RC22552 (W0209-004) September 3, 2002
Computer Science

IBM Research Report

Detecting Unwanted Synchronization in Java Programs

George Leeman, Aaron Kershenbaum, Larry Koved, Darrell Reimer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Detecting Unwanted Synchronization in Java Programs

George Leeman, Aaron Kershenbaum, Larry Koved, Darrell Reimer
IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598
(914)784-7702
{ gbl, aaronk, koved, dreimer }@us.ibm.com

ABSTRACT

A program's performance can be significantly improved by removing unwanted synchronization that causes time consuming tasks to run serially instead of in parallel. In large programs, especially those using libraries, it is usually difficult to manually detect unwanted synchronization. We describe an approach for automatically detecting unwanted synchronization in Java programs, including a detailed algorithm for computing the monitors involved in the synchronization. Our approach is highly scalable and is thus applicable to programs of realistic size. We have implemented this approach and tested it on several real problems, some of which are large. We present computational experience on both small and large examples, demonstrating that unwanted synchronization exists in practice, and that significant performance improvements are obtainable when unwanted synchronization is removed.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Performance, Languages

Keywords

Invocation graph, Java, locking, monitors, static analysis, synchronization

1. INTRODUCTION

Synchronization is necessary to eliminate race conditions arising when two or more threads of execution simultaneously access a shared resource. The simplest example is multiple threads making concurrent modification of a field's value. Synchronization forces these threads to execute serially. If the synchronization is necessary, this loss of efficiency is unavoidable.

In this paper we are concerned with improving a program's performance by analyzing its use of synchronization. We introduce three terms. If it can be proven that the removal of a specific synchronization will not change program behavior, we say that the synchronization is *unnecessary*; otherwise, the synchronization is *necessary*. For example, if eliminating a synchronization could cause a race condition, the synchronization is necessary. Next, we say that a synchronization is *unwanted* if it causes significant performance degradation. Thus there can be two types of unwanted synchronization: necessary and unwanted, or unnecessary and unwanted. In the first case, the only hope of eliminating the synchronization would be to restructure the code. In the latter case, the synchronization could be removed. Thus three issues should be considered:

1. Identify places in the program where synchronization may occur that can result in performance degradation or dangerous system failure (e.g. a remote call which could hang).
2. Identify whether the synchronization is necessary; if it is not, it may be removed.
3. If the synchronization is necessary, identify transformations of the code, if possible, to eliminate the synchronization.

The objective and primary contributions of this paper are in the first of these issues, identifying the synchronization of expensive or dangerous operations. We present empirical results demonstrating that unwanted synchronization has a significant detrimental impact on performance. Our analysis can be used in conjunction with other tools to distinguish necessary from unnecessary synchronization (issue 2). Furthermore, they may use the results of our analysis as a starting point. The results of our analysis also provide detailed execution sequence information which would help in determining how to change the code, if required (issue 3).

We are motivated by experiences with code deployed in very large web based applications that perform poorly or fail when unwanted synchronization is present. Examples include synchronized calls to database operations and LDAP servers. Symptoms of unwanted synchronization which we have observed include slow and erratic response times and throughput and application hangs. These problems in the application can even lead to entire site outages. The effects of unwanted synchronization surface most commonly under heavy workloads in production environments where minor problems with hardware, network or software components can trigger severe problems from unwanted synchronization. These problems are very hard to simulate during system test. Our experience is that the unwanted synchronization can be found with a set of sophisticated runtime analysis tools, but with much

greater difficulty. We have also observed that the skills required, and cost to diagnose and repair the problem are much higher once the code is deployed in a production environment. This paper describes an approach which will identify the occurrences of unwanted synchronization prior to deploying the code in a production environment, when the cost for modification is much lower.

Our algorithm is based on an augmented invocation graph [12]. The input to the algorithm is the code for a program, a list of relevant entry points (*head methods*), and a set of methods (*tail methods*) for which synchronization is unwanted. The algorithm determines whether any tail methods are in any execution paths from the head methods where they could be executing under the constraints of synchronization. We assume a closed world analysis. Furthermore, we do not require developers to add annotations to the program to guide the analysis.

The invocation graph we use represents intraprocedural and interprocedural analysis. It includes a control flow graph for each invoked method. The interprocedural graph contains edges (*A*, *B*), representing the invocation of method *B* from within method *A*. The edge is augmented with method *A*'s program counter for the call site to *B*.

After the invocation graph is constructed, we distinguish three types of nodes. A *head* is a node whose target method begins an execution path in which we are interested; it may be a root of the graph, or its method may initiate actions relevant to synchronization, such as the start of a new thread (e.g. `java.lang.Thread.run`). A *tail* is a node whose method execution is potentially time consuming (e.g.,. computationally expensive, system calls outside the Java runtime, or calls to remote resources), or may not return due to system or software failure. Therefore we don't want the tail method to be synchronized. One way to identify tail methods is to analyze execution traces of running programs. Typical examples are methods which create sockets, connect to databases, make Remote Method Invocation (RMI) calls, perform directory lookups, initiate expensive database queries, parse XML documents, or write to local files. The heads are automatically generated by our system, but the user may pick a subset of the system choices. The tails are provided as input to our algorithm. Our algorithm determines *source* nodes, the places where synchronization originates, and it identifies paths from the heads to the tails and passing through the sources. In Java the sources arise either via synchronized methods or synchronization blocks of code.

Once the synchronized nodes have been identified, further investigation is required to determine whether the synchronization is in fact unnecessary or if the program could be restructured (perhaps automatically) to make it unnecessary. For example, a race detection algorithm such as Choi et al. [7], [8] might indicate that races could occur, or the application might need synchronization, because it uses a resource without its own locking model.Finally, if the synchronization is found to be unnecessary, it could be eliminated either by the developer or by a transformation program, thereby obtaining a performance improvement and a more stable system. Detecting synchronization requirements (e.g., race detection) and code restructuring issues are outside the scope of this paper.

In summary, the contributions of this paper are as follows:

- Abstraction. We create a new abstraction, the monitor dataflow graph, which is useful for detecting and tracking synchronization points.
- Analysis. We produce an algorithm for the computation of monitor sets and prove that it always terminates. Although the algorithm is stated for Java, only minor modifications are required for other languages and systems that support monitors.
- Scalability. Our approach uses the two preceding pieces to detect unwanted synchronization, and it has been shown to work for large programs whose invocation graphs have $O(10^5)$ nodes and edges.

The organization of the paper is as follows. We discuss related work in Section 2, and we give several detailed examples in Section 3. Section 4 presents the basic pieces of our approach. Section 5 points out limitations of our technique and suggests further work.

2. RELATED WORK

The literature focuses on two different techniques for dealing with the unnecessary synchronization when it is found.

The most prevalent approach is to transform programs automatically to more optimal counterparts. Fitzgerald et al. [13] used static analysis to detect if no thread objects are ever started; in that case, all synchronization may be removed. Aldrich et al. [2] used an analysis based on a CFA(1) [24] call graph (and a CFA(0) call graph for the larger problems) to eliminate unnecessary synchronization where a monitor is accessible to only one thread, where a monitor is entered by the same thread multiple times, and where one monitor is nested within another. They succeeded in eliminating up to 70% of the synchronization and improving runtimes by as much as 5%. A number of authors employed "escape analysis," i.e. deriving conditions under which objects are reachable only by single threads. Bogda and Hölzle [4] located such objects with a flow insensitive and context insensitive analysis; they were able to get speedups of up to 36%. Choi et al. [6] and Blanchet [3] applied escape analysis to both stack allocation and synchronization removal; they removed averages of 51% and 20% of synchronizations on test programs and got speed improvements of up to 23% and 44%, respectively. Whaley and Rinard [28] added a "points to" analysis to the escape analysis to achieve synchronization removal on test cases between 24% and 64%. Ruf [22] tracked objects synchronized by only one thread but also partitioned aliased expressions into equivalence classes to get the effect of context sensitive analyses. His methods achieved impressive optimizations of both single and multiple threaded programs. Finally, Sălcianu and Rinard [23] also created a new abstraction, the "parallel interaction graph," to compute pointer, escape, and ordering information in multiple threads. They eliminated some synchronization between objects that span multiple threads, and their approach was especially useful for the use of region-based storage allocation.

A second approach is to use the unnecessary synchronization information in performing further manual analysis to decide how to tune the original design, possibly by making extensive program changes. Heydon and Najork [15] discussed the effect of removing synchronization from methods in the Java core libraries. They used the `srcjava` Java runtime as a performance debugging tool, utilizing reporting features within it to monitor aspects of program performance for a Web crawler they had designed. They identified places where synchronization inside the Java runtime library significantly affected performance and manually identified cases where the synchronization was unnecessary and could be removed. After removing it, the fraction of cycles spent on synchronization dropped from 20% to 1.5%. Gunther [14] observed that performance in servlets could be improved by avoiding unnecessary synchronization, especially in the major methods such as `service`, `doGet`, or `doPost`; he recommended the use of fine-grained synchronized blocks to protect critical sections of the code.

Thus several techniques exist for detecting unnecessary synchronization. All of the work deals with problems of moderate size and complexity (e.g. `javac`, `jacacup`, `pizza`, `jlex`), typically of size $O(10^4)$–$O(10^5)$ lines of source code. Usually unnecessary synchronization was detected via static analysis or related techniques Most of these analyses focus on reducing the overhead caused by the synchronization itself, i.e. the cost of performing locking and unlocking operations. This paper, on the other hand, focuses on the potentially much larger gains obtainable by allowing threads to proceed in parallel when unwanted synchronization is removed. In some cases, the payoff approaches a factor of the number of threads.

3. MOTIVATING EXAMPLES

Our fundamental observation is that when "out of the box" operations are synchronized, their cost is proportional to the amount of time the call takes to execute. Since our objective is to improve throughput, removing such synchronization achieves speedups by exploiting parallelism. These results apply to any scenario that involves a remote call. In Java, these include calls to databases (JDBC), directories (JNDI / LDAP), network operations, EJBs, servlets, JSPs, authentication, and web services. We also observe that unsynchronized operations outperform synchronized ones for CPU-bound ("in the box") cases as well.

Web applications often need to obtain information from remotely located resources. Portal servers generate web pages by pulling resources (e.g., XML and HTML documents, graphics) to generate customized web pages. Also, clients retrieve configuration values (e.g., security authorization policies) that reside in a server.

We provide several examples of the preceding observations. We use a simplified program to illustrate the problem. In Figure 1 the class `webExampleSyn` simulates the effects of multiple clients performing interactions on the web. The main method uses the constructor (lines 11–13) to create *n* threads and then starts them. Each thread calls the run method that invokes 1000 network-based interactions, each of which begins by opening a network (URL) connection and fetching a properties file (line 31). The `getProperties` method creates an input stream from a Uniform Resource Locator (line 18) and uses it to load the properties (line 19). In our terminology, `webExampleSyn.run` is a head, because it initiates the execution of threads. The method `java.util.Properties.load` is a tail, an expensive operation. Our analysis identifies the synchronization sources, one of which is `webExample.getProperties`, since it is declared as synchronized. We ran this program on an IBM IntelliStation with one 933 MHz Pentium III processor, 1GB of memory, Windows 2000, and Java JRE 1.3.1 accessing property files of various sizes on an IBM IntelliStation M Pro with one 400 MHz Pentium II processor, 256M of memory, and Windows NT. The two machines were on the same local area network. The results appear in Table 1. The number of threads and size of the property file are in columns 1 and 2, respectively, and all other columns give running times in milliseconds. There are two sets of data: when `getProperties` is not synchronized (col. 3–4) and when it is synchronized (col. 5–6). The Elapsed time column indicates how long it took for all the threads to complete. The Work time column is the sum of the times across all threads to complete the URL and load methods (lines 18 and 19); the cost of all other instructions was found to be negligible. Table 1 shows that the response time per thread is longer in the unsynchronized case than the synchronized case. However, we met our objective of improving throughput, as measured by the inverse of Elapsed time.

Note that in most cases

$$\text{Elapsed time} \approx \text{Work time (synchronized)}$$

and

$$\text{Elapsed time} \approx \text{Work time / Threads (unsynchronized)},$$

indicating no overlap and nearly perfect thread processing overlap, respectively. Although there are instances where the synchronized times are smaller, for the most part unsynchronized outperforms synchronized; sometimes the differences are striking, e.g. a 229% increase for 16 threads and file size 10,000.

Suppose the application designer concluded that the input stream (line 18) could be created once (i.e. moved after line 29) and passed into `getProperties` by each thread. The results would be as in Table 2. The unsynchronized cases win more often than before, but by smaller amounts. However, there are still considerable gains by removing synchronization, e.g. a 68% increase for 16 threads and file size 10,000.

For this simple example our algorithm reports the path from `run` to `getProperties` and from `getProperties` to `load`. The designer may examine the paths generated by our analysis and reassess whether synchronization is really necessary. The information provided localizes the places where changes are most likely required to avoid the synchronization.

As a second example, we replace the property file access by a call to a servlet that has a simulated execution time specified as a number of milliseconds. The results appear in Table 3. Note that in the synchronized case, the times grow linearly. But, the unsynchronized data show the benefits of parallelism, i.e. again Elapsed time ≈ Work time / Threads.

```
    ...
 5. public class webExampleSyn extends Thread {
        ...
 8.     private int threadNumber;
        ...
11.     public webExampleSyn( int threadNumber ) {
12.         this.threadNumber = threadNumber;
13.     }
14.
15.     static private synchronized Properties getProperties( ) {
16.         Properties props = new Properties( );
17.         try {
18.             InputStream is = new URL( ... ).openStream( );
19.             props.load( is );
20.             is.close( );
21.         }
            ...
25.         return props;
26.     }
27.
28.     public void run( ) {
29.         System.out.println( "Thread " + threadNumber + " started." );
30.         for ( int i = 0; i < 1000; i++ ) {
31.             Properties props = getProperties( );
32.             // Do processing involving the properties ...
33.         }
34.     }
35.
36.     public static void main( String[ ] s ) {
            ...
39.         int n = Integer.parseInt( s[0] );
40.         Thread[ ] thread = new Thread[ n ];
41.         for ( int i = 0; i < n; i++ )
42.             ( thread[ i ] = new webExampleSyn( i ) ).start( );
43.         ...
53.     }
54. }
```

Figure 1. Web client example

For a third example, we replace the property file access by a CPU-bound computation, the multiplication of 50x50 matrices on a two-processor machine. As Table 4 shows, we get increases of between 23% and 61% in Elapsed time for synchronized cases vs. unsynchronized ones.

Our final example is a large one, with an invocation graph of more than 25,000 nodes and more than 60,000 edges. The application includes a serialized data structure with a

Table 1. Web client example, URL in loop

		Unsynchronized		Synchronized	
Threads	**File size**	**Elapsed time**	**Work time**	**Elapsed time**	**Work time**
1	10	5,812	5,718	5,766	5,671
16	10	77,578	1,202,875	88,578	87,954
128	10	634,140	77,808,115	715,063	710,232
1	1,000	15,422	15,313	15,141	15,001
4	1,000	65,453	248,767	60,703	60,264
16	1,000	335,281	5,033,210	231,344	230,286
1	4,000	422,625	422,453	417,485	417,264
4	4,000	315,250	1,191,122	1,699,000	1,698,640
16	4,000	1,137,625	17,631,950	7,044,157	7,041,796
1	10,000	537,703	537,609	536,828	536,735
4	10,000	668,266	2,634,600	2,100,438	2,100,201
16	10,000	2,697,156	42,432,394	8,866,797	8,865,391

Table 2. Web client example, URL out of loop

		Unsynchronized		Synchronized	
Threads	**File size**	**Elapsed time**	**Work time**	**Elapsed time**	**Work time**
1	10	313	187	312	219
16	10	6,875	100,395	9,266	9,190
128	10	101,765	10,742,483	81,031	81,888
1	1,000	359	249	344	281
4	1,000	1,000	3,375	2,391	2,281
16	1,000	7,250	108,291	9,344	9,297
1	4,000	687	593	531	423
4	4,000	1,375	4,750	2,656	2,563
16	4,000	6,890	106,193	10,454	10,453
1	10,000	640	562	2,000	1,922
4	10,000	1,687	6,096	3,203	3,047
16	10,000	7,172	90,072	12,015	12,158

Table 3. Servlet example

		Unsynchronized		Synchronized	
Threads	**Wait**	**Elapsed time**	**Work time**	**Elapsed time**	**Work time**
1	1	21,033	20,904	21,113	20,987
8	1	22,049	174,459	160,598	160,126
32	1	27,143	803,804	637,345	636,314
1	10	20,951	20,818	21,012	20,865
8	10	22,120	174,828	162,052	161,633
32	10	27,662	805,704	644,326	643,235
1	100	111,236	111,098	111,215	111,073
8	100	111,536	890,232	884,225	883,783
32	100	114,974	3,628,926	3,535,043	3,533,103
1	1000	1,011,294	1,011,172	1,012,453	1,012,253
8	1000	1,011,182	8,088,108	8,099,932	8,098,954
32	1000	1,011,823	32,695,932	33,088,823	33,086,292

Table 4. CPU-bound example

	Unsynchronized		Synchronized	
Threads	**Elapsed time**	**Work time**	**Elapsed time**	**Work time**
1	7,828	7,750	7,813	7,718
2	9,532	18,799	15,328	15,127
4	21,797	82,544	30,563	30,376
8	46,828	350,546	61,172	60,604
16	98,563	1,431,629	122,610	121,564
32	200,125	5,709,709	246,391	244,935

`StringBuffer` field. Our analysis runs in 218 sec., producing 10,368 paths from heads to sources and 7,373 paths from sources to tails. For instance, our algorithm locates the path, i.e. the calling sequence, shown in Figure 2 from a method `java.langStringBuffer.readObject`, with undocumented synchronization, to the tail `java.net.InetAddress.getByName`, found to be costly by Heydon and Najork [15]. This path could cause a bottleneck and is unlikely to be discovered by a human.

4. MONITOR COMPUTATION

In this section we derive the quantities required for our unwanted synchronization analysis. First we describe an enhanced control flow graph called the *monitor dataflow graph*. Then we formally define the sets and functions that are needed. Finally, we present the *monitor dataflow algorithm* that describes how to compute the sets and functions. We omit discussion of the path generation, since it can be computed in a number of ways, e.g. see [9].

```
 1. StringBuffer.readObject Line 1090 calls
 2. ObjectInputStream.defaultReadObject Line 525 calls
 3. ObjectInputStream.inputClassFields Line 2268 calls
 4. ObjectInputStream.readObject Line 372 calls
 5. ObjectInputStream.inputClassDescriptor Line 942 calls
 6. ObjectStreamClass.setClass Line 572 calls
 7. ObjectStreamClass.lookupInternal Line 118 calls
 8. ObjectStreamClass.init Line 407 calls
 9. AccessController.doPrivileged calls
10. ObjectStreamClass$2.run Line 426 calls
11. ObjectStreamClass.access$200 Line 52 calls
12. ObjectStreamClass.computeSerialVersionUID Line 861 calls
13. MessageDigest.getInstance Line 129 calls
14. Security.getImpl Line 809 calls
15. Security.reloadProviders Line 183 calls
16. AccessController.doPrivileged calls
17. Security$2.run Line 187 calls
18. Provider.loadProvider Line 152 calls
19. ClassLoader.loadClass Line 257 calls
20. Launcher$AppClassLoader.loadClass Line 381 calls
21. ClassLoader.loadClass Line 325 calls
22. URLClassLoader.findClass Line 203 calls
23. AccessController.doPrivileged calls
24. URLClassLoader$ClassFinder.run Line 542 calls
25. URLClassPath.getResource Line 139 calls
26. URLClassPath$FileLoader.getResource Line 688 calls
27. URLClassPath.check Line 346 calls
28. SecurityManager.checkConnect Line 1048 calls
29. SocketPermission.<init> Line 214 calls
30. SocketPermission.init Line 373 calls
31. InetAddress.getByName
```

Figure 2. Large example.

4.1 INVOCATION GRAPH

Our starting point is the Java Bytecode Analysis (JaBA) system [18], [19]. It takes as input the object code of a collection of Java classes, and produces an interprocedural invocation graph [12] and data flow analysis. These form the basis upon which the monitor dataflow graph is constructed:

- The intraprocedural analysis of each method referenced in the call graph is path insensitive and flow sensitive. A *path insensitive* control flow graph analyzes all paths through all basic blocks in each method. The control flow graph is *flow sensitive*, because it considers the order of execution of the instructions within each basic block, accounting for local variable kills and casting of object references. However, instance and class (static) fields are flow insensitive because field kills are not respected, though casting of object references is respected.
- Each node in the invocation graph contains the following state:
  - the target method;
  - for instance methods, an allocation site (or type) for each of the method's potential receivers;
  - all parameters to the method, represented as a vector of sets of possible allocation sites (or possible types);
  - a set of possible return value allocation sites (or types) from this method at this node;
  - the set of basic blocks which comprise the target method;
  - the set of monitor objects used by `monitorenter` and `monitorexit` instructions;
- Each node in the invocation graph is uniquely identified by its calling context, namely, the target method, the set of possible receiver types, and the parameters' types. Thus the graph is *context sensitive*.
- The edges in the invocation graph are directed, where each edge points from a call site within a method to a target method.
- The invocation graph is rooted and may contain cycles.
- Our implementation of the invocation graph allows bi-directional traversal, even though the edges in the graph are unidirectional. Therefore, from any node n within the invocation graph, we can find the set of its predecessor nodes, which we denote by $\pi(n)$; similarly, the set of successor nodes is indicated by $\sigma(n)$. We use the same successor / predecessor notation for the basic blocks in the control flow graph.
- Each node, edge, and basic block is labeled with a monitor set, described in the next subsection.

In addition to the invocation graph, a data flow analysis is performed with a precision to the level of allocation sites, as in the Cartesian Product Algorithm [1]. It is important to observe that the number of objects in a Java program as modeled by the monitor dataflow graph is always finite. There are a finite number of calls to `new` in the object code, and the elements of arrays and other collections are modeled as single elements.

We now give a more precise definition of the head nodes. A head node is any successor of the invocation graph root node. The user can prune the set of head nodes and/or add additional head nodes.

4.2 MONITOR SET DEFINITION

In this section we formalize the notion of Java synchronization monitors. We describe sets of monitors associated with nodes, edges, or basic blocks of the monitor dataflow graph. The monitor set for a node indicates all monitors that may be used as locks when the node's target method is invoked. Since the method may contain synchronized blocks, different edges leading out of the node may have different monitors that arise from those synchronized blocks. The monitors are propagated along edges and nodes in a manner that will be made clear by our formalism.

Let $G = (N, E)$ be the monitor dataflow graph, derived from the invocation graph described in the previous section, representing a collection of Java classes with a specified set of entry points. The nodes N are given by a set of triples $n = (M, R, P)$, where M is the target method and R and P are the sets of possible receiver types and parameter types, respectively. Each method consists of a set B of basic blocks b. Because in Java a method has only one entry point, there is a unique basic block b_0 for which $\pi(b_0) = \phi$. The edges are described by a set of triples $e = (n, b, n')$, where $n = (M, R, P)$ and $n' = (M', R', P')$ are nodes and b is a basic block in M ending with an edge to M'. This definition allows the possibility of more than one edge between a single pair of nodes. We can of course define paths $v_1 \to v_2 \to \ldots \to v_m$, where the v_i are all either nodes or basic blocks and $v_i = \sigma(v_{i-1})$, $i = 2, \ldots, m$, i.e., v_i is a successor of v_{i-1}. Paths may include cycles.

A Java expression of the form

```
synchronized(o) { … }
```

is represented in object code by a matching pair of `monitorenter` and `monitorexit` instructions. We require that the basic blocks of any method be well formed, in that they correspond to legal Java code. This assumption ensures that synchronized blocks will be properly balanced: for each `monitorenter` instruction, there exists a matching set of `monitorexit` instructions. There may be more of the latter than the former, because Java compilers insert extra `monitorexits` in `catch` and `finally` blocks.

A *monitor object* is a pair (o, c), where o is an object in the Java program and c (the *counter*) is a positive integer bounded by a fixed constant Ω. A *monitor set* is a collection of monitor objects in which all the objects o are distinct. We have observed that in the monitor dataflow graph model of a Java program, the total number of Java objects is finite. Hence the number of different possible monitor sets for that program is finite.

In the Java virtual machine specification [20] there is no value provided for the maximum counter size Ω; presumably it is so large that other significant resource limits would be reached first. We simply assume that Ω is larger than any nesting of `monitorenter-monitorexit` pairs in the program we are analyzing.

There is a natural partial order $\leq$, based on counter values, on the set of all monitor sets: $m_1 \leq m_2$, if for each $(o, c) \in m_1$, $(o, d) \in m_2$ for some counter d, $c \leq d \leq \Omega$. One can then define $<$ as $m_1 < m_2$ if $m_1 \leq m_2$ and $m_1 \neq m_2$. For a fixed Java program the set of all possible monitor sets for that program satisfies an important condition: any ascending chain of the form

$$m_1 < m_2 < \ldots < m_k < \ldots \text{ is finite.} \qquad (4.2.1)$$

We next want to define monitor sets for basic blocks, edges, and nodes. For a given method M with basic block set B and $b \in B$, we first motivate the definition of the monitor set $\mu(b)$. Suppose exactly one basic block, say $b_n \neq b$ (where n represents `monitorenter`) contains a `monitorenter` instruction associated with object o, and another basic block, say b_x (where x represents `monitorexit`) contains the matching `monitorexit` instruction. Suppose also that there are no possible paths of basic blocks with cycles. If there is a path

$$b_n \to \ldots \to b \to \ldots \to b_x$$

of basic blocks from b_n to b_x which includes b, we would say that $(o,1)$ is in $\mu(b)$, i.e. all the instructions in b execute under the guard of the monitor object o. Of course, we have to allow for cycles and synchronization blocks nested c levels deep. The natural generalization and formal definition, which handles both cycles and nesting is as follows. For a fixed object o, let c be the largest positive integer such that there exist:

- A set of integers $n_1, \ldots, n_c$ representing blocks containing `monitorenter` instructions;
- A set of integers $x_1, \ldots, x_c$ representing blocks containing `monitorexit` instructions;
- A collection of basic blocks such that b_{n_i} in B contains a `monitorenter` instruction for which o is the associated object, $i = 1, \ldots, c$;
- A collection of basic blocks such that b_{x_i} in B contains one of the `monitorexit` instructions corresponding to b_{n_i}'s `monitorenter` instruction, $i = 1, \ldots, c$.
- A path of the form:

$$b_{n_1} \to \ldots \to b_{n_2} \to \ldots \to b_{n_c} \to \ldots \to b \to \ldots \to b_{x_c} \to \ldots \to b_{x_2} \to b_{x_1}$$

Then we say that (o, c) is in $\mu(b)$ if $c \leq \Omega$, and (o, Ω) is in $\mu(b)$ if $c > \Omega$, where Ω, is the maximum possible counter value for any monitor object. In the absence of cycles c will be less than Ω, by the assumption we made about Ω. However, when cycles are present, then monitor objects of the form (o, Ω) may exist.

Next, for edges e and nodes n, the definitions of the monitor sets $\mu(e)$ and $\mu(n)$ are recursive and interdependent. We give a three-part definition for $\mu(e)$. An edge $e = (n, b, n')$ joins a node n to a node n' via a call from a basic block b. Hence any monitors in effect at the node n or the block b will be in force for e. Thus a monitor object (o, c) is in $\mu(e)$ if either of the following conditions holds:

i. $(o, c) \in \mu(b)$;
ii. $(o, c) \in \mu(n)$.

However, when $(o, p) \in \mu(n)$ and $(o, q) \in \mu(b)$, then both the edge and block contribute to synchronizing the successors of e. For example, (o, p) could arise via p consecutive calls of a synchronized method, and (o, q) could come from q nested

synchronized blocks. Backing out of the synchronization would require $p + q$ levels. Therefore,

iii. if $(o, p) \in \mu(n)$ and $(o, q) \in \mu(b)$, then $(o, p + q)$ is in $\mu(e)$ when $p + q \leq \Omega$, and otherwise (o, Ω) is in $\mu(e)$.

Finally, a node $n = (M, R, P)$ will be guarded by monitors on any edge which leads to n. Furthermore, n will have monitors if its target method M is declared to be `synchronized`. With these observations we can see that a monitor object (o, c) is in $\mu(n)$ for node $n = (M, R, P)$ if one of the following three conditions holds:

i. M is `static` and `synchronized`, o is the class object of the class which declares M. Either $c = 1$ or $(o, c - 1) \in \mu(e)$ for an edge ending at n;

ii. M is `synchronized`, but not `static`, o is any element of R, and either $c = 1$ or $(o, c - 1) \in \mu(e)$ for an edge ending at n;

iii. M is not `synchronized`, and $(o, c) \in \mu(e)$ for an edge ending at n.

Observe that the monitor sets for basic blocks capture only local behavior within a single method, i.e. their computation only requires an intraprocedural analysis. The monitor sets for nodes and edges necessitate interprocedural analysis using the results of the basic block computation. All the $\mu(b)$ are computed first, and then used to compute $\mu(e)$ and $\mu(n)$ together.

4.3 MONITOR DATAFLOW ALGORITHM

Next we present the monitor dataflow algorithm for computing the monitor sets according to the preceding definitions. It breaks into two pieces, one which handles basic blocks and one which deals with nodes and edges. They appear in Figures 3 and 4.

The monitor dataflow algorithm can be viewed as calculations on a semilattice [17] with $\cup$ as the join operation. We introduce two other operations, + and –, involving monitor sets. The state transformation functions are defined using + and – in Figure 3, and + in Figure 4. We first define all three operations for a monitor set m and an object o.

We define plus (+), corresponding to `monitorenter`, as follows:

- if $(o, c) \in m$ for some c, we replace (o, c) in m by $(o, \min(c + 1, \Omega))$;
- otherwise, we add $(o,1)$ to m.

The resulting set is $m + o$.

For the minus (–) operation, corresponding to `monitorexit`,

- if $(o,1) \in m$, we remove it from m;
- if $(o, c) \in m$ for $c > 1$, we replace (o, c) in m by $(o, c - 1)$.

The resulting set is $m - o$. Note the importance of our assumption that Ω is larger than the maximum nesting in programs to be analyzed. Choosing an Ω that is too small will result in premature removal of o from the monitor set m.

For the union ($\cup$) of a set m with an object o,

```
 1. μ(b) ← φ for all basic blocks b
 2. queue ← b0 (the entry basic block)
 3. while queue ≠ φ:
 4.   b ← pop(queue)
 5.   mb ← μ(b)
 6.   if b has monitorenter with object O then mb ← mb + O
 7.   if b has monitorexit with object O then mb ← mb − O
 8.   for all s ∈ σ(b):
 9.     ms ← μ(s)
10.     μ(s) ← ms ∪ mb
11.     if μ(s) ≠ ms then push(s)
```

Figure 3. Computing $\mu(b)$.

```
 1. μ(x) ← φ for all call sites and edges.
 2. Compute in(x), ν, and ε for all call sites and edges.
 3. push n with in(n) ≠ φ or with in(e) ≠ φ and e=(n,b,n') for some b and n'.
 4. while queue ≠ φ:
 5.   pop(n)
 6.   μ(n) ← μ(n) + in(n)
 7.   for all edges e=(n,b,n') leaving n:
 8.     μ(e) ← in(e) + μ(n)
 9.     t ← μ(n') ∪ μ(e)
10.     if t ≠ μ(n') then:
11.       μ(n') ← t
12.       push(n')
```

Figure 4. Computing $\mu(n)$, $\mu(e)$, and source functions ν and ε.

- if o does not appear in any monitor object of m, add (o, c) to m;
- if $(o, d) \in m$ with $d < c$, replace (o, d) by (o, c) in m.
- if $(o, d) \in m$ with $d \geq c$, m remains unchanged.

The resulting set is $m \cup (o, c)$.

We extend +, –, and $\cup$ to operations on two sets m_1 and m_2 by computing m_1 (+, –, or $\cup$) o for each $o \in m_2$.

The union operation expresses the fact that a graph node successor inherits the monitor sets of its predecessors: if node i has monitor set m_i, $i = 1, 2$, and node 2 is a successor of node 1, then at some point in the computation m_2 will be replaced by $m_1 \cup m_2$. In terms of the partial ordering $\leq$ introduced earlier, the union operation satisfies

$$m_1 \leq m_1 \cup m_2 \text{ for all } m_1 \text{ and } m_2. \qquad (4.2.2)$$

Given a method with a set B of basic blocks, the piece of the monitor dataflow algorithm for computing $\mu(b)$ for each $b \in B$ appears in Figure 3. Steps 1 and 2 initialize all the sets and prime the queue with the entry basic block. Steps 3–11 perform a fixed point iteration to refine the definitions of each $\mu(b)$. A basic block is popped (step 4), and its current monitor set is saved (step 5). The saved value is updated if the block ends with either a `monitorenter` or `monitorexit`, using the + or – operation, respectively (steps 6, 7). Then for each successor of the popped block (step 8), we compute the union of its monitor set with the saved value (steps 9–10), thus propagating predecessors' monitor sets to successors. If the successor's monitor set changed, the successor is added to the queue (step 11). Without this test, the algorithm would loop endlessly in the presence of cycles.

The portion of the monitor dataflow algorithm for computing $\mu(e)$ and $\mu(n)$ for nodes and edges appears in Figure 4. It also computes two functions that are needed for reporting source information, i.e. where synchronization originates. For each object o contained in a monitor object (o, c), the *node source* function $\nu(o)$ is the set of nodes whose target methods are synchronized on the object o. The *edge source* function $\varepsilon(o)$ is the set of edges each of which is contained within the basic block sequence initiating the synchronization under object o via a matched `monitorenter`-`monitorexit` pair.

The main idea of the algorithm is to propagate monitor sets from predecessors to successors in the monitor dataflow graph, by passing the monitor sets and associated nodes into a queue and performing a fixed point iteration (steps 4–12). Step 1 defines all $\mu(e)$ and $\mu(n)$ to be null sets, and then initial values $in(e)$ and $in(n)$ are computed (step 2). The value $in(e)$ is simply $\mu(b)$, where $e = (n, b, n')$. The value $in(n)$ is the result of carrying out parts i and ii of the definition of $\mu(n)$ in Section 4.2. At this time the source maps are also computed, and they are simply inversions of $in(e)$ and $in(n)$: e [respectively n] $\in \varepsilon(o)$ [respectively $\nu(o)$] if $(o, c) \in in(e)$ [respectively $in(n)$] for some c. Step 3 initializes the queue with all the relevant nodes for which computation should begin. They are any nodes n whose initial value $in(n)$ is non-empty or which begin an edge, that is, $e = (n, \ldots)$, whose initial value $in(e)$ is non-empty. The main loop (steps 4–12) begins by popping a node (step 5) and updating its monitor set value to include the initial values (step 6). Then the monitor information is propagated from all edges $e = (n, b, n')$ leaving n (steps 7–12). The edge monitor set is updated to include the monitors which cover the entire node (step 8), and the monitors of the edge apply to the successor node n' (steps 9–11). If the computation changed the successor's monitor set, it is placed back on the queue (steps 10–12). Note that in updating we use the + operator in steps 6 and 8, because we want to reflect the cumulative change in the counters (see condition iii in the definition of $\mu(e)$ in Section 4.2). In relating a successor's monitor set to that of its predecessor, we use the $\cup$ operator, as mentioned earlier in the definition of union.

We conclude this section by proving that the algorithms in Figures 3 and 4 always terminate. In both algorithms a basic block or graph node x is pushed onto the queue only if $\mu(x)$ changes as the result of a union operation

$$\mu(x) \leftarrow \mu(x) \cup m$$

for some monitor set m, that is,

$$\mu(x) < \mu(x) \cup m.$$

However, by (4.2.1) and (4.2.2), this condition can hold for a fixed x only a finite number of times. Thus the queue empties after a finite number of steps, and the algorithms always terminate. Our proof is similar to that of Kildall [17].

We remark that the interprocedural control and data flow precision in our analysis is crucial. We identify attributes such as parameter types and receiver types associated with method invocations. With less precision, our results would be more conservative. As an example, in Figure 1 `load()` is `synchronized`. However, our analysis concludes that there is a unique receiver `props` for each thread. Hence this particular synchronization is not of concern, since each thread has its own separate monitor.

In the worst case, the algorithm is exponential. However, on realistic applications, the runtime is quite reasonable. For example, an analysis of ECPerf [25], with an analysis scope of 20,000 classes, it ran for less than five minutes.

Although we have stated our algorithms in the context of analyzing Java code, they can be applied to any language that supports monitors. Examples that come to mind are Concurrent Euclid [16], Concurrent Pascal [5], Mesa/Cedar [26], and Modula-2 [10].

5. ASSESSMENT AND FUTURE WORK

It is appropriate to compare our approach to the use of dynamic techniques, including runtime profilers. There are at least three weaknesses of only using runtime profiling:

- the tools to profile a large set of library routines are not always available to all developers;
- programmers usually do not have access to a comprehensive set of test cases to profile;
- profiling usually can not adequately cover all load conditions that may arise, especially in complicated web application environments. This includes the simulation of a large number of failure scenarios that are hard to replicate.

The advantage of our approach is that all paths through the code are covered, and unwanted synchronization can be discovered

during development, before the code goes into production where it is much more expensive to diagnose and repair. Because our approach is conservative and can generate false positives, the developer must analyze the results. The key is to minimize the false positive rate. However, the results from our algorithm can be used to direct the use of a runtime profiler. We suspect that the combination can further reduce the false positive rate. We believe that profiling and static methods are complementary and can best be used together (e.g., Choi et al.[7]).

Another important question is how we handle native methods (methods written in a language other than Java). This is more a question about the underlying JaBA static analysis framework rather than about the algorithms presented in this paper. In fact, the JaBA implementation includes hand coded implementations of the most widely used native methods. However, we do not have a complete model of all native methods in the Java runtime libraries. Native methods must be added as needed to achieve complete control and data flow analyses.

The next issue is the modeling of Java reflection. As with native methods, the issue is about how the interprocedural control and data flow analyses handle it. We have successfully constructed a prototype of JaBA that handles `Class.forName()` and `Class.newInstance()`. In some cases the names of the classes are known, because the names are string constants. In cases where the name of the class is computed or read from an external source (e.g., a file), cast operations on the result of `newInstance()` are used to get a first order approximation of the classes being instantiated. Class hierarchy analysis can also be used to determine the possible classes to instantiate. In some cases, programmers will need to provide assistance in identifying which classes should be instantiated by the `newInstance()` calls. We believe that other Java reflection methods can also be handled, including calls to methods (including constructors) and access to fields.

The next challenge is with modeling Java's dynamic loading and binding of classes during runtime. JaBA models `ClassLoader` trees, which is important in defining name spaces for complex applications. In server environments, and particularly those that conform to the J2EE[1], the analysis can be closed world, since all of the code to be deployed in the server is known. Dynamically loaded code from unknown sources (e.g., mobile code from a web site) is not part of the J2EE programming model. So, the algorithms we describe in this paper are sufficient for a large and important set of applications. The algorithms would need to be extended to handle open world analysis where the classes being called are outside the analysis scope. In particular, the reporting of unwanted synchronization would need to identify places where the interprocedural analysis might call code outside the analysis scope, and therefore not identify synchronized tail nodes and/or sources of synchronization.

Our algorithm detects unwanted synchronization. The next step is to identify whether the synchronization is actually necessary. The addition of race detection (see e.g. [7], [8]) is one possible approach to determine whether the synchronization is really required. In the presence of potential race conditions, there is a need to identify program transformations that can eliminate the unwanted synchronization. One possible approach is to use program slicing [11], [27] to identify statements in the program that require synchronization versus those parts of the program for which synchronization is unwanted.

An important issue is the integration of this algorithm into a tool with acceptable usability characteristics. In particular, mechanisms are needed to minimize the potentially large number of paths that may be reported. Specifically, synchronization often is necessary. We need to develop reporting filters that can be applied to omit paths with such synchronization sources.

In our prototype, we made a number of simplifying assumptions. At the moment, our implementation of the algorithm does not process `catch` and `finally` blocks. Handling of `catch` blocks should be straightforward when the invocation graph contains edges for thrown and caught exceptions.

6. CONCLUSIONS

In this paper we have presented an approach to improving program performance by automatically calculating paths in an invocation graph from head nodes to tail nodes that have been identified as having unwanted synchronization. The paths go through the program points that originate the synchronization and thus create the opportunity to assess the feasibility of removing the synchronization, perhaps via a program restructuring. For large applications our approach performs an analysis that can not be done manually. The algorithms can be integrated within software development environments so that the programmer can identify unwanted synchronization trouble spots while the code is being developed. Since our prototype works on object code, it can handle middleware and applications, even when the source code is unavailable. While our algorithm and prototype described in this paper were written for Java, the basic concept and algorithms applies to other languages and runtimes that have a comparable structure for monitor enters / exits.

7. ACKNOWLEDGMENTS

We wish to thank Peter Sweeney for building program performance traces to validate out intuitions about expensive tail nodes; Jong-Doek Choi, Harini Srinivasan, and Peter Sweeney for constructive criticism of the paper; and David Grove, Julian Dolby, and Jong-Doek Choi for insights about the monitor dataflow algorithm. We would also like to thank our colleagues in the IBM Haifa Research Laboratory, particularly Sara Porat, and Marina Biberstein, for their contributions of the class file parser, control flow graph, and a Java virtual machine simulator framework used by JaBA.

8. REFERENCES

[1] O. Agesen. The cartesian product algorithm, in Proceedings of the Ninth European Conference on Object-Oriented Programming (Aarhus, Denmark, August 1995), 2-26.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggars, Static Analyses for Eliminating Unnecessary Synchronization from Java Programs, in Proceedings of Static Analysis Symposium (Venice, Italy, September 1999), Lecture

[1] J2EE is a trademark of Sun Microsystems, Inc.

Notes in Computer Science, A. Cortesi and G. File (Eds.), vol. 1694, Springer-Verlag, 1999, 19-38.

[3] B. Blanchet. Escape analysis for object oriented languages. Application to Java, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Denver, Colorado, November 1999), 20-34.

[4] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Denver, Colorado, November 1999), 35-46.

[5] P. Brinch Hansen. The programming language concurrent Pascal. IEEE Transactions on Software Engineering 1 (2), June 1975, 199-207.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Denver, Colorado, November 1999), 1-19.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs, in Proceedings of the ACM SIGPLAN Conference on programming language design and implementation (Berlin, Germany, June 2002), 258-269.

[8] J.-D. Choi and S. L. Min. Race frontier: reproducing data races in parallel-program debugging, in Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming (Williamsburg, Virginia, April 1991), 145-154.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms, The MIT Press, Cambridge, Massachusetts, 1992.

[10] B. Cornelius. Programming with TopSpeed Modula-2. Addison-Wesley, Reading, Massachusetts, 1991.

[11] A. De Lucia. Program slicing: methods and applications, in Proceedings of the First International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California (Florence, Italy, November 2001), 142-149.

[12] M. Emami, R. Ghiya, and L. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (Orlando, Florida, June 1994), 242-256.

[13] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steengaard, and D. Tarditi. Marmot: an optimizing compiler for Java, Software Practice and Experience, 30 (3), March 2000, 199-232.

[14] H. W. Gunther. White paper, Development Best Practices for Performance and Scalability, IBM WebSphere Application Server, Standard and Advanced Editions, September, 2000, http://www-900.ibm.com/websphere/download/pdf/ws_bestpractices.pdf.

[15] A. Heydon and M. Najork, Performance Limitations of the Java Core Libraries, JAVA '99 (San Francisco, California, June 1999), 35-41.

[16] R. C. Holt. Concurrent Euclid, the Unix System, and Tunis, Addison-Wesley, Reading, Massachusetts, 1983.

[17] G. A. Kildall. A Unified Approach to Global Program Optimization, in Proceedings of Principles of Programming Languages (Boston, Massachusetts, October 1973), 194-206.

[18] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Seattle, Washington, November 2002), 359-372.

[19] L. Koved, JABA—JAva Bytecode Analysis http://www.research.ibm.com/javasec/JaBA.html.

[20] T. Lindholm and F. Yellin. The Java Virtual Machine Specification, Addison-Wesley, Reading, Massachusetts, 1997.

[21] J. Plevak and A. Chien. Precise concrete type inference for object-oriented languages, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland Oregon, October 1994), 324-340.

[22] E. Ruf. Effective synchronization removal for Java, in Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (Vancouver, British Columbia, Canada, June 2000), 208-218.

[23] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs, in Proceedings of the 8th ACM Symposium on Principles and Practice of Parallel Programming (Snowbird, Utah, June 2001), 12-23.

[24] O. Shivers. Control-flow analysis in Scheme, in Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (Atlanta, Georgia, June 1988), 164-174.

[25] Sun Corporation. Java 2 platform, Enterprise edition (J2EE) Ecperf, http://java.sun.com/j2ee/ecperf/.

[26] W. Teitelman. A tour through Cedar. IEEE Transactions on Software Engineering, SE-11 (3), March 1985, 285-302.

[27] F. Tip. A survey of program slicing techniques, Journal of Programming Languages, 3 (3), September 1995, 121-189.

[28] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs, in Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Denver, Colorado, November 1999), 187-206.