the constraints between alias sets. Our implementation applies these rules, ignoring control flow, to each statement within a method, starting with the `main` method. After examining all statements, it has constructed (not necessarily complete yet) alias and s-escape sets for all local variables, including formal parameters. Upon encountering a call site, the analysis processes the statements within the callee, if it has not yet analyzed that method. Then, guided by the rule for method invocations, it applies the callee's set information pertaining to its formal parameters to the actual arguments. See [31] for similar techniques of transferring summary information from the called method to the caller. To handle program recursion, the entire process iterates until no set changes.

Since the constraints follow directly from Java's semantics, we discuss each rule only briefly. For convenience, we define the operator \equiv to denote set equivalence, in order to distinguish it from the comparison and assignment "equals." Specifically, $AS(x) \equiv AS(y)$ implies $AS(x) \supseteq AS(y)$ and $AS(y) \supseteq AS(x)$.

For the assignment $x = y$, the analysis merges the alias sets of x and y . If one variable is marked s-escaping, the other inherits the property. Because our analysis is flow-insensitive and hence ignores loop constructs, the rules apply to both x and y . A flow-sensitive analysis, on the other hand, could merely propagate the properties of y into x but then would require additional rules for the merging of control flow.

The instance field accesses $y = x.f$ and $x.f = y$ imply that the heap contains a reference to y ; hence the analysis deems y 's alias set s-escaping. In this phase of the analysis, we treat static field accesses identically.

Across the method call `foo(a0, ..., an)`, we simply propagate the constraints imposed on the formal parameters within `foo` to the actual parameters: if formal parameter p_i escapes, then actual parameter a_i also escapes. Similarly, if a parameter is returned, the analysis merges the alias set of the corresponding actual parameter and the alias set of the destination. We handle native methods

$x = y$	Intuitively, x and y must have the same alias sets and s-escape property. For convenience, we let $AS(x) \cong_1 AS(y)$ denote the following constraints: $AS(x) \equiv AS(y)$ $s\text{-escape}(x) \equiv s\text{-escape}(y)$
$x.f = y$ $y = x.f$	The program stores or loads y from the heap, so y is s-escaping. $s\text{-escape}(y) \supseteq \{T\}$
$C.f = y$ $y = C.f$	The program stores or loads y from the heap, so y is s-escaping. $s\text{-escape}(y) \supseteq \{T\}$
$foo(a_0, \dots, a_n)$	For each method that the program may invoke at run-time, the s-escape property must flow from the formal parameters to the actual parameters and any returned parameters must be equated with the destination (a_n), if one exists. $\forall g \in \text{methods-invoked}(foo)$ $\forall i \in [0 \dots n]$ $s\text{-escape}(a_i) \supseteq s\text{-escape}(p_i)$ $\forall i \text{ such that } \text{connected}(p_i, \text{return}_{foo})$ $AS(a_i) \cong_1 AS(a_n)$

Table 1. Constraints for Phase 1

specially. For the commonly used native methods in the core library classes (e.g., `Object.equals`) we hand-construct the alias sets for each formal parameter, marking it s-escaping if the parameter is known to escape the stack. For all other native methods, we conservatively assume that all parameters are s-escaping.

4.2 Detecting f-escaping objects

After identifying s-escaping alias sets, we run the second phase of the analysis, which detects f-escaping alias sets. For this phase, we extend the notion of alias sets to include fields of objects. That is, $AS(x.f)$ includes all local variables that may reference the object accessible via field f in the object referenced by x . We define the set $f\text{-escape}(x)$ to be either empty or $\{T\}$, where the latter indicates that x , or an alias of x , is stored into a field of an s-escaping object.

The constraints, quite similar to those of the first phase, appear in Table 2. The assignment constraint merges the alias sets of x and y as well as any alias sets reachable via one field access. Let C_x and C_y be the static classes of variables x and y , respectively. Then $fields(x, y)$ is the set of all pointer field names in C_x , C_y , and any subclass of C_x or C_y .

The field accesses $y = x.f$ and $x.f = y$ imply that y aliases $x.f$, so we connect their alias sets. If x is s-escaping, however, we mark y as f-escaping, since it is either reachable from a class variable or by more than one level of indirection from an object. If the algorithm equates two alias sets, it will not equate their fields because y will be s-escaping, causing the if-test in \cong_2 to fail. This rule limits the number of dereferences to one. The static field accesses $y = C.f$ and $C.f = y$ naturally cause y to become f-escaping since y references an object held by a global variable and hence reachable by multiple threads.

Across a method call, an actual parameter and its fields inherit the f-escape properties of the corresponding formal parameter and fields. In addition, if the fields of two formal parameters may refer to the same object, we equate the fields of the corresponding actual parameters. Furthermore, as we did in the first phase, we assume native methods have pre-determined characteristics based on their known behavior. For instance, the native `arraycopy` method in

class `System` passes array elements from one parameter to another, so $AS(p_0.array) \cong_2 AS(p_2.array)$.

The above constraint problems must converge to a fixpoint since the number of local variables is finite, the alias sets only grow, the s- and f-escape sets are monotonic, and the number of indirections is limited to one. When this fixpoint is reached, we have computed the alias sets and have determined which sets are f-escaping.

4.3 Example

Although the rules may appear slightly daunting, the analysis is quite simple. This section walks the reader through the analysis of method `m` in the hand-concocted code given in Table 3. The first column gives the sample code, written in Java source code to aid in compactness and understandability. The code snippet creates two `Handle` objects, sequentially wraps them around two other freshly created objects, and stores the second handle in a static variable. The second and third columns depict the alias sets (circles) resulting from the first and second phase of the analysis, respectively. We write inside each circle the variables in the corresponding alias set, we label escaping alias sets with an “s” or an “f,” and we denote field dereferences with an arrow from the container object to the contained object.

To analyze method `m` for s-escaping alias sets, our algorithm analyzes `m`’s statements in lexical order. The first time it encounters `createHandle`, it pauses to analyze this method. The analysis ignores the `new` statement within `createHandle`, so no action occurs to the lone alias set of this method. Continuing in `m`, the analysis encounters `attach`. The `attach` method is slightly more interesting in that it assigns `o` to an instance field, causing the alias set of `o` to be s-escaping. Mapping `attach`’s parameter structure to the caller, the analysis marks the alias sets of `h1` and `h2` s-escaping. Finally, the statement `globalHandle = h2` causes `h2`’s alias set to be s-escaping. The phase repeats but finds no changes to the alias set structures or classifications.

The second phase steps through the entire program again and determines f-escaping alias sets. Again, nothing interesting occurs until it comes to the statement `h1.attach(o1)`. It first analyzes `attach` and attaches `o`’s alias set to `this`’ alias set via the field `ref`.

$x = y$	<p>Variables x and y, as well as any fields of x and y, must have the same alias sets and f-escape property. As we did in Phase 1, we let $AS(x) \equiv_2 AS(y)$ denote the following recursive constraints:</p> $AS(x) \equiv AS(y)$ $f\text{-escape}(x) \equiv f\text{-escape}(y)$ <p>if $s\text{-escape}(x) \cup s\text{-escape}(y) = \emptyset$:</p> $\forall f \in \text{fields}(x,y)$ $AS(x.f) \equiv_2 AS(y.f)$
$x.f = y$ $y = x.f$	<p>Since we must keep track of aliases of at most one dereference, we merge the alias sets of $x.f$ and y, if x is not s-escaping, and mark y f-escaping otherwise. We denote the constraints below by $\text{FieldAccess}(x,f,y)$.</p> <p>if $s\text{-escape}(x) = \emptyset$:</p> $AS(x.f) \equiv_2 AS(y)$ <p>otherwise:</p> $f\text{-escape}(y) \supseteq \{T\}$
$C.f = y$ $y = C.f$	<p>Variable y is f-escaping, because it is stored into or loaded from a variable reachable by more than one thread.</p> $f\text{-escape}(y) \supseteq \{T\}$
$\text{foo}(a_0, \dots, a_n)$	<p>For each method that may be invoked at run-time, we have four constraints. First, we propagate the f-escape property from the formal parameters to the actual parameters. This includes one level of field accesses if the actual parameter is not s-escaping. Second, if it is possible for a formal parameter to be returned, we equate the actual parameter and the destination. Third, if it is possible for a parameter to be stored into a field of another parameter, we model this in the caller by treating it as if it were a field access. Last, we equate fields of one parameter with fields of another parameter if they are connected in the callee. The tests for s-escaping variables merely restrict accesses to one level of indirection.</p> $\forall g \in \text{methods-invoked}(\text{foo})$ $\forall i \in [0 \dots n]$ $f\text{-escape}(a_i) \supseteq f\text{-escape}(p_i)$ <p>if $s\text{-escape}(a_i) = \emptyset$:</p> $\forall f \in \text{fields}(p_i)$ $f\text{-escape}(a_i.f) \supseteq f\text{-escape}(p_i.f)$ $\forall i \text{ such that } \text{connected}(p_i, \text{return}_{\text{foo}})$ $AS(a_i) \equiv_2 AS(a_n)$ $\forall f, p_i, p_j \text{ such that } \text{connected}(p_i.f, p_j)$ $\text{FieldAccess}(a_i.f, a_j)$ $\forall f, h, p_i, p_j \text{ such that } \text{connected}(p_i.f, p_j.h)$ <p>if $s\text{-escape}(a_i) \cup s\text{-escape}(a_j) = \emptyset$:</p> $AS(a_i.f) \equiv_2 AS(a_j.h)$ <p>otherwise if $s\text{-escape}(a_i) = \emptyset$:</p> $f\text{-escape}(a_i.f) \supseteq \{T\}$

Table 2. Constraints for Phase 2

Mapping this structure back to the caller, the analysis links actual arguments $h1$ and $o1$ via the field ref . The second call to `attach` does not require any analysis of `attach` but merely a mapping of the structure of its parameters. In this case, however, since $h2$ is marked s-escaping from phase one, argument $o2$ is marked f-escaping because the object pointed to by $o2$ is reachable via the potentially shared variable `globalHandle`. Finally, the last statement in m makes the alias set of $h2$ f-escaping.

The alias sets marked f-escaping are not considered for optimization. Thus we only consider the alias sets of $h1$ and $o1$. Since class `Object` does not have any synchronized methods,¹ we disregard $o1$. As a final test, we check if $h1$ is reachable from an alias set of any formal parameter of m . Since it is not, m 's scope begins before and ends after the lifetime of any objects referenced by $h1$, and we can safely deduce that $h1$ references only thread-

local objects. Any objects $h1$ references do not need to be synchronized; they are *optimizable*.

5. Transformation

We transform the `class` files of the original program in such a way that no synchronized method will be invoked on an optimizable object. In general, we cannot simply remove the `synchronized` attribute from a method, because some invocations may need to be synchronized while others may not. Many possible transformation techniques exist, each with its advantages and disadvantages. We

¹ During the second phase, we also attach the class name of the referenced object to the corresponding alias set. This allows us to recognize that $AS(o1)$ refers to an object of type `Object`. In addition, we attach to the corresponding alias set the execution paths to any allocation sites. This enables us to perform the code transformation described in the following section.

Sample Code	Alias Sets after Phase 1	Alias Sets after Phase 2
<pre> class Sample { static Handle globalHandle = null; static void m() { Handle h1 = createHandle(); Handle h2 = createHandle(); Object o1 = new Object(); Object o2 = new Object(); h1.attach(o1); h2.attach(o2); globalHandle = h2; } static Handle createHandle() { return new Handle(); } } class Handle { private Object ref = null; synchronized void attach(Object o) { this.ref = o; } } </pre>		

Table 3. Sample application of the analysis

sought a transformation that added no run-time overhead and could be applied as easily as possible without modifying the JVM.

We chose one that clones classes and call chains leading to allocation sites. Since an optimizable object does not require synchronized methods, our transformation changes the class of this object to a new class with no synchronized methods. We make this new class a subclass of the original class, copying its parent’s synchronized methods but making them unsynchronized. Also, we copy down the parent’s constructor signatures and fill the bodies with calls to the original constructor of the parent. The following program fragment shows the unsynchronized `Handle` class, which first appeared in Table 3.

```

class Handle$Unsync extends Handle {
    public Handle$Unsync() { super(); }
    public void attach(...) { /* same code as in parent */ }
}

```

By letting the new class extend the original class, the former can be substituted for the latter, and no method accepting an optimizable object need be modified.

Some Java features complicate this task. Since one cannot subclass final classes, the transformation removes any final modifiers before subclassing. Also, since subclasses cannot access private fields of a parent class, we change these fields’ accessibility to protected.¹ For example, the `ref` field in class `Handle` becomes protected. Last, any program relying on an object’s exact class name may work

incorrectly as a result of renaming the class; however, this did not occur in our benchmark suite.

In order to use the unsynchronized version of a class, we modify the creation site of an optimizable object so it constructs an instance of the corresponding unsynchronized subclass. In our example, the statement `new Handle` becomes `new Handle$Unsync`. If an allocation site constructs synchronized versions in some cases and unsynchronized versions in others (perhaps in factory methods), we duplicate the method and rewrite the clone to construct the unsynchronized version. The following code shows an example of these transformations.

```

h1 = createHandle$Clone1()           // h1 is optimizable
h2 = createHandle()                   // h2 is f-escaping
...
h1.attach(o1);                       // unsynchronized call
h2.attach(o2);                       // synchronized call

static Handle createHandle$Clone1 { // cloned method
    return new Handle$Unsync();
}

```

Since `h1` references an optimizable object, it receives an instance of `Handle$Unsync` from the cloned method `createHandle$Clone1`.

To see why this transformation reduces the number of synchronizations, consider the lines `h1.attach(...)` and `h2.attach(...)`. In the original program each call is synchronized. In the optimized version, however, dynamic dispatch correctly chooses between the synchronized and unsynchronized versions: the first call finds the unsynchronized `attach` method in class

¹ Although this solution may cause a name conflict with another field, we have not encountered this situation.

Handle\$Unsync, while the second triggers the synchronized version in Handle. Hence the optimized code performs fewer synchronizations. Note that we do not need to alter these call sites or any code to which optimized objects are passed.

To carry out this transformation, we must identify both the allocation site that should construct an unsynchronized object and the execution path from the method that recognizes the object as thread-local to the allocation site. This can straightforwardly be done by attaching the locations of `new` instructions to the appropriate alias sets as well as by propagating this information across method calls. If the allocation site resides more than one call away from the site using the created object, we clone the entire call chain. For the applications used in our experiments, only two call chains deeper than four were cloned.

Cloning faces two difficulties worth mentioning. First, since the names of constructors cannot be modified, to clone a constructor we create another constructor that takes a different set of arguments. Depending on the number of times the constructor has already been cloned, we add additional `CloneFiller` arguments, where `CloneFiller` is an empty class that is never instantiated. The allocation site passes in null values as these extra arguments.

A second difficulty arises when cloning the target of an `invokeinterface` bytecode. To keep the program legal, we must add the prototype of the cloned method to the interface and ensure that all classes implementing this interface define this method.

The information provided by the analysis phase can also guide other optimizations. For example, `s-local` objects can be allocated on the stack instead of in the heap. Our optimizer currently does not perform any optimizations other than eliminating synchronizations.

6. Evaluation

To evaluate the static and dynamic effectiveness of the analysis as well as the code growth and performance gain resulting from the transformation, we created a test environment and analyzed ten benchmark programs. The following two subsections describe our experimental framework and results.

6.1 Experimental Framework

Our testbed relies on OSUIF, a compiler system that allows simplified manipulation of object-oriented language constructs [11]. By feeding a test program to the Java-to-SUIF compiler `j2s` [21], we can work on an OSUIF representation of this program. While pulling in the transitive closure of all classes statically reachable from the main method, `j2s` converts Java `class` files into an OSUIF representation. Each OSUIF instruction is a list of expression trees, which makes it easier to analyze than the stack-based bytecodes of Java `class` files.

Figure 2 illustrates the overall optimization process. Instead of directly transforming the program’s OSUIF representation into optimized form and using a backend to emit native code, we choose to transform the original Java `class` files. The transformation program is written in Java and uses `JavaClass` [18], a library to parse and to manipulate Java `class` files. It reads the results of the analysis as well as the original program, adds the necessary nonsynchronized versions of optimizable classes, and writes out the optimized `class` files. This approach allows us to run optimized programs on any commercial Java virtual machine and

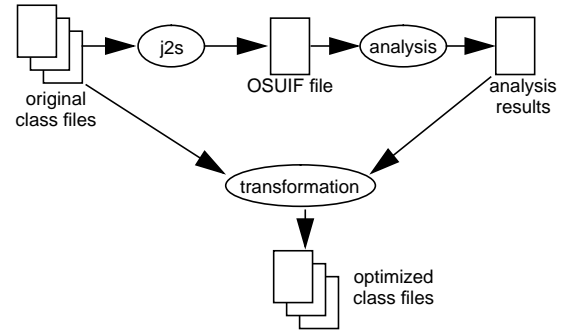


Figure 2. Overview of synchronization elimination

thus to measure directly the impact of our optimizations in a realistic setting.

6.2 Results

We tested our optimization on the benchmarks listed in Table 4. For all measurements except run-time performance we used the JDK 1.2 beta2 JVM, because the `j2s` front end does not yet work with the JDK 1.2. For all run-time measurements, we used the First Customer Ship (FCS) version of the Solaris JDK 1.2 Production Release, a high-performance Java implementation with an efficient implementation of synchronized calls [1]. We did not use any special command-line flags (such as flags to set the heap size) for the runs. All timing measurements were taken on an otherwise idle 400 MHz dual-processor Sun Ultra Enterprise 450 with 1GB of RAM.

Part of our benchmark suite consists of the SPECjvm98 benchmarks [30], run individually from the command line with the `-s100` option. In addition, we used *JavaCUP* [19], processing a grammar for Java 1.1, and *JLex* [20], reading as input `sample.lex`. The *SortingBenchmarks* program tests array manipulations in the JGL library [17].

For each program, Table 4 shows its size in bytecodes¹ (including bytecodes in the standard libraries), the number of methods, and the number of synchronized methods. Since our analysis ignores uninstantiated classes and unreachable methods, the data reflect the streamlined versions of both application and library code. Each program contains tens of thousands of bytecode instructions and thousands of methods; on average, 6% of the methods are synchronized.

For a static evaluation of the analysis, we determined the percentage of candidate alias sets that were optimized (Figure 3); an ideal analysis (without loss of precision) would reach 100% for a single-threaded application. Each unique alias set is a candidate for optimization if it includes a variable that references a newly created synchronized object and if it does not contain any formal parameters or their fields. This latter restriction ensures that we only consider an object once. For the programs in our benchmark suite, the analysis optimizes between 91% and 96% of candidate alias sets. The remaining candidate sets represent objects that either truly escape to the heap or that appear to escape because of imprecisions in the analysis (e.g., the limit on nesting to a depth of one). We were unable to determine how many candidates could be recognized by a more precise analysis; fortunately, the current

¹ Note that we present the number of bytecode instructions, not the code size in bytes.

Benchmark	Description	Bytecode instructions	Methods	Synchronized methods
compress	A file compression tool that comes with the SPECjvm98 benchmark suite.	64,296	2,554	169
db	A database application that comes with the SPECjvm98 benchmark suite.	64,380	2,564	170
jack	A Java parser generator that comes with the SPECjvm98 benchmark suite.	78,259	2,817	170
javac	The jdk1.0.2 Java compiler that comes with the SPECjvm98 benchmark suite.	100,600	3,753	196
JavaCUP	A Java parser generator.	42,345	1,513	69
jess	An expert system shell that comes with the SPECjvm98 benchmark suite.	78,067	3,062	190
JLex	A Java lexical analyzer generator.	40,517	1,270	71
mpegaudio	An audio file decompression tool that comes with the SPECjvm98 benchmark suite.	74,290	2,776	169
mtrt	A two-threaded raytracer that comes with the SPECjvm98 benchmark suite.	69,177	2,700	172
SortingBenchmarks	A benchmark that tests array manipulations in the JGL library.	30,843	1,264	79

Table 4. Java benchmarks used

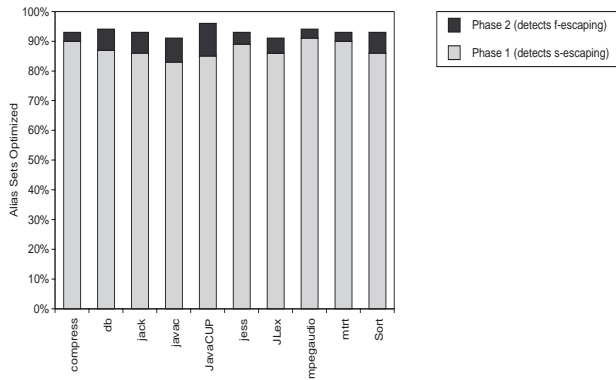


Figure 3. Percentage of candidate alias sets optimized

analysis already performs very well. Most candidates are recognized during phase one and are therefore s-local (never stored into any object field). The remaining candidates (typically 5-10%) are recognized during the analysis' second phase and are therefore nested one level within s-local objects.

To exploit this information, the transformation step clones classes and methods, as explained in the previous section. Each optimizable alias set leads to a transformation, but the overall code growth remains quite small (Table 5). On average, our transformation introduced 10 synchronized classes and cloned 18 methods per benchmark. Interestingly, all interface methods added belong to the `Enumeration` interface.

We also measured the impact of the optimization on the dynamic number of synchronizations. To obtain these numbers, we wrote a tool that inserts a counter increment at the beginning of every synchronized method, as well as code to print the counter on program exit. Figure 4 shows the results. Despite the seemingly small increment in precision provided by phase two of the analysis (recall Figure 3), this phase accounts for the majority of the synchronizations eliminated. The results reveal a bimodal

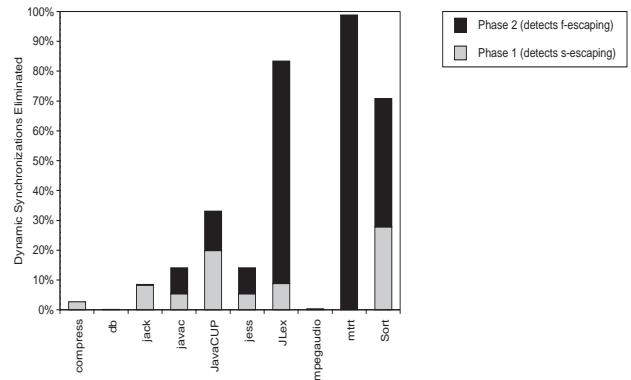


Figure 4. Dynamic number of synchronizations eliminated by optimization

distribution in optimization effectiveness: three of the benchmarks show a reduction in dynamic synchronizations of 70% or more, but six experience a reduction of less than 20%. In other words, synchronization elimination appears to either work very well, or not at all. Only *JavaCup* shows a reduction that lies between these two extremes.

Why does synchronization perform so poorly for some programs and so well for others, even though it eliminates over 90% of the optimization candidates for each program? On one hand, the analysis may optimize an alias set that contains a frequently accessed object. This is the case for *mtrt* in which the compiler is able to optimize the alias set for the `BufferedInputStream` object that represents 99% of all synchronizations. On the other hand, nearly all of a program's synchronizations may occur on objects represented by f-escaping alias sets, as is the case for *db*. In *db*, the `Vector` objects referenced by `Entry` objects account for essentially all synchronizations. None are optimizable, however, since the program places all `Entry` objects into a hash table, causing their vectors to be nested at a depth greater than one. The majority of the

Benchmark	Number of		
	Classes cloned	Methods cloned	Interface methods added
compress	10	21	8
db	10	23	8
jack	10	26	8
javac	15	23	10
JavaCUP	9	14	8
jess	12	19	4
JLex	8	9	8
mpegaudio	9	18	8
mtrt	10	18	8
SortingBenchmarks	8	9	7

Table 5. Code growth caused by optimization

Benchmark	Buffered-InputStream		ByteArray-OutputStream		Hashtable		Stack		String-Buffer		Vector		Other Classes		Total Synchroniz.	Exec. Time (sec.)
compress			12.4	0	19.9	0			35.2	3.9	0.7	0	31.8	45.2	3,906	39.7
db									0	0.6	99.9	0	0	62.7	48,111,356	54.3
jack					28.8	0			8.4	99.6	59.8	0.4	3.0	0	9,742,430	24.5
javac	50.7	3.4	24.1	27.6	18.2	0.1			6.0	89.8	0.7	12.1	0.2	6.6	16,284,407	27.5
JavaCUP	4.2	0			39.8	17.1	6.6	20.0	19.7	100	29.7	0			521,962	2.7
jess			0.1	0	73.2	0	0.3	36.6	0.3	37.1	25.8	0	0.3	0	4,801,127	14.4
JLex					0.3	0	8.8	100	0	100	89.3	89.5	1.6	0	1,834,569	0.6
mpegaudio			5.1	0	18.3	0			17.2	2.1	0.1	0	59.3	0	4,121	36.8
mtrt	98.8	100	0.2	0	0.1	0			0.4	1.2			0.5	0	703,266	9.0
SortingBenchmarks									27.8	100			72.2	59.4	9,942,128	5.7

Table 6. Percentage breakdown of important synchronizations executed at run-time

optimized alias sets in *db* either pertain to unexecuted code or infrequently executed code, such as code in exception handlers.

To better understand the source of the synchronizations, we collected additional data to identify six commonly synchronized classes (Table 6). For each class column, the left half shows the percentage of synchronized instance methods invoked on that class with respect to the total number of synchronizations, while the right column shows the percentage of synchronizations eliminated within that class. For example, in *JavaCUP*, class *StringBuffer* represents 19.7% of all synchronizations during the benchmark run, and the optimization eliminates 100% of these synchronizations. Blank entries signify that the corresponding class was not used in a benchmark. Finally, the original (unoptimized) program’s execution time and total number of executed synchronizations appear in the two rightmost columns of Table 6.

Despite having fewer than 200 synchronized methods, the majority of the benchmarks execute a large number of synchronized invocations, with *db* topping the charts with over 48 million synchronizations. *Compress* and *mpegaudio*, on the other hand, execute practically no synchronizations. Clearly, the performance benefit of synchronization elimination will be greatest for programs that perform frequent synchronizations.

To measure the performance impact, we timed all programs ten times and took the fastest run. For the SPECjvm98 programs we used the self-reported time of the third iteration (forced with *-m3* and *-M3*). For the other benchmarks we used the smallest sum of user and system time as measured by the Solaris *ptime* command. Since *JLex* executed only for a very short time, we measured a benchmark harness that repeated it 100 times and then divided by 100. We were unable to similarly treat *JavaCUP*, which runs for only 2.7 seconds, because it apparently does not reinitialize its global variables.

To correlate the execution time measurements with the reductions in the number of synchronizations, we also computed an estimated speedup by multiplying the number of synchronizations eliminated by the cost of an “average” synchronization (66 cycles, or 0.165 microseconds). By running a small microbenchmark, we measured the average cost of uncontended synchronization to be between 57 and 76 cycles. The cost apparently depends on the surrounding code, such as the instructions in the called methods, which affects processor instruction scheduling.

Figure 5 shows both measured and estimated speedups. As expected, programs with a high frequency of synchronizations benefit most from the optimization. *JLex* demonstrates the highest speedup (36.4%) because it executes 1.8 million synchronized invocations in 0.6 seconds, of which approximately 83% are optimized away. *SortingBenchmarks*, too, exhibits a large speedup

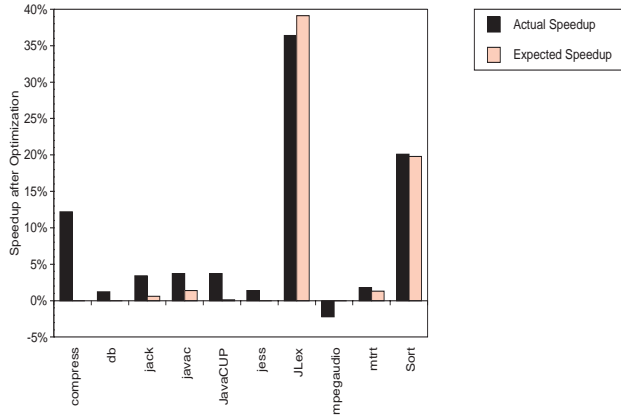


Figure 5. Actual vs. predicted speedups of optimized programs

(20.1%), with 71% of its nearly 10 million synchronizations eliminated. The remaining applications appear unaffected by the optimization; even though optimization removes nearly all synchronizations from *mtrt*, the original program has too few synchronizations to matter in its nine-second run. The two programs with significant speedups demonstrate gains close to their predicted values. Most other programs show a larger speedup than expected, indicating secondary optimization effects. For example, the removal of synchronization may allow some methods to be inlined, or the cloning of code may make call sites less polymorphic and reduce the cost of method calls.

As the Java specification mandates, a JVM must flush a thread's working memory to main memory at lock accesses. By completely eliminating synchronization operations, our transformation causes an optimized program to execute fewer flushes of memory, thereby potentially violating Java semantics. In order to retain the behavior of the original program, it would be necessary to replace some of the eliminated points of synchronization with memory barrier instructions, possibly reducing the performance benefits of optimization. Nonetheless, when run on uniprocessor machines, our optimization strictly conforms to Java semantics since the above issues only apply to multiprocessor machines. Speedups on multiprocessors will depend on the particular machine since the cost of memory barrier instructions (as well as the cost of synchronization) varies greatly among multiprocessors.

A more aggressive analysis could potentially improve performance even more. Our analysis fails to eliminate all synchronizations because it ignores synchronized static methods as well as synchronized blocks. Also, it refuses to optimize an object reachable by more than one object dereference; a more aggressive shape analysis may eliminate this restriction. Finally, since our analysis is a whole-program analysis, it is uncertain how it can be used in the face of dynamic loading of unknown classes.

7. Related Work

The most relevant related work is in reducing synchronization and statically analyzing heap objects. In general, by monitoring heap assignments, parameter passing, and return values, our analysis conservatively calculates the lifetime of objects; in this sense, our analysis resembles lifetime and escape analysis [27][5], studied primarily for functional languages. Similarly, by maintaining one level of object dereferences, our analysis resembles a limited shape analysis [29][14]. The escape analysis of Choi *et al.* [6] and the lock analysis of Aldrich *et al.* [2], both developed concurrently

with our work, are the most similar to ours. As far as we know, they are the only other projects that perform a general static analysis and use the results to eliminate unnecessary synchronization. Other work reduces synchronization by other means [3][4][8][12][23][26] or analyzes heap objects for other purposes [7][9][13][28].

Choi *et al.* at IBM [6] recognize that two threads may not share a Java object whose lifetime is bounded by a method. At every program point they perform an escape analysis that is very similar to but both more and less conservative than our analysis. The increase in accuracy derives mainly from its flow-sensitivity and arbitrary object nesting, whereas the decrease in accuracy stems from their interprocedural treatment of allocation sites. They represent a new statement as a single object, which their analysis deems as thread-local or not, regardless of the calling context. In doing so, the transformation simplifies immensely for they only need to modify allocation sites. At the time of lock access the run-time system checks a bit in the object's header to determine its thread reachability. Our transformation neither imposes this additional run-time check nor requires modification of the run-time system but may increase code size. Despite these differences, their results corroborate our findings that one can improve a program's execution time by unsynchronizing its thread-local objects.

Another system, created by Aldrich *et al.* [2] at the University of Washington, determines statically when a class will not have an instance reachable from a static variable. In such a case, the class occurs only in a single-threaded setting and does not require any specialized methods. Our analysis not only distinguishes allocation sites but also distinguishes execution paths; hence it can be more precise. Despite this difference, their analysis, although presented in a curried notation and capable of detecting other synchronization optimizations, resembles ours.

Previous work addresses the overhead of Java synchronization by finding more efficient ways to implement a locking operation. Bacon *et al.* [4] lessen the burden by speeding up the common synchronization cases. Compared to an earlier implementation, Bacon demonstrates a median speedup of 22% for single-threaded programs. Similarly, Krall and Probst [23] reduce the locking overhead in CACAO. The efficient implementation of lock accesses applies to all synchronizations (*i.e.*, to both synchronized methods and synchronized blocks), whereas our approach deals only with synchronized instance methods, and is orthogonal to our work.

Some authors have proposed eliminating synchronization in single-threaded programs by determining that no second thread is constructed [26][3][12]. The idea is that after scanning an entire program and not finding any calls to `Thread` constructors, a compiler can conclude that the program is single-threaded and remove *all* synchronizations. Similarly, a run-time system can omit synchronization until the second thread is created. Unfortunately, these approaches will not succeed in general. In particular, they do not improve multi-threaded programs, such as GUI-based applications. The fact that the JDK 1.2 system classes spawn several helper threads at the start of every application, making all programs multi-threaded, merely aggravates the situation.

Diniz and Rinard strive to reduce statically the amount of synchronizations with *lock coarsening* in a C++ parallelizing compiler [8]. Although their fundamental goal is the same as ours,

their approach is significantly different. We focus on thread-local objects, while they work on global objects. Consequently, we unsynchronize objects by removing unnecessary synchronization constructs, whereas they unsynchronize objects by repositioning necessary constructs to coarser levels of granularity. One way they accomplish this is by moving synchronization constructs outside a region of code that repeatedly acquires and releases a lock (e.g. the body of a loop). Another way is by allowing an object to share a lock with other objects—an infeasible task in Java since, by definition, each object owns its own lock. Despite this difference, both optimizations perform a similar transformation: once the compiler designates a region of code as synchronization-free, it removes all synchronization constructs in this region, cloning methods when necessary.

Corbett employs a shape analysis in order to shrink finite-state models used to prove certain properties of concurrent Java programs [7]. By being flow-sensitive and by summarizing objects spanning loop iterations, his analysis is less conservative than ours; however, he simplifies the interprocedural problem by inlining all called methods. As a result, a single allocation site wholly identifies an object's static context, but only statically bounded recursive programs work.

Gay and Steensgaard use an interprocedural analysis to determine when it is safe for the Marmot compiler to allocate a heap object on the stack [13]. An object whose reference never appears in the heap exhibits a known, limited lifetime and thus can be stack allocated; synchronizations on such stackable objects can safely be eliminated, although Marmot does not currently do so. Our analysis optimizes a larger set of objects since it allows a level of object nesting. Furthermore, Marmot's analysis considers the propagation of a freshly created object only for variables that are not aliased, whereas our analysis does not impose this limitation.

Dolby [9][10] uses a more extensive yet similar analysis to tag inlinable objects in C++ programs. Instead of examining when an object escapes its owning thread and hence is not thread-local, he uses an escape analysis to determine when an object escapes its owning object. In principle, this analysis can be extended to target synchronizations in Java programs.

8. Conclusions

To counter the current high cost of synchronization in Java, we have developed a global compile-time optimization that removes unnecessary synchronizations based on a simple idea: objects reachable from only a single thread do not need synchronization. A global, context-sensitive analysis first identifies objects that will be local to a thread, before a transformation program modifies allocation sites in the original class files to avoid synchronization on these objects.

The analysis is effective in detecting thread-local objects. For all programs in our benchmark suite, it optimizes over 90% of the candidate alias sets. These optimized alias sets account for a greater than 70% reduction in dynamic synchronizations for three of these programs. For programs where synchronization is frequent, these reductions translate into substantial speedups. Two programs, *JLex* and *SortingBenchmarks*, show speedups of 36% and 20%, respectively, as a result of synchronization elimination.

Our results demonstrate that in some realistic Java programs, a compiler can omit a substantial fraction of synchronizations without the programmer's help. Having proven the potential for

this optimization, we plan to investigate ways that would allow dynamically-compiled Java implementations to perform similar optimizations and eliminate the dependence on the whole program.

Acknowledgments

This work was funded in part by Sun Microsystems, the State of California MICRO program, and the National Science Foundation under CAREER grant CCR96-24458. The SUIF/OSUIF compiler infrastructure is funded by DARPA grant PR-9836. We also appreciate the helpful feedback from the anonymous reviewers.

References

- [1] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, 1-5 November 1999.
- [2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of the Sixth International Static Analysis Symposium*, Venezia, Italy, September 1999.
- [3] David F. Bacon. *Fast and Efficient Optimization of Statically Typed Object-Oriented Languages*. Ph.D. thesis, University of California, Berkeley, October 1997.
- [4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 258-268, Montreal, Canada, May 1998.
- [5] Bruno Blanchet. Escape analysis: Correctness Proof, Implementation and Experimental Results. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 25-37, San Diego, California, 19-21 January 1998.
- [6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, 1-5 November 1999.
- [7] James C. Corbett. *Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs*. Technical Report ICS-TR-98-20, Department of Information and Computer Science, University of Hawaii, October 14, 1998.
- [8] Pedro Diniz and Martin Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. In *Journal of Parallel and Distributed Computing* 49(2), March 1998.
- [9] Julian Dolby. Automatic Inline Allocation of Objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*, pages 7-17, Las Vegas, Nevada, June 1997.
- [10] Julian Dolby and Andrew A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 1-20, 1998.
- [11] Andrew Duncan, Bogdan Cocosel, Costin Iancu, Holger Kienle, Radu Rugina, Urs Hölzle, and Martin Rinard. *OSUIF: SUIF 2.0 With Objects*. Overview paper from the SUIF Workshop at Stanford University, August 1997.

- [12] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. *Marmot: an Optimizing Compiler for Java*. Microsoft Technical Report, November 1998.
- [13] David Gay and Bjarne Steensgaard. *Stack Allocating Objects In Java*. Microsoft Technical Report, November 1998.
- [14] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 1-15, St. Petersburg Beach, Florida, 21-24 January 1996.
- [15] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley: Berkeley, California, 1996.
- [16] Allan Heydon and Marc Najork. Performance Limitations of the Java Core Libraries. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 35-41, June 1999.
- [17] Java Generic Library (JGL). <http://www.objectspace.com/products/jgl>.
- [18] JavaClass parsing library. <http://www.inf.fu-berlin.de/~dahm/JavaClass>.
- [19] JavaCUP parser generator. <http://www.cs.princeton.edu/~appel/modern/java/CUP>.
- [20] JLex lexical analyzer generator. <http://www.cs.princeton.edu/~appel/modern/java/JLex>.
- [21] Holger Kienle and Urs Hölzle. j2s: A SUIF Java compiler. In *Proceedings of the Second SUIF Compiler Workshop*, pages 8-15, August 1997. Also available as Technical Report TRCS98-18, Department of Computer Science, University of California, Santa Barbara.
- [22] Andreas Krall. Efficient Java VM Just-in-Time Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 12-18, Paris, France, October 1998.
- [23] Andreas Krall and Mark Probst. Monitors and Exceptions: How to implement Java efficiently. In S. Hassanzadeh and K. Schauer, editors, *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15-24, Palo Alto, March 1998. ACM.
- [24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley: Berkeley, California, 1997.
- [25] C. E. McDowell. Reducing garbage in Java. In *ACM SIGPLAN Notices* 33(9), September 1998.
- [26] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Conzel. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 1-20, Berkeley, California, June 1997.
- [27] Cristina Ruggieri and Thomas P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285-293, San Diego, California, 13-15 January 1988.
- [28] Radu Rugina and Martin Rinard. Pointer Analysis for Multi-threaded Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 77-90, Atlanta, GA, May, 1999.
- [29] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*, pages 16-31, St. Petersburg Beach, Florida, 21-24 January 1996.
- [30] SPEC Java virtual machine benchmark suite. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0*. August 1998. <http://www.spec.org/osg/jvm98/jvm98/doc/index.html>.
- [31] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, pages 1-12, La Jolla, California, 18-21 June 1995.