Portable and Efficient Distributed Threads for Java

Eli Tilevich and Yannis Smaragdakis

College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{tilevich,yannis}@cc.gatech.edu
http://j-orchestra.org

Abstract. Java middleware mechanisms, such as Java RMI or CORBA implementations, do not support thread coordination over the network: synchronizing on remote objects does not work correctly and thread identity is not preserved for executions spanning multiple machines. The current approaches dealing with the problem suffer from one of two weaknesses: either they require a new middleware mechanism, making them less portable, or they add overhead to the execution to propagate a thread identifier through all method calls. In this paper we present an approach that works with an unmodified middleware implementation, yet does not impose execution overhead. The key to our technique is the bytecode transformation of only stub routines, instead of the entire client application. We argue that this approach is portable and can be applied to mostly any middleware mechanism. At the same time, we show that, compared to past techniques, our approach eliminates an overhead of 5.5-12% (of the total execution time) for applications from the SPEC JVM suite.

1 Introduction

The Java programming language offers high-level support for both distributed programming and concurrency, but the two mechanisms are unaware of each other. Java has an integrated middleware mechanism (Java RMI [15]). Any object can make its methods remotely accessible by implementing a `Remote` interface. This enables distributed programming without separate interface definitions and IDL tools. At the same time, Java supports the easy creation and monitor-style synchronization of threads. Any Java object is mapped to a unique monitor, with a single mutex and condition queue. Java code can synchronize and wait on any object. Nevertheless, synchronization does not carry over to remote objects. The problem is dual. First, synchronization operations (like `synchronized`, `wait`, `interrupt`, etc.) do not get propagated by Java RMI. For instance, attempting to explicitly lock the mutex of a remote object will lock the mutex of its RMI stub object instead. Second, thread identity is not maintained over the network. For instance, a thread that calls a remote method may self-deadlock if the remote operation happens to call back the original site.

In this paper, we present a mechanism that enables Java thread synchronization in a distributed setting. Our mechanism addresses monitor-style synchronization (mutexes and condition variables) which is well-suited for a distributed threads model. (This is in contrast to low-level Java synchronization, such as volatile variables and atomic operations, which are better suited for symmetric multiprocessor machines.)

Our work is not the first in this design space. Past solutions fall in two different camps. A representative of the first camp is the approach of Haumacher et al. [5]

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 478–492, 2004.
© IFIP International Federation for Information Processing 2004

where a replacement of Java RMI is proposed that maintains correct multithreaded execution over the network. If employing special-purpose middleware is acceptable, this approach is sufficient. Nevertheless, it is often not desirable to move away from standard middleware, for reasons of portability and ease of deployment. Therefore, the second camp, represented by the work of Weyns, Truyen, and Verbaeten [18], advocates transforming the client application instead of replacing the middleware. Unfortunately, clients (i.e., callers) of a method do not know whether its implementation is local or remote. Thus, to support thread identity over the network, *all* method calls in an application need to be automatically re-written to pass one extra parameter – the thread identifier. This imposes both space and time overhead: extra code is needed to propagate thread identifiers and adding an extra argument to every call incurs a run-time cost. Weyns, Truyen, and Verbaeten [18] quantify this cost to about 3% of the total execution time of an application. Using more representative macro-benchmarks (from the SPEC JVM suite) we found the cost to be between 5.5 and 12% of the total execution time. A secondary disadvantage of the approach is that the transformation becomes complex when application functionality can be called by native system code, as in the case of application classes implementing a Java system interface.

The technique we describe in this paper addresses both the problem of portability and the problem of performance. We follow the main lines of the approach of Weyns, Truyen, and Verbaeten: we replace all monitor operations in the bytecode (such as `monitorenter`, `monitorexit`, `Object.wait`) with calls to operations of our own distribution-aware synchronization library. Nevertheless, we avoid instrumenting every method call with an extra argument. Instead, we perform a bytecode transformation on the generated RMI stubs. The transformation is general and portable: almost every RPC-style middleware mechanism needs to generate stubs for the remotely invokable methods. By transforming those when needed, we can propagate thread identity information for all remote invocations, without unnecessarily burdening local invocations. Our approach also has the advantage of simplicity with respect to native system code. Finally, our implementation is fine-tuned, making the total overhead of synchronization be negligible (below 4% overhead even for empty methods and no network cost).

Our technique is implemented in the context of the J-Orchestra system [17]. J-Orchestra is an *automatic partitioning* system for Java programs: given a Java application and under user guidance, J-Orchestra can split the application into parts that execute on different machines. For a large subset of Java, the resulting distributed application will behave exactly like the original centralized one. Beyond J-Orchestra, the distributed synchronization technique described in this paper is applicable to other partitioning systems (e.g., Addistant [16], AdJava [3], FarGo [6]), language tools for distribution (e.g., Java Party [4][11], Doorastha [2]), or stand-alone mechanisms for distributed Java threads (e.g., Brakes [18]).

2 Background: Problem and Application Context

2.1 Distributed Synchronization Complications

Modern mainstream languages such as Java or C# have built-in support for concurrency. Java, for example, provides the class `java.lang.Thread` for creating and

managing concurrency, monitor methods `Object.wait`, `Object.notify`, and `Object.notifyAll` for managing state dependence, and `synchronized` methods and code blocks for maintaining exclusion among multiple concurrent activities. (An excellent reference for multithreading in Java is Lea's textbook [7].)

Concurrency constructs usually do not interact correctly with middleware implementations, however. For instance, Java RMI does not propagate synchronization operations to remote objects and does not maintain thread identity across different machines.

To see the first problem, consider a Java object `obj` that implements a `Remote` interface `RI` (i.e., a Java interface `RI` that extends `java.rmi.Remote`). Such an object is remotely accessible through the `RI` interface. That is, if a client holds an interface reference `r_ri` that points to `obj`, then the client can call methods on `obj`, even though it is located on a different machine. The implementation of such remote access is the standard RPC middleware technique: the client is really holding an indirect reference to `obj`. Reference `r_ri` points to a local RMI "stub" object on the client machine. The stub serves as an intermediary and is responsible for propagating method calls to the `obj` object. What happens when a monitor operation is called on the remote object, however? There are two distinct cases: Java calls monitor operations (locking and unlocking a mutex) implicitly when a method labeled `synchronized` is invoked and when it returns. This case is handled correctly through RMI, since the stub will propagate the call of a synchronized remote method to the correct site. Nevertheless, all other monitor operations are not handled correctly by RMI. For instance, a `synchronized` block of code in Java corresponds to an explicit mutex lock operation. The mutex can be the one associated with any Java object. Thus, when clients try to explicitly synchronize on a remote object, they end up synchronizing on its stub object instead. This does not allow threads on different machines to synchronize using remote objects: one thread could be blocked or waiting on the real object `obj`, while the other thread may be trying to synchronize on the stub instead of on the `obj` object. Similar problems exist for all other monitor operations. For instance, RMI cannot be used to propagate monitor operations such as `Object.wait`, `Object.notify`, over the network. The reason is that these operations cannot be indirected: they are declared in class `Object` to be `final`, which means that the methods can not be overridden in subclasses including the `Remote` interfaces required by RMI.

The second problem concerns preserving thread identities in remote calls. The Java RMI runtime, for example, starts a new thread for each incoming remote call. Thus, a thread performing a remote call has no memory of its identity in the system. Fig. 1

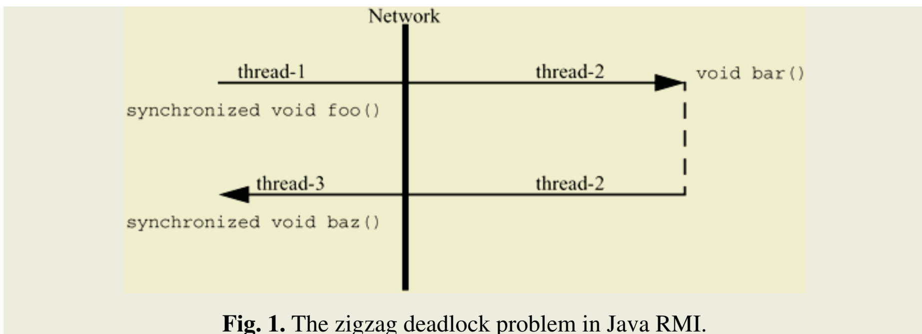

Fig. 1. The zigzag deadlock problem in Java RMI.

demonstrates the so-called "zigzag deadlock problem", common in distributed synchronization. Conceptually, methods `foo`, `bar`, and `baz` are all executed in the same thread – but the location of method `bar` happens to be on a remote machine. In actual RMI execution, thread-1 will block until `bar`'s remote invocation completes, and the RMI runtime will start a new thread for the remote invocations of `bar` and `baz`. Nevertheless, when `baz` is called, the monitor associated with thread-1 denies entry to thread-3: the system does not recognize that thread-3 is just handling the control flow of thread-1 after it has gone through a remote machine. If no special care is taken, a deadlock condition occurs.

2.2 J-Orchestra

Our technique for correct and efficient monitor-style distributed synchronization has been applied in the context of J-Orchestra. J-Orchestra is a system that rewrites existing Java programs at the bytecode level into distributed programs that can be executed on multiple machines. The transformation is done automatically, once the user specifies (through a GUI) which parts of the code should be available on which machine. The emphasis is on the correctness of the partitioning process: for a large subset of Java, J-Orchestra-partitioned applications behave just like the original centralized ones [17]. That is, J-Orchestra emulates many of the language mechanisms of a Java VM over a collection of distinct VMs.

The reason we bring up the context of our work is that the need for correct distributed synchronization is even more pronounced in the case of J-Orchestra than in the case of regular distributed programming. Since J-Orchestra creates distributed applications automatically (i.e., without programmer intervention, beyond choosing locations for code parts) it is important to maintain the same synchronization mechanisms over a network as for a single machine. Furthermore, J-Orchestra is mainly applicable when an application needs to be distributed to take advantage of unique resources of different machines, instead of parallelism. For example, J-Orchestra can be used to partition a traditional Java application so that its GUI runs on one machine, its computation on another, sensor input is produced and filtered on a third machine and file storage occurs on a fourth. Nevertheless, the entire application may only have a single thread, even though it uses libraries that employ synchronization. J-Orchestra will partition this application so that its logic is still single-threaded, yet the implementation consists of multiple Java threads (at least one per machine), only one of which can be active at a time. Thus, with J-Orchestra, the deadlock problems resulting from the lack of remote thread identity can exhibit themselves even for a single-threaded application!

Other than the context and motivation, however, the discussion in the rest of this paper is not specific to J-Orchestra. Indeed, our technique can be applied to any system in the literature that supports distributed communication and threading.

3 Solution: Distribution-Aware Synchronization

As we saw, any solution for preserving the centralized concurrency and synchronization semantics in a distributed environment must deal with two issues: each remote method call can be executed on a new thread, and standard monitor methods such as `Object.wait`, `Object.notify`, and `synchronized` blocks can become invalid

when distribution takes place. Taking these issues into account, we maintain per-site "thread id equivalence classes," which are updated as execution crosses the network boundary; and at the bytecode level, we replace all the standard synchronization constructs with the corresponding method calls to a per-site synchronization library. This synchronization library emulates the behavior of the monitor methods, such as `monitorenter`, `monitorexit`, `Object.wait`, `Object.notify`, and `Object.notifyAll`, by using the thread id equivalence classes. Furthermore, these synchronization library methods, unlike the `final` methods in class `Object` that they replace, get correctly propagated over the network using RMI when necessary so that they execute on the network site of the object associated with the monitor.

In more detail, our approach consists of the following steps:

- Every instance of a monitor operation in the bytecode of the application is replaced, using bytecode rewriting, by a call to our own synchronization library, which emulates the monitor-style synchronization primitives of Java
- Our library operations check whether the target of the monitor operation is a local object or an RMI stub. In the former case, the library calls its local monitor operation. In the latter case, an RMI call to a remote site is used to invoke the appropriate library operation on that site. This solves the problem of propagating monitor operations over the network. We also apply a compile-time optimization to this step: using a simple static analysis, we determine whether the target of the monitor operation is an object that is known statically to be on the current site. This is the case for monitor operations on the `this` reference, as well as other objects of "anchored" types [17] that J-Orchestra guarantees will be on the same site throughout the execution. If we know statically that the object is local, we avoid the runtime test and instead call a local synchronization operation.
- Every remote RMI call, whether on a synchronized method or not, is extended to include an extra parameter. The instrumentation of remote calls is done by bytecode transformation of the RMI stub classes. The extra parameter holds the thread equivalence class for the current calling thread. Our library operations emulate the Java synchronization primitives but do not use the current, machine-specific thread id to identify a thread. Instead, a mapping is kept between threads and their equivalence classes and two threads are considered the same if they map to the same equivalence class. Since an equivalence class can be represented by any of its members, our current representation of equivalence classes is compact: we keep a combination of the first thread id to join the equivalence class and an id for the machine where this thread runs. This approach solves the problem of maintaining thread identity over the network.

We illustrate the above steps with examples that show how they solve each of the two problems identified earlier. We first examine the problem of propagating monitor operations over the network. Consider a method as follows:

```
//original code
void foo (Object some_remote_object) {
 this.wait();
 ...
 some_remote_object.notify();
 ...
}
```

At the bytecode level, method `foo` will have a body that looks like:

```
//bytecode
aload_0
invokevirtual     java.lang.Object.wait
...
aload_1
invokevirtual     java.lang.Object.notify
...
```

Our rewrite will statically detect that the first monitor operation (`wait`) is local, as it is called on the current object itself (`this`). The second monitor operation, however, is (potentially) remote and needs to be redirected to target machine using an RMI call. The result is shown below:

```
//rewritten bytecode
aload_0
//dispatched locally
invokestatic jorchestra.runtime.distthreads.wait_
...
aload_1
//get thread equivalence info from runtime
invokestatic jorchestra.runtime.ThreadInfo.getThreadEqClass
//dispatched through RMI;
//all remote interfaces extend DistSyncSupporter
invokeinterface jorchestra.lang.DistSynchSupporter.notify_
...
```

(The last instruction is an interface call, which implies that each remote object needs to support monitor methods, such as `notify_`. This may seem to result in code bloat at first, but our transformation adds these methods to the topmost class of each inheritance hierarchy in an application, thus minimizing the space overhead.)

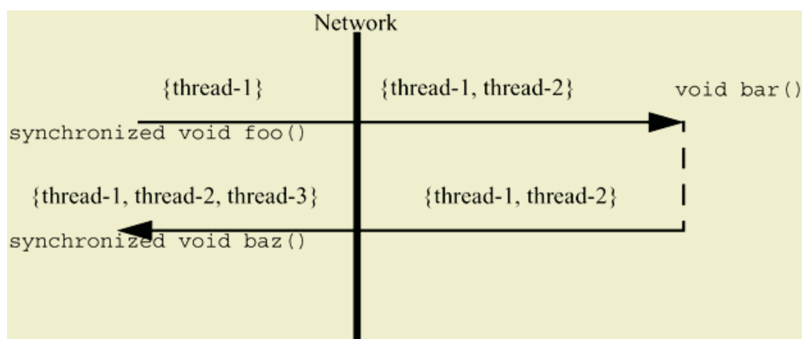

Fig. 2. Using thread id equivalence classes to solve the "zigzag deadlock problem" in Java RMI.

Let's now consider the second problem: maintaining thread identity over the network. Fig. 2 demonstrates how using the thread id equivalence classes can solve the "zigzag deadlock problem" presented above. These thread id equivalence classes enable our custom monitor operations to treat all threads within the same equivalence

class as the same thread. (We illustrate the equivalence class by listing all its members in the figure, but, as mentioned earlier, in the actual implementation only a single token that identifies the equivalence class is passed across the network.) More specifically, our synchronization library is currently implemented using regular Java mutexes and condition variables. For instance, the following code segment (slightly simplified) shows how the library emulates the behavior of the bytecode instruction `monitorenter`. (For readers familiar with monitor-style concurrent programming, our implementation should look straightforward.) The functionality is split into two methods: the static method `monitorenter` finds or creates the corresponding `Monitor` object associated with a given object: our library keeps its own mapping between objects and their monitors. The member method `enter` of class `Monitor` causes threads that are not in the equivalence class of the holder thread to wait until the monitor is unlocked.

```
public static void monitorenter (Object o) {
 Monitor this_m = null;
 synchronized (Monitor.class) {
    this_m = (Monitor)_objectToMonitor.get(o);
    if (this_m == null) {
       this_m = new Monitor();
       _objectToMonitor.put(o, this_m);
    }
 } //synchronized
 this_m.enter();
}

private synchronized void enter () {
 while (_timesLocked != 0 &&
        curThreadEqClass != _holderThreadId)
   try { wait(); } catch(InterruptedException e) {...}

 if (_timesLocked == 0) {
    _holderThreadId = getThreadID();
 }
 _timesLocked++;
}
```

The complexity of maintaining thread equivalence classes determines the overall efficiency of the solution. The key to efficiency is to update the thread equivalence classes only when necessary – that is, when the execution of a program crosses the network boundary. Adding the logic for updating equivalence classes at the beginning of every remote method is not the appropriate solution: in many instances, remote methods can be invoked locally within the same JVM. In these cases, adding any additional code for maintaining equivalence classes to the remote methods themselves would be unnecessary and detrimental to performance. In contrast, our solution is based on the following observation: the program execution will cross the network boundary only after it enters a method in an RMI stub. Thus, RMI stubs are the best location for updating the thread id equivalence classes on the client site of a remote call.

Adding custom logic to RMI stubs can be done by modifying the RMI compiler, but this would negate our goal of portability. Therefore, we use bytecode engineering

on standard RMI stubs to retrofit their bytecode so that they include the logic for updating the thread id equivalence classes. This is done completely transparently relative to the RMI runtime by adding special delegate methods that look like regular remote methods, as shown in the following code example. To ensure maximum efficiency, we pack the thread equivalence class representation into a long integer, in which the less significant and the most significant 4 bytes store the first thread id to join the equivalence class and the machine where this thread runs, respectively. This compact representation significantly reduces the overhead imposed on the remote method calls, as we demonstrate later on. Although all the changes are applied to the bytecode directly, we use source code for ease of exposition.

```
//Original RMI stub: two remote methods foo and bar
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {...}
 public int bar () throws RemoteException {...}
}

//Retrofitted RMI stub
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {
   foo__tec (Runtime.getThreadEqClass(), i);
 }
 public void foo__tec (long tec, int i) throws
                                       RemoteException
 {...}

 public int bar () throws RemoteException {
   return bar__tec (Runtime.getThreadEqClass());
 }
 public int bar__tec (long tec) throws RemoteException {...}
}
```

Remote classes on the callee site provide symmetrical delegate methods that update the thread id equivalence classes information according to the received `long` parameter, prior to calling the actual methods. Therefore, having two different versions for each remote method (with the delegate method calling the actual one) makes the change transparent to the rest of the application: neither the caller of a remote method nor its implementor need to be aware of the extra parameter. Remote methods can still be invoked directly (i.e., not through RMI but from code on the same network site) and in this case they do not incur any overhead associated with maintaining the thread equivalence information.

4 Benefits of the Approach

The two main existing approaches to the problem of maintaining the centralized Java concurrency and synchronization semantics in a distributed environment have involved either using custom middleware [5] or making universal changes to the distributed program [18]. We argue next that our technique is more portable than using custom middleware and more efficient than a universal rewrite of the distributed pro-

gram. Finally, we quantify the overhead of our approach and show that our implementation is indeed very efficient.

4.1 Portability

A solution for preserving the centralized concurrency and synchronization semantics in a distributed environment is only as useful as it is portable. A solution is portable if it applies to different versions of the same middleware (e.g., past and future versions of Java RMI) and to different middleware mechanisms such as CORBA and .NET Remoting. Our approach is both simple and portable to other middleware mechanisms, because it is completely orthogonal to other middleware functionality: We rely on bytecode engineering, which allows transformations without source code access, and on adding a small set of runtime classes to each network node of a distributed application. The key to our transformation is the existence of a client stub that redirects local calls to a remote site. Using client stubs is an almost universal technique in modern middleware mechanisms. Even in the case when these stubs are generated dynamically, our technique is applicable, as long as it is employed at class load time.

For example, our bytecode instrumentation can operate on CORBA stubs as well as it does on RMI ones. Our stub transformations simply consist of adding delegate methods (one for each client-accessible remote method) taking an extra thread equivalence parameter. Thus, no matter how complex the logic of the stub methods is, we would apply to them the same simple set of transformations.

Some middleware mechanisms such as the first version of Java RMI also use server-side stubs (a.k.a. *skeletons*) that dispatch the actual methods. Instead of presenting complications, skeletons would even make our approach easier. The skeleton methods are perfect for performing our server-side transformations, as we can take advantage of the fact that the program execution has certainly crossed the network boundary if it entered a method in a skeleton. Furthermore, having skeletons to operate on would eliminate the need to change the bytecodes of the remote classes. Finally, the same argument of the simplicity of our stub transformations being independent of the complexity of the stub code itself equally applies to the skeleton transformations.

In a sense, our approach can be seen as adding an orthogonal piece of functionality (concurrency control) to existing distribution middleware. In this sense, one can argue that the technique has an aspect-oriented flavor.

4.2 The Cost of Universal Extra Arguments

Our approach eliminates both the runtime and the complexity overheads of the closest past techniques in the literature. Weyns, Truyen, and Verbaeten [18][19] have advocated the use of a bytecode transformation approach to correctly maintain thread identity over the network. Their technique is occasionally criticized as "incur[ring] great runtime overhead" [5]. The reason is that, since clients do not know whether a method they call is local or remote, every method in the application is extended with an extra argument – the current thread id – that it needs to propagate to its callees. Weyns et al. argue that the overhead is acceptable and present limited measurements where the overhead of maintaining distributed thread identity is around 3% of the total execution time. Below we present more representative measurements that put this cost at

between 5.5 and 12%. A second cost that has not been evaluated, however, is that of complexity: adding an extra parameter to all method calls is hard when some clients cannot be modified because, e.g., they are in native code form or access the method through reflection. In these cases a correct application of the Weyns et al. transformation would incur a lot of complexity. This complexity is eliminated with our approach.

It is clear that some run-time overhead will be incurred if an extra argument is added and propagated to every method in an application. To see the range of overhead, we wrote a simple micro-benchmark, where each method call performs one integer arithmetic operation, two comparisons and two (recursive) calls. Then we measured the overhead of adding one extra parameter to each method call. Table 1 shows the results of this benchmark. For methods with 1-5 integer arguments we measure their execution time with one extra reference argument propagated in all calls. As seen, the overhead varies unpredictably but ranges from 5.9 to 12.7%.

Table 1. Micro-benchmark overhead of method calls with one more argument.

#params	1 (base)	1+1	2+1	3+1	4+1	5+1
Execution time (sec) for 10^8 calls	1.945	2.059	2.238	2.523	2.691	2.916
Slowdown relative to previous	-	5.9%	8.7%	12.7%	6.7%	8.4%

Nevertheless, it is hard to get a representative view of this overhead from micro-benchmarks, especially when running under a just-in-time compilation model. Therefore, we concentrated on measuring the cost on realistic applications. As our macro-benchmarks, we used applications from the SPEC JVM benchmark suite. Of course, some of the applications we measured may not be multithreaded, but their method calling patterns should be representative of multithreaded applications, as well.

We used bytecode instrumentation to add an extra reference argument to all methods and measured the overhead of passing this extra parameter. In the process of instrumenting realistic applications, we discovered the complexity problems outlined earlier. The task of adding an extra parameter is only possible when all clients can be modified by the transformation. Nevertheless, all realistic Java applications present examples where clients will not be modifiable. An application class can be implementing a system interface, making native Java system code a potential client of the class's methods. For instance, using class frameworks, such as AWT, Swing, or Applets, entails extending the classes provided by such frameworks and overriding some methods with the goal of customizing the application's behavior. Consider, for example, a system interface `java.awt.TextListener`, which has a single method `void textValueChanged (TextEvent e)`. A non-abstract application class extending this interface has to provide an implementation of this method. It is impossible to add an extra parameter to the method `textValueChanged` since it would prevent the class from being used with AWT. Similarly a Java applet overrides methods `init`, `start`, and `stop` that are called by Web browsers hosting the applet. Adding an extra argument to these methods in an applet would invalidate it. These issues can be addressed by careful analysis of the application and potentially maintaining two inter-

faces (one original, one extended with an extra parameter). Nevertheless, this would result in code bloat, which could further hinder performance.

Since we were only interested in quantifying the potential overhead of adding and maintaining an extra method parameter, we sidestepped the complexity problems by avoiding the extra parameter for methods that could be potentially called by native code clients. Instead of changing the signatures of such methods so that they would take an extra parameter, we created the extra argument as a local variable that was passed to all the callees of the method. The local variable is never initialized to a useful value, so no artificial overhead is added by this approach. This means that our measurements are slightly conservative: we do not really measure the cost of correctly maintaining an extra thread identity argument but instead conservatively estimate the cost of passing one extra reference parameter around. Maintaining the correct value of this reference parameter, however, may require some extra code or interface duplication, which may make performance slightly worse than what we measured.

Another complication concerns the use of Java reflection for invoking methods, which makes adding an extra argument to such methods impossible. In fact, we could not correctly instrument all the applications in the SPEC JVM suite, exactly because some of them use reflection heavily and we would need to modify such uses by hand.

The results of our measurements appear in Table 2. The table shows total execution time for four benchmarks (compress – a compression utility, javac – the Java compiler, mtrt – a multithreaded ray-tracer, and jess – an expert system) in both the original and instrumented versions, as well as the slowdown expressed as the percentage of the differences between the two versions, ranging between 5.5 and 12%. The measurements were on a 600MHz Pentium III laptop, running JDK 1.4.

Table 2. Macro-Benchmarks.

Benchmark	compress	javac	mtrt	jess
Original version (sec)	22.403	19.74	6.82	8.55
Instrumented version (sec)	23.644	21.18	7.49	9.58
Slowdown	5.54%	7.31%	9.85%	12.05%

The best way to interpret these results is as the overhead of pure computation (without communication) that these programs would incur under the Weyns et al. technique if they were to be partitioned with J-Orchestra so that their parts would run correctly on distinct machines. We see, for instance, that running jess over a network would incur an overhead of 12% in extra computation, just to ensure the correctness of the execution under multiple threads. Our approach eliminates this overhead completely: overhead is only incurred when actual communication over distinct address spaces takes place. As we show next, this overhead is minuscule, even when no network communication takes place.

4.3 Maintaining Thread Equivalence Classes Is Cheap

Maintaining thread equivalence classes, which consists of obtaining, propagating, and updating them, constitutes the overhead of our approach. In other words, to maintain the thread equivalence classes correctly, each retrofitted remote method invocation

includes one extra local method call on the client side to obtain the current class, an extra argument to propagate it over the network, and another local method call on the server side to update it. The two extra local calls, which obtain and update thread equivalence classes, incur virtually no overhead, having a hash table lookup as their most expensive operation and causing no network communication. Thus, the cost of propagating the thread equivalence class as an extra argument in each remote method call constitutes the bulk of our overhead.

In order to minimize this overhead, we experimented with different thread equivalence classes' representations. We performed preliminary experiments which showed that the representation does matter: the cost of passing an extra reference argument (any subclass of `java.lang.Object` in Java) over RMI can be high, resulting in as much as 50% slowdown in the worst case. This happens because RMI accomplishes the marshalling/unmarshalling of reference parameters via Java serialization, which involves dynamic memory allocation and the use of reflection. Such measurements led us to implement the packed representation of thread equivalence class information into a long integer, as described earlier. A `long` is a primitive type in Java, hence the additional cost of passing one over the network became negligible.

To quantify the overall worst-case overhead of our approach, we ran several microbenchmarks, measuring total execution time taken by invoking empty remote methods with zero, one `java.lang.String`, and two `java.lang.String` parameters. Each remote method call was performed 10^6 times. The base line shows the numbers for regular uninstrumented RMI calls. To measure the pure overhead of our approach, we used an unrealistic setting of collocating the client and the server on the same machine, thus eliminating all the costs of network communication. The measurements were on a 2386MHz Pentium IV, running JDK 1.4. The results of our measurements appear in Table 3.

Table 3. Overhead of Maintaining Thread Equivalence Classes.

No. of Params	Base Line (ms)	Maintaining Thread Equivalence Classes (ms)	Overhead (%)
0	145,328	150,937	3.86%
1	164,141	166,219	1.27%
2	167,984	168,844	0.51%

Since the remote methods in this benchmark did not perform any operations, the numbers show the time spent exclusively on invoking the methods. While the overhead is approaching 4% for the remote method without any parameters, it diminishes gradually to half a percent for the method taking two parameters. Of course, our settings for this benchmark are strictly worst-case – had the client and the server been separated by a network or had the remote methods performed any operations, the overhead would strictly decrease.

5 Discussion

As we mentioned briefly earlier, our distributed synchronization technique only supports monitor-style concurrency control. This is a standard application-level concur-

rency control facility in Java, but it is not the only one and the language is actively evolving to better support other models. For example, high-performance applications may use `volatile` variables instead of explicit locking. In fact, use of non-monitor-style synchronization in Java will probably become more popular in the future. The upcoming JSR-166 specification will standardize many concurrent data structures and atomic operations. Although our technique does not support all the tools for managing concurrency in the Java language, this is not so much a shortcoming as it is a reasonable design choice. Low-level concurrency mechanisms (volatile variables and their derivatives) are useful for synchronization in a single memory space. Their purpose is to achieve optimized performance for symmetric multiprocessor machines. In contrast, our approach deals with correct synchronization over middleware – i.e., it explicitly addresses distributed memory. The technique we presented in this paper is likely to be employed in a cluster or even a more loosely coupled network of machines. In this setting, monitor-style synchronization makes perfect sense.

On the other hand, in the future we can use the lower-level Java concurrency control mechanisms to optimize our own library for emulating Java monitors. As we saw in Section 3, our current library is itself implemented using monitor-style programming (`synchronized` blocks, `Object.wait`, etc.). With the use of optimized low-level implementation techniques, we can gain in efficiency. We believe it is unlikely, however, that such a low-level optimization in our library primitives will make a difference for most client applications of our distributed synchronization approach.

Finally, we should mention that our current implementation does not handle all the nuances of Java monitor-style synchronization, but the issue is one of straightforward engineering. Notably, we do not currently propagate `Thread.interrupt` calls to all the nodes that might have threads blocked in an invocation of the `wait` method. Even though it is not clear that the `interrupt` functionality is useful for distributed threads, our design can easily support it. We can replace all the calls to `Thread.interrupt` with calls to our synchronization library, which will obtain the equivalence class of the interrupted thread and then broadcast it to all the nodes of the application. The node (there can be only one) that has a thread in the equivalence class executing the `wait` operation of our library will then stop waiting and the operation will throw the `InterruptedException`.

6 Related Work

The technique described in this paper was applied in the context of J-Orchestra, but can be re-used in multiple different contexts. For instance, our implementation of correct synchronization can be combined with a mechanism for capturing and migrating threads, as in the Brakes system [18]. So far, we have not explored thread migration at all in the context of J-Orchestra. In fact, J-Orchestra prohibits the migration of all objects that can be potentially passed to native code (based on a heuristic analysis) in an effort to ensure the correctness of the resulting partitioned application. Thus, thread objects explicitly cannot be mobile in a J-Orchestra-partitioned application: a thread always executes on the site where it was created. This fits well the requirements of the system, i.e., to ensure correct distributed execution even for completely unsuspecting centralized Java applications.

Several projects have concentrated on offering distributed capabilities for Java applications. Examples include CJVM [1] and Java/DSM [21]. Nevertheless, most of

these approaches are in the Distributed Shared Memory (DSM) space, instead of in explicit middleware support for synchronization. As a result, communication is not explicit and the programming model is one of shared memory with relaxed consistency semantics, instead of one of communicating distributed objects. Furthermore, DSMs do not support the portability and ease of deployment goal of our technique. An application deployed on a DSM will require a specialized run-time system and system library. Our distributed synchronization approach adds correct synchronization handling on top of traditional Java middleware.

Our technique has wide applicability in the space of automatic partitioning systems. Several such systems have been proposed recently, including Addistant [16], AdJava [3], FarGo [6], and Pangaea [13][14]. To our knowledge, J-Orchestra is the first automatic partitioning system to concentrate on the correctness of partitioning multithreaded programs. It is worth noting that a partitioning system cannot just transparently inherit distributed synchronization capabilities from its underlying middleware. Even with specialized middleware that tries to handle distributed synchronization issues (e.g., KaRMI [10][12]), a partitioning system will need to transform an application so that Java monitor commands, like `monitorenter`, are propagated to a remote site.

Language tools for distribution, such as JavaParty [4][11], can also benefit from our technique. JavaParty already supports distributed synchronization [5] through a combination of language-level transformation (to intercept monitor actions) and specialized middleware [12]. As discussed earlier, our approach enables the same functionality over standard Java middleware, such as RMI.

Our technique also complements work on offering versions of RMI optimized for parallel computing on cluster machines [8][9]. In a parallel computing setting, it is advantageous to have distributed synchronization as a technique that can be transparently added to a middleware implementation.

Finally, there are approaches to richer middleware that would simplify the implementation of our technique. For instance, DADO [20] enables passing custom information between client and server of a remote call. This would be an easy alternative to our custom bytecode transformations of stubs and skeletons. Nevertheless, using DADO would not eliminate the need for bytecode transformations that replace monitor control methods and synchronization blocks.

7 Conclusions

In this paper we presented a technique for correct monitor-style synchronization of distributed programs in Java. Our technique addresses the lack of coordination between Java concurrency mechanisms and Java middleware. We argue that the technique is important because it comprehensively solves the problem and combines the best features of past approaches by offering both portability and efficiency. Furthermore, we believe that the value of our technique will strictly increase in the future. With the increased network connectivity of all computing resources, we expect a need for distributed programming models that look more like centralized programming models. In such a setting, the need for correct and efficient distributed synchronization will become even greater.

Acknowledgments. This research was supported by the NSF through grants CCR-0238289 and CCR-0220248, and by the Georgia Electronic Design Center.

References

1. Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster", in Proc. *ICPP'99*.
2. Markus Dahm, "Doorastha – a step towards distribution transparency", *JIT,* 2000. See `http://www.inf.fu-berlin.de/~dahm/doorastha/`.
3. Mohammad M. Fuad and Michael J. Oudshoorn, "AdJava – Automatic Distribution of Java Applications", 25th *Australasian Computer Science Conference (ACSC)*, 2002.
4. Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, "JavaParty: A distributed companion to Java", `http://wwwipd.ira.uka.de/JavaParty/`
5. Bernhard Haumacher, Thomas Moschny, Jürgen Reuter, and Walter F. Tichy, "Transparent Distributed Threads for Java", *5th International Workshop on Java for Parallel and Distributed Computing*, in conjunction with the *International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.
6. Ophir Holder, Israel Ben-Shaul, and Hovav Gazit, "Dynamic Layout of Distributed Applications in FarGo", *Int. Conf. on Softw. Engineering (ICSE)* 1999.
7. Doug Lea, "Concurrent Programming in Java -- Design Principles and Patterns", Addison-Wesley, Reading, Mass., 1996.
8. Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Aske Plaat, "An Efficient Implementation of Java's Remote Method Invocation", *Proc. of ACM Symposium on Principles and Practice of Parallel Programming, Atlanta, GA May 1999.*
9. Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman, "Efficient Java RMI for Parallel Programming", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747-775, November 2001.
10. Christian Nester, Michael Phillipsen, and Bernhard Haumacher, "A More Efficient RMI for Java", in Proc. *ACM Java Grande Conference*, 1999.
11. Michael Philippsen and Matthias Zenger, "JavaParty - Transparent Remote Objects in Java", *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.
12. Michael Philippsen, Bernhard Haumacher, and Christian Nester, "More Efficient Serialization and RMI for Java", *Concurrency: Practice & Experience*, 12(7):495-518, May 2000.
13. Andre Spiegel, "Pangaea: An Automatic Distribution Front-End for Java", 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, San Juan, Puerto Rico, April 1999.
14. Andre Spiegel, "Automatic Distribution in Pangaea", *CBS 2000*, Berlin, April 2000. See also `http://www.inf.fu-berlin.de/~spiegel/pangaea/`
15. Sun Microsystems, Remote Method Invocation Specification, `http://java.sun.com/products/jdk/rmi/`, 1997.
16. Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software", *European Conference on Object-Oriented Programming (ECOOP)*, Budapest, June 2001.
17. Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002.
18. Danny Weyns, Eddy Truyen, and Pierre Verbaeten, "Distributed Threads in Java", *International Symposium on Distributed and Parallel Computing (ISDPC)*, July 2002.
19. Danny Weyns, Eddy Truyen and Pierre Verbaeten, "Serialization of Distributed Threads in Java", *Special Issue of the International Journal on Parallel and Distributed Computing Practice, PDCP* (to appear).
20. Eric Wohlstadter, Stoney Jackson and Premkumar Devanbu, "DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems", *International Conference on Software Engineering (ICSE)*, 2003.
21. Weimin Yu, and Alan Cox, "Java/DSM: A Platform for Heterogeneous Computing", *Concurrency: Practice and Experience*, 9(11):1213-1224, 1997.