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
http://www.cs.ucsb.edu/oocsb

Technical Report TRCS99-10
April 2, 1999

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 a flow-insensitive, context-sensitive data-flow analysis that finds such situations and a global optimizing transformation that 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%, respectively.

1. Introduction

Java provides synchronization constructs to allow multiple threads to access shared data structures safely. The standard 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. This incurs significant synchronization overhead, prompting researchers to focus on efficient implementations of the Java synchronization primitives [B+98]. In contrast to this work, we focus on the complementary goal of completely eliminating synchronization operations where possible, thereby reducing the overhead to zero for these situations.

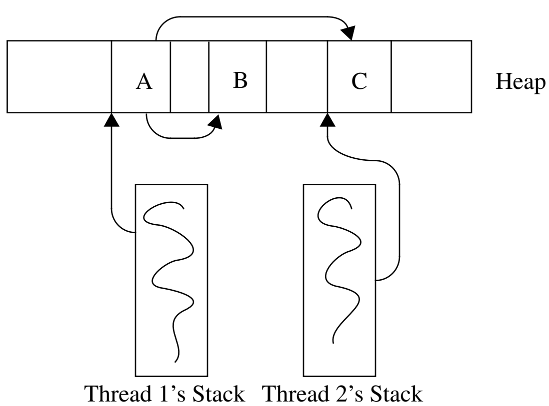

Figure 1. A multi-threaded Java program.

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.

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 thus share the same code. How can we avoid synchronizing in the latter cases 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 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).

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 this 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 [GJS96][LY97].

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).

While desirable from a software engineering point of view, locking 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 [KP98]. 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 [F+98], 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. The next section provides an overview of an optimization targeted at removing as much of this overhead as possible.

3. Overview of Synchronization Elimination

Objects that can only be 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 a single thread, as observed above. In general, it is impossible to know statically the precise set of objects that will be accessed by one thread only—some objects may be reachable by more than one thread but may never actually be accessed by more than one thread.

With the help of a global data-flow analysis, a compiler can prove to some extent that certain objects will be thread-local. Our optimization is based on such an 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 of the ideas behind synchronization elimination. First, we 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, this would entail solving a difficult reachability problem (*e.g.*, recognizing all objects in a tree hanging off a local variable), a task that is beyond the capabilities of current static analyses.

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.

The analysis step operates on OSUIF intermediate files [D+97] produced by the Java-to-SUIF compiler j2s [KH98]. 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. Instead of directly transforming the program's OSUIF representation into optimized form and using a backend to emit native code,

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

we choose to transform the original Java class files. This approach allows us to run optimized programs on any commercial Java virtual machine and thus to measure directly the impact of our optimizations in a realistic setting.

Figure 2 illustrates the overall optimization process. Instead of directly producing optimized code, the analysis outputs its results to a file. A subsequent transformation step reads these results and the original program, adds the necessary nonsynchronized versions of optimizable classes, and writes out the optimized

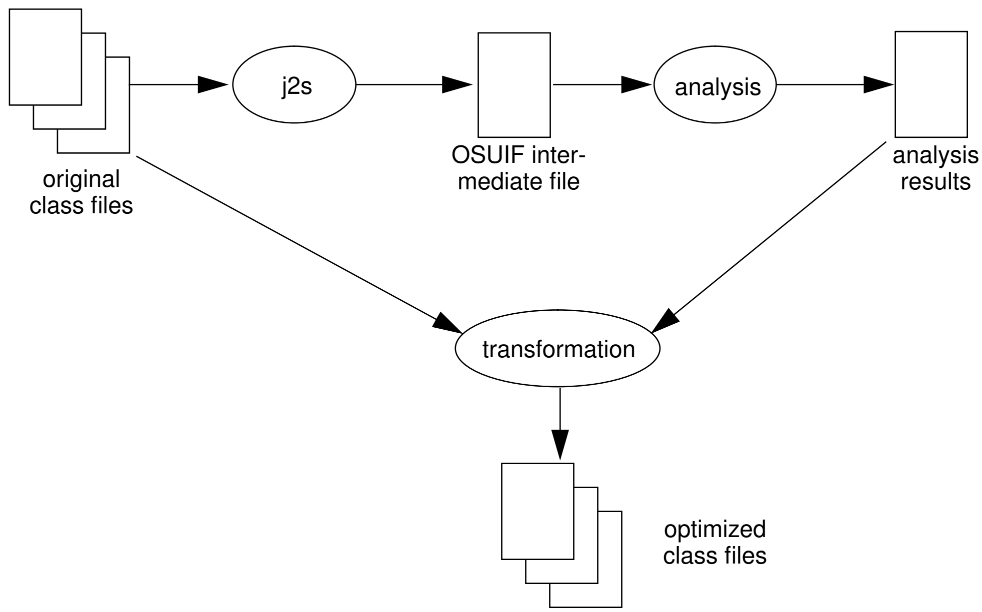

Figure 2. Overview of synchronization elimination.

class files. The transformation program is written in Java and uses JavaClass [JavaClass], a library to parse and to manipulate Java class files.

4. Analysis

Our system statically detects thread-local objects with a flow-insensitive, context-sensitive, 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).

We present the two stages as constraint problems. The analysis concerns itself only with instructions involving pointer variables, so we ignore all other operations. Furthermore, to simplify 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 = new T	object creation
foo(a_0,...,a_n)	method or static call[1]

Here x, y, a_0, ..., 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 Figure 3 the temporary variables t_0 and t_1 capture the stack assignments of the aload and getfield instructions. We refer the reader to [KH98] or [F+98] for more information regarding similar conversions.

aload x getfield f astore y	$\longrightarrow$	t_0 = x t_1 = t_0.f y = t_1

Figure 3. Example transformation of bytecodes into canonical form

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, methods do not return an object but merely store it into its *return* variable. Therefore, letting $return_m$ denote the *return* variable of method *m*, we transform the statement return x in *m* to return_m = 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 return_put). 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. Java's method invocation rules (sec. 15.11 in [GJS96]) and the program's complete class hierarchy determine the elements of *methods-invoked(m)*.

The analysis is based on the notion of *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*. 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 only 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 all local variables *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.

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

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 [WL95] 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 ≡ 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 $AS(y) \supseteq AS(x)$.

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) ≡ AS(y) s-escape(x) ≡ s-escape(y)
x.f = y (instance) y = x.f (instance)	The program stores or loads y from the heap, so y is s-escaping. s-escape(y) ⊇ {T}
x.f = y (class) y = x.f (class)	The program stores or loads y from the heap, so y is s-escaping. s-escape(y) ⊇ {T}

foo(a_0,...,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.
	
	∀ g ∈ methods-invoked(foo)
	  ∀ i ∈ [0...n]
	    s-escape(a_i) ⊇ s-escape(p_i)
	  ∀ i such that connected(p_i,$return_{foo}$)
	    AS(a_i) $\cong_1$ AS(a_n)

Table 1. Constraints for Phase 1.

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. Note that, 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 control flow merging.

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(a_0,...,a_n), 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 specially. For the native methods in the core library classes (*e.g.*, Object.equals) the analysis hand-constructs 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-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.

x = y	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) $\cong_2$ AS(y) denote the following recursive constraints: AS(x) $\equiv$ AS(y) f-escape(x) $\equiv$ f-escape(y) if s-escape(x) $\cup$ s-escape(y) = $\emptyset$: $\forall$ f $\in$ fields(x,y) AS(x.f) $\cong_2$ AS(y.f)
x.f = y (instance) y = x.f (instance)	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 FieldAccess(x,f,y). if s-escape(x) = $\emptyset$: AS(x.f) $\cong_2$ AS(y) otherwise: f-escape(y) $\supseteq$ {T}
x.f = y (class) y = x.f (class)	Variable y is f-escaping, because it is stored into or loaded from a variable reachable by more than one thread. f-escape(y) $\supseteq$ {T}

foo(a_0,...,a_n)	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 we did 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.
	
	$\forall$ g $\in$ methods-invoked(foo)
	
	$\forall$ i $\in$ [0...n]
	f-escape(a_i) $\supseteq$ f-escape(p_i)
	if s-escape(a_i) = $\emptyset$:
	$\forall$ f $\in$ fields(p_i)
	f-escape(a_i.f) $\supseteq$ f-escape(p_i.f)
	
	$\forall$ i such that connected(p_i,$return_{foo}$)
	AS(a_i) $\cong_2$ AS(a_n)
	
	$\forall$ f,p_i,p_j such that connected(p_i.f,p_j)
	FieldAccess(a_i,f,a_j)
	
	$\forall$ f,h,p_i,p_j such that connected(p_i.f,p_j.h)
	if s-escape(a_i) $\cup$ s-escape(a_j) = $\emptyset$:
	AS(a_i.f) $\cong_2$ AS(a_j.h)
	otherwise if s-escape(a_i) = $\emptyset$:
	f-escape(a_i.f) $\supseteq$ {T}

Table 2. Constraints for Phase 2.

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, from them. 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. Note that if the alias sets are equated, their fields will not be equated 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 = x.f and x.f = y naturally cause y to become f-escaping, since it references an object 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 impose that constraint on 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. The next section describes how the optimization pass exploits the results of this analysis.

5. Transformation

The non-f-escaping alias sets that the analysis identifies suggest objects that we can optimize. Before describing our optimizing transformation, we explain two terms. A *bounding method* identifies a lower bound on the stack in which an object reference can appear. It only pertains to a non-f-escaping object and denotes the method that uses the object but does not let it leave (either by returning it or by making it accessible from a formal parameter) once the method finishes. In terms of our constraints this means that no local variable referencing the object is aliased with a formal parameter or with a field of a formal parameter. We can then say that an object is *optimizable* if it has a bounding method and is synchronized.

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, but each has its advantages and disadvantages. We chose a transformation 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 a simplified Vector class and its unsynchronized version:

```
class Vector {
    public synchronized void add(...) { }
}
class Vector$Unsync extends Vector {
    public Vector$Unsync() { super(); }
    public void add(...) { /* 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.

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. Specifically, the statement new Vector becomes new Vector$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.

```
v1 = createVector()                 // v1 is f-escaping
v2 = createVector()                 // v2 is optimizable
v2 = createVector$Clone1()          // modified call site
v1.add(...);                        // synchronized invocation
v2.add(...);                        // unsynchronized invocation

Vector createVector() {             // original method
    return new Vector();
}
Vector createVector$Clone1 {        // cloned method
    return new Vector$Unsync();
}
```

Since v2 references an optimizable object, it receives an instance of Vector$Unsync from the cloned class createVector$Clone1.

To see why this transformation reduces the number of synchronizations, consider the lines v1.add(...) and v2.add(...). 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 synchronized add method in class Vector, while the second triggers the unsynchronized version in Vector$Unsync. Hence the optimized code performs fewer synchronizations. Note that no changes need to occur to these call sites or to any code to which optimized objects are passed.

To carry out this transformation, we must identify both the allocation sites that should construct an unsynchronized object and the call chain from the bounding method to the allocation site. This can easily 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 for 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 that method.

The information provided by the analysis phase can also guide other optimizations. For example, s-local objects can be stack allocated. 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 tested our optimization on the benchmarks listed in Table 3. 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 FCS version. For all run-time measurements, we used the FCS version

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 3. Java benchmarks used.

of the Solaris JDK 1.2 Production Release, a high-performance Java implementation with an efficient implementation of synchronized calls. 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 [SPEC98], run individually from the command line with the -s100 option. In addition, we used *JavaCUP* [JavaCUP], processing a grammar for Java 1.1, and *JLex* [JLex], reading as input sample.lex. The *SortingBenchmarks* program tests array manipulations in the JGL library [JGL].

For each program, Table 3 shows its size in bytecodes[1] (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 4); 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 newly created synchronized object and if it does not contain any parameters or fields of parameters. This latter restriction ensures that we only consider an object in its bounding method. For the programs in our benchmark suite, the analysis opti-

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

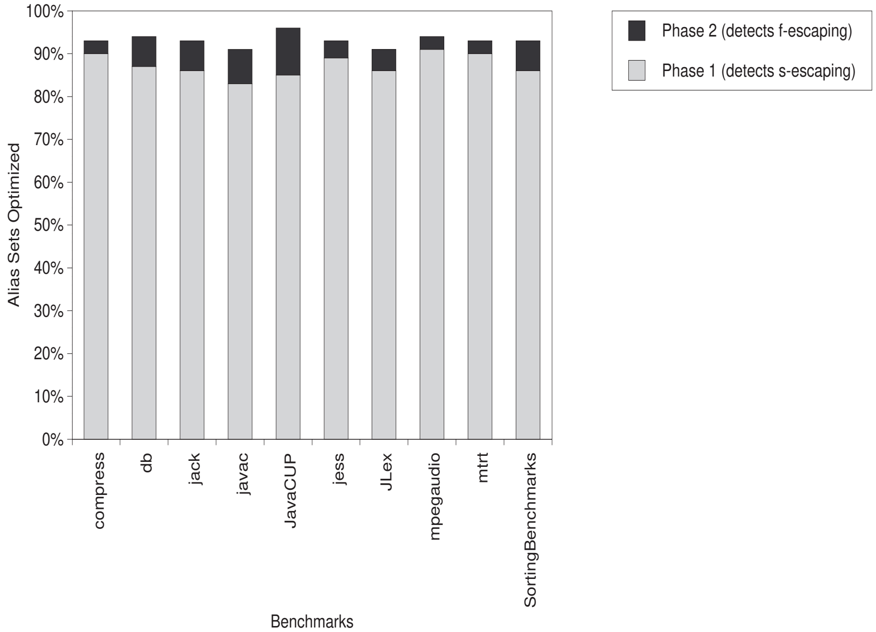

Figure 4. Percentage of candidate alias sets optimized.

mizes 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 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 4). On average, the transformation introduced 10 synchronized classes and cloned 18 methods per benchmark. Interestingly, all interface methods added belong to the Enumeration interface.

Benchmark	Cloned classes	Cloned methods	Added interface methods
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 4. Code growth due to eliminating synchronizations.

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 5 shows the results. Despite the seemingly small increment in precision provided by phase two of the analysis (recall Figure 4), that phase turns out to account for the majority of the synchronizations eliminated. The results reveal a bimodal distribution

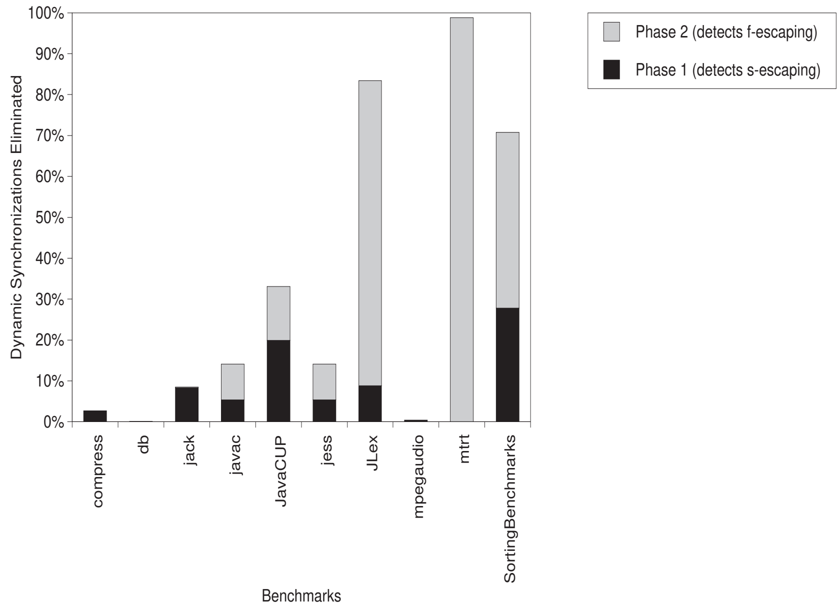

Figure 5. Dynamic number of synchronizations eliminated by optimization.

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 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 5). 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 rightmost column of Table 5 shows the total number of synchronizations executed by the original (unoptimized) programs.

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 perfor-

Benchmark	Buffered-InpStream		ByteArray-OutpStrm		Hashtable		Stack		String-Buffer		Vector		Other Classes		Total Synchroniz.
compress			12.4	0	19.9	0			35.2	3.9	0.7	0	31.8	45.2	3,906
db									0	0.6	99.9	0	0	62.7	48,111,356
jack					28.8	0			8.4	99.6	59.8	0.4	3.0	0	9,742,430
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
JavaCUP	4.2	0			39.8	17.1	6.6	20.0	19.7	100	29.7	0			521,962
jess			0.1	0	73.2	0	0.3	36.6	0.3	37.1	25.8	0	0.3	0	4,801,127
JLex					0.3	0	8.8	100	0	100	89.3	89.5	1.6	0	1,834,569
mpegaudio			5.1	0	18.3	0			17.2	2.1	0.1	0	59.3	0	4,121
mtrt	98.8	100	0.2	0	0.1	0			0.4	1.2			0.5	0	703,266
SortingBenchmarks									27.8	100			72.2	59.4	9,942,128

Table 5. Breakdown of important synchronizations executed at run-time.

mance benefit of synchronization elimination will be greatest for programs that perform frequent synchronizations.

To measure the performance impact, we ran all programs ten times and used the fastest run as determined by the smallest sum of user and system time as measured by the Solaris ptime command. For the SPECjvm98 programs we used the self-reported time of the third iteration (forced with -m3 and -M3). Since *JLex* ran only for a very short time, we first measured a benchmark harness that repeated it 100 times and then calculated its execution time by dividing by 100. We were unable to similarly treat *JavaCUP*, which runs for only 2.5 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 6 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 (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 9.0-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.

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; it also refuses to optimize an object reachable by more than one object dereference. The latter point may be partially tamed with a more aggressive shape analysis. Finally, since our analysis is a whole-program analysis, it is uncertain how it could be used in the face of dynamic loading of unknown classes.

7. Related Work

Previous work addresses the overhead of Java synchronization by finding more efficient ways to implement a locking operation. Bacon *et al.* [B+98] lessen the burden by speeding up the common synchronization cases. Compared to an earlier implementation, Bacon demonstrates a median speedup for single-threaded

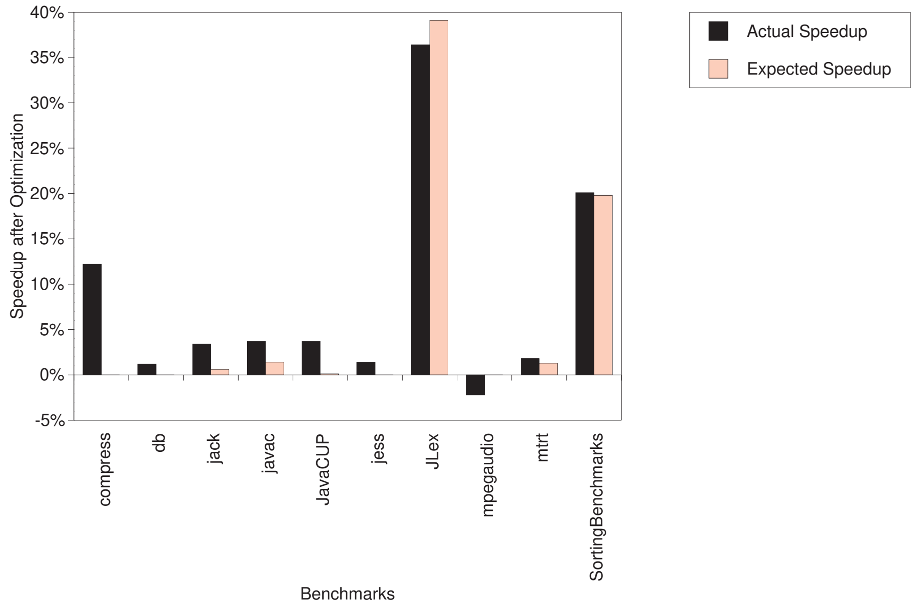

Figure 6. Actual and predicted speedups of optimized programs.

programs of 22%. Similarly, Krall and Probst [KP98] 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.

Some authors have proposed eliminating synchronization in single-threaded programs by detecting that no second thread is constructed [M+97][B97][F+98]. 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 could omit synchronization until the second thread is created. Unfortunately, this approach will not succeed in general. In particular, it does not improve multi-threaded programs, such as GUI-based applications. Also, the JDK 1.2 system classes spawn helper threads at the start of every application, making all programs multi-threaded.

Diniz and Rinard strive to reduce statically the amount of synchronizations with *lock coarsening* in a C++ parallelizing compiler [DR98]. 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 bytecode. 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.

By monitoring heap assignments, parameter passing, and return values, our analysis conservatively calculates the lifetime of objects; in this sense, our analysis resembles lifetime analysis [RM88][Bla98], studied primarily for functional languages. Similarly, by maintaining one level of object dereferences, our analysis resembles a limited shape analysis [SRW96][GH96].

Gay and Steensgaard use an interprocedural analysis quite similar to ours to determine when it is safe for the Marmot compiler to allocate a heap object on the stack [GS98]. 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 could 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 [D97][DC98] uses a more extensive yet similar analysis to tag inlinable objects in C++ programs. In principle, this analysis could 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.

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.

References

[B97] David F. Bacon. *Fast and Efficient Optimization of Statically Typed Object-Oriented Languages*. Ph.D. thesis, University of California, Berkeley, October 1997.

[B+98] 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.

[Bla98] 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.

[DR98] 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.

[D97] 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.

[DC98] 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.

[F+98] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. *Marmot: an Optimizing Compiler for Java*. Technical Report. http://www.research.microsoft.com/apl.

[GH96] 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.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley: Berkeley, California, 1996.

[GS98] David Gay and Bjarne Steensgaard. *Stack Allocating Objects In Java*. Technical Report. http://www.research.microsoft.com/apl.

[JavaClass]JavaClass parsing library. http://www.inf.fu-berlin.de/~dahm/JavaClass.

[JavaCUP]JavaCUP. http://www.cs.princeton.edu/~appel/modern/java/CUP.

[JGL] Java Generic Library (jgl). http://www.objectspace.com/products/jgl.

[JLex] JLex. http://www.cs.princeton.edu/~appel/modern/java/JLex.

[KH98] 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.

[K98] 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.

[KP98] Andreas Krall and Mark Probst. Monitors and Exceptions: How to implement Java efficiently. In S. Hassanzadeh and K. Schauser, editors, *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15-24, Palo Alto, March 1998. ACM.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley: Berkeley, California, 1997.

[M98] C. E. McDowell. Reducing garbage in Java. In *ACM SIGPLAN Notices* 33(9), September 1998.

[M+97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. 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.

[D+97] 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.

[RM88] 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.

[SPEC98] SPEC Java virtual machine benchmark suite. http://www.spec.org/osg/jvm98.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, pages 16-31, St. Petersburg Beach, Florida, 21-24 January 1996.

[WL95] 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.