Implementation of Strong Mobility for Multi-Threaded Agents in Java

Arjav J. Chakravarti Xiaojin Wang Jason O. Hallstrom Gerald Baumgartner
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210, USA
{arjav,hallstro,gb}@cis.ohio-state.edu frozen_wang@hotmail.com

Abstract

Strong mobility, which allows multi-threaded agents to be migrated transparently at any time, is a powerful mechanism for implementing a peer-to-peer computing environment, in which agents carrying a computational payload find available computing resources. Existing approaches to strong mobility either modify the Java Virtual Machine or do not correctly preserve the Java semantics when migrating multi-threaded agents.

We give an overview of our implementation strategy for strong mobility in which each agent thread maintains its own serializable execution state at all times, while thread states are captured just before a move. We explain how to solve the synchronization problems involved in migrating a multi-threaded agent and how to cleanly terminate the Java threads in the originating virtual machine. We present experimental results that indicate that our implementation approach is feasible in practice.

1 Introduction

The advent of Grid Computing [9] has improved the reliable utilization of shared computational resources for the solution of complex problems, such as for [13]. Peer-to-Peer systems adopt a highly decentralized, though less reliable, approach to resource sharing, and are mainly used for embarrassingly parallel applications like [27] and simple applications like file-sharing [11]. A confluence of these two technologies will facilitate the building of flexible systems to support dynamic communities of users [15, 8].

The vast majority of distributed applications are currently built with distributed object technologies, such as Java RMI, CORBA, COM, or SOAP. These RPC-based approaches do not, however, consider the execution state of their arguments. If a thread is active within one of the arguments passed to a remote procedure, it does not travel along with the argument.

The Mobile Agent abstraction is the movement of code, data and threads from one location to another [19]. In the peer-to-peer applications envisaged for Grid systems, mobile agents offer a flexible means of distributing data and code around a network, of dynamically moving between hosts as resources become available, and of carrying multiple threads of execution to simultaneously perform computation, the scheduling of other agents, and communication with other agents on a network. Approaches to using mobile agents for Grid Computing have been discussed in [4, 24, 22].

Java is the language of choice for an overwhelming majority of the mobile agent systems that have been developed until now. However, the execution model of the Java Virtual Machine does not permit an agent to access the run-time stack and program counter.

A ramification of this constraint is that Java based mobility libraries can only provide *weak mobility* [6]. Weakly mobile agent systems, such as IBM's Aglets framework [20] do not migrate the execution state of methods. The `go()` method, used to move an agent from one virtual machine to another, simply does not return. The agent environment kills the threads currently executing in the agent, without saving their state. The lifeless agent is then shipped to its destination and is resurrected there. Weak mobility forces programmers to use a difficult programming style, i.e., the use of callback methods, to account for the absence of migration transparency.

By contrast, agent systems with *strong mobility* provide the abstraction that the execution of the agent is uninterrupted, even as its location changes. Applications that require agents to migrate from host to host while communicating with one another to solve a problem, are severely restricted by the absence of strong mobility. The ability of a system to support the migration of an agent at any time by an external thread, is termed *forced mobility*. This is particularly useful for load-balancing, and for fault-tolerant applications, and is difficult to implement without strong mobility. Strong mobility also allows programmers to use a far more natural programming style.

A number of different approaches have been followed to

add strong mobility to Java. These can be separated into two broad categories - those that use modified or custom VMs, and those that change the compilation model.

JavaThread[3], D'Agents[12], Sumatra[1], Merpati[28] and Ara[23], all depend on extensions to the standard VM from Sun, whereas the CIA[16] project uses a modification of the Java Platform Debugger Architecture. Forced mobility is not supported by JavaThread, CIA and Sumatra. In addition, JavaThread depends on the deprecated `stop()` method in `java.lang.Thread` to migrate an agent. The D'Agents, Sumatra, Ara and CIA systems do not migrate multi-threaded agents. Merpati does not migrate agent threads that are blocked in monitors. The NOMADS[29] project uses a custom virtual machine known as Aroma, to provide support for forced mobility and the migration of multi-threaded agents. Support for thread migration within a cluster is provided by JESSICA2[32]. The solution does not scale to the Internet or a Grid, however, because a distributed VM is used.

Modifying the Java VM, or using a custom VM, has the major disadvantage of a lack of portability. Existing virtual machines cannot be used. It is very difficult to maintain complete compatibility with the Sun Java specification. For example, JavaThread and NOMADS are JDK 1.2.2 compatible, D'Agents relies on a modified Java 1.0 VM, and Merpati and Sumatra are no longer supported. It is also difficult to achieve the performance of the JVM from Sun. NOMADS, Sumatra and Merpati do not support JIT compilation. In addition, some users may prefer to use other VMs of their choice. These problems greatly impact the acceptability and widespread use of mobile agent systems that rely on VM modifications.

Another approach to adding strong mobility to Java is to change the compilation model (by using a preprocessor, by modifying the compiler, or by modifying the generated bytecode) such that the execution state of an agent can be captured before migration.

This approach is followed by WASP[10] and JavaGo[26]. These use a source code preprocessor. However, neither supports forced mobility. In addition, JavaGo does not migrate multiple threads of execution or preserve locks on migration. Correlate[30] and JavaGoX[25] modify bytecode. Migration may only be initiated by the agent itself, i.e., forced mobility is not supported.

We have chosen to provide strong mobility for Java by using a preprocessor to translate strongly mobile source code to weakly mobile source code [5, 31]. We present an overview of our implementation approach, in which an agent maintains a movable execution state for each thread at all times. The generated weakly mobile code saves the state of a computation before moving an agent so that the state can be recovered once the agent arrives at the destination. The code translation could be done at the level of bytecode as well. The translation of method calls requires type information, however, and this would involve decompiling bytecode. To avoid this, we use the more convenient source translation mechanism.

Jiang and Chaudhary [18, 17] use a similar approach for C and C++. The scalability of their system is limited by its dependence on a global scheduler to migrate threads. It is also unclear whether they can handle multiple concurrent migrations, which could impact performance. Bettini and De Nicola [2] also use the same idea for agent migration, but they do this for a toy language. Our implementation is designed for the full Java programming language.

2 Overview

Our implementation approach for strong mobility in Java is to translate strongly mobile code into weakly mobile code. We currently target the IBM Aglets weak mobility system.

Every method in the original agent class is translated to a `Serializable` inner class which represents the activation record for that method. The local variables, parameters and program counter are converted to fields of this class. This inner class contains a `run()` method to represent the body of the original method. The generated weakly mobile agent class contains an array of activation record objects that acts as a virtual method table.

Threads in Java are not `Serializable` because they use native code. However, the state of every thread of execution also needs to be maintained so that the thread can be restarted at the destination. This is achieved by using a `Serializable` wrapper around each Java `Thread`. This wrapper contains its own stack of activation records that mirrors the run-time stack of the underlying thread of execution. When a method is called, the appropriate entry from the method table is cloned and put on the stack. Its `run()` method is then executed.

Statement execution and the program counter update should be executed atomically to allow an agent to be moved at arbitrary points of time. The original source code is translated to a form that allows the state of the agent to be saved for each executed statement, while maintaining the semantics of these statements. Translation rules for the different types of statements in the Java language are required.

A *go()* method is called on a multi-threaded agent to send it to a new location. The `Serializable` wrappers then bring the agent threads to a standstill and save the state of these threads. The agent then relocates and carries along with it only the `Serializable` wrappers of the threads. These wrappers create new `Thread` objects at the destination and recreate their execution state. Potentially long-running operations like `Object.wait(long)` are inter-

rupted and the time left for them to finish execution is saved before the move.

The use of multi-threaded agents makes synchronization issues very important. For a multi-threaded system, the program counter must be incremented atomically with the following instruction; two agents must not dispatch one another at the same time, and two threads within the same agent must not dispatch the agent simultaneously. User-specified `synchronized` blocks in the original Java source code also need to be translated so that they can be carried along by an agent. Synchronization control in mobile agents is non-trivial, but we offer an approach that is no more taxing than programming for a traditional non-mobile system.

Each statement and its corresponding program counter update are wrapped inside a logical synchronized block to preserve their atomicity and prevent agent relocation before their completion. It is unacceptable for the implementation to synchronize on the agent instance because that would prevent threads from executing translated statements in parallel. The problem is a basic *readers/writers conflict*, where the threads that execute the translated statements are readers, and the thread that executes the `go()` method acts as a writer. A *writers priority* solution is used. Each agent maintains *locks* that represent the predicate, 'OK to execute statements?'. The number of locks is the same as the number of reader threads, and are acquired and released by readers before and after statement execution. When a call is made to `go()`, the writer thread acquires all the locks, saves the agent state and moves the agent to a new site.

The call to `go()` is synchronized on the agent context instead of on the caller, in order to prevent deadlock when two agents call one another. Similarly, if multiple threads within the same agent attempt to move the agent, deadlock is prevented by having each thread test a synchronized condition variable in the agent context. The first writer thread will set this variable, and the subsequent writers will test the variable and then give up their locks.

Threads must acquire and release a lock on a particular object on entering and leaving a `synchronized` block. If an agent moves when a thread is executing inside this protected region, the lock held by the thread is released. Protection is extended across machine boundaries by introducing serializable locks in place of standard Java locks, for every object that is synchronized upon. `synchronized` blocks are often used in conjunction with the `wait()` and `notify()` operations. These too, are appropriately translated to preserve their semantics.

We have run a number of benchmarks to test our translator for strong mobility. A comparison of the performance of the translated agents and the corresponding IBM Aglets has been performed. Some simple optimizations to the generated code were performed by hand, and the performance enhancement was observed. The measurements confirm the feasibility of our approach.

3 Language and API Design

Our support for strong mobility consists currently of the interface `Mobile` and the two classes `MobileObject` and `ContextInfo`.

3.1 Interface Mobile

Every mobile agent must (directly or indirectly) implement the interface `Mobile`. A client of an agent must access the agent through a variable of type `Mobile` or a subtype of `Mobile`.

Interface `Mobile` is defined as follows:

```
public interface Mobile extends
                java.io.Serializable {
  public void go(java.net.URL dest)
    throws com.ibm.aglet.RequestRefusedException,
    edu.ohio_state.cis.brew.MoveRefusedException,
    java.io.IOException; ... }
```

Like `Serializable`, interface `Mobile` is a *marker interface*. It indicates to a compiler or preprocessor that special code might have to be generated for any class implementing this interface. `go()` moves the agent to the destination with the URL `dest`. This method can be called either from a client of the agent or from within the agent itself. The second parameter indicates whether the call was made from within the agent or from outside.

3.2 Class MobileObject

Class `MobileObject` implements interface `Mobile` and provides the two methods `getContextInfo()` and `go()`. To allow programmers to override these methods, they are implemented as wrappers around `native` implementations that are translated into weakly mobile versions. A mobile agent class is defined by extending class `MobileObject`.

The method `getContextInfo()` provides any information about the context in which the agent is currently running.

3.3 Class ContextInfo

Class `ContextInfo` is used for an agent to access any resources on the machine it is currently running on, including any system objects that the host wants to make accessible to a mobile agent.

Currently, we only provide a method `getHostURL()`, that returns the URL of the agent environment in which the agent is running. We will extend the functionality of class `ContextInfo` in future translator versions.

For providing access to special-purpose resources such as databases, an agent environment can implement the method `getContextInfo()` to return an object of a subclass of class `ContextInfo`.

3.4 Strongly Mobile User Code

For writing a mobile agent, the programmer must first define an interface, say `Agent`, for it. This interface should extend interface `Mobile` and declare any additional methods. All additional methods must be declared to throw `AgletException`. An implementation of the mobile agent then extends class `MobileObject` and implements interface `Agent`. A client of the agent must access the agent through a variable of the interface type `Agent` and through a proxy object similar as in Java RMI or in Aglets.

4 Translation from Strong to Weak Mobility

In this section, we present the translation mechanism for methods, classes, statements, and exceptions.

4.1 Translation of Methods

For each agent method, the preprocessor generates a class whose instances represent the activation records for that method. As multiple invocations may be active simultaneously (e.g., recursive methods), these objects are cloneable. An activation record class for a method is a subclass of the abstract class `Frame`.

```
public void foo(int x) throws AgletsException {
  int y;
  // blocks of statements
  BC1
  BC2 }
```

The parameter `x`, local variable `y` and the program counter become fields of class `Foo`. A `setPCForMove()` method is necessary to allow the arbitrary suspension and movement of a thread of execution. This method saves the current `programCounter`, before setting it to -1 to ensure that no further instructions get executed before the agent moves. The `run()` method contains the translated version of the body of `foo()`, which includes code for incrementing the program counter, as well as code which allows `run()` to resume computation after a move. Every thread needs to poll whether it is time to move or not. It does this by acquiring and releasing a lock before and after every logical statement in the code. This is done by the `AgentImpl.this.request_read()` and `AgentImpl.this.read_accomplished()` calls. The generated activation record class for `foo` is:

```
protected class Foo extends Frame {
  int x, y, progCounter = 0; Object trgt;
  void setPCForMove() { ... }
  void run() {
    try { ...
      AgentImpl.this.request_read();
      if ((progCounter == 0)) {
        progCounter+=1; BC1 }
      AgentImpl.this.read_accomplished();
      AgentImpl.this.request_read();
      if ((progCounter == 1)) {
        progCounter+=1; BC2 }
      AgentImpl.this.read_accomplished(); }
    catch(AgletsException e) { ... }} ... }
```

4.2 Translation of Agent Classes

The generated agent class contains an array of `Frame` objects that is used as a virtual method table. When a method is called, the appropriate entry from the method table is cloned and put on the thread wrapper stack.

For example, suppose that we have an agent class `AgentImpl` of the form:

```
public class AgentImpl extends
       MobileObject implements Agent {
  int a; public AgentImpl() {/* init code */}
  public void foo(int x) throws AgletsException {
    BC; } }
```

Because this class (indirectly) implements the `Mobile` interface, the preprocessor translates it into the code described below:

The original agent method `foo()` gets translated into an inner class `Foo`. There are two `foo()` methods in the generated code, of which `foo(Object, Object)` is a preparatory method. Its first parameter is a reference to the wrapper of the thread on which the method is to be executed. An activation record is created and pushed onto the wrapper stack. The second parameter is an `Object` array that contains the arguments to the original `foo()` method. These are given to the activation record.

The second `foo()` method has the same order, type and number of parameters as the original untranslated method. All the calls to the original `foo` method from within the agent now go to this method. The method obtains a reference to the wrapper of the currently executing thread and packages its parameters in an `Object` array, before calling the `foo(Object, Object)` method described above. The activation record on the top of the stack is then executed.

```
public void foo(Object target, Object init){...}

public void foo(int x) { ...
  //fooThread - wrapper of current thread
  foo(fooThread,
       new Object[]{new Integer(x), ... });

  //method call to execute original method body
  fooThread.run1(); return; }
```

The Aglets system does not allow method invocations from outside the agent, only message sends. The `handleMessage()` method is an Aglets method that receives messages sent to the agent. If the `foo()` method in the untranslated agent could be invoked by an external thread, a new thread is *created* when a message is received for `foo()`. `foo(Object, Object)` is then called and the activation record on top of the stack is executed.

```
public boolean handleMessage(Message msg) {
  if (msg.sameKind("foo")) { ...
    // fooThread is the wrapper of the new thread
    foo(fooThread, msg.getArg());
    fooThread.start(); ... return true; } ... }
```

Our translator translates almost the entire Java language. Some portions of the translator have not been implemented completely due to time constraints. The mobility translator is a preprocessor to the Brew compiler. The compiler is still under development, and as yet does not do type-checking. For this reason, it needs to be hard-coded into the translator as to whether method calls and returns are to targets outside or within the agent. The translation of inner classes, `try` blocks, `labels`, and the `assert`, `break` and `continue` statements has not yet been implemented. Name-mangling to support nested blocks and overloaded methods, and the translation of method calls inside expressions also need to be completed. We believe that these issues are simple enough to be satisfactorily resolved.

5 Resource Access

When accessing global resources it is desirable to distinguish between global names on the current virtual machine and global names on the home platform of the agent. To allow agent developers to access platform-bound resources remotely, we introduce the `global` field declaration prefix. Use of the prefix indicates that a particular field should be created (and accessed) on the home platform. For example:

```
private global InputStream is;
```

For each field prefixed with the `global` keyword, the preprocessor generates code to register an RMI server with the home platform. Each RMI server is a simple wrapper, delegating calls to the original field instance, to which it maintains a reference. Special accessor methods are also provided by these servers to handle field assignment, scalar field access, and access to field members within a global field. These field servers are created and registered immediately after the agent is constructed. Any agent code that accesses the global field is translated to access the resource through the corresponding RMI proxy.

A similar problem arises when examining accesses to fields and methods which are both public and static. Consider, for example, an agent that wishes to roughly approximate the time it takes for it to move between two platforms. The agent needs to access the method `System.currentTimeMillis()` from the home platform.

To provide agent developers flexibility in specifying whether access to a static method or field refers to the home VM, we introduce syntax for retroactively making a static method or field global. By default, access to a static method or field will refer to the VM on which the agent currently resides. To indicate that the home VM should be used to perform the access, we use a retroactive `global` declaration as follows:

```
global long System.currentTimeMillis();
global PrintStream System.out;
```

The implementation of remote resource access has not yet been completed.

6 Multi-Threaded Agents

The multi-threading support provided by Java consists of the classes `Thread` and `ThreadGroup` and the interface `Runnable`, which allow us to create multiple threads of execution within the agent, and to manage groups of threads as a unit.

Java Threads are not serializable because they involve native code. The state of each thread needs to be saved in a serializable format that can then be relocated.

6.1 MobileThread and MobileThreadGroup

The serializable wrapper classes `MobileThread` and `MobileThreadGroup`, are used around the Java library classes `Thread` and `ThreadGroup`. When `MobileThread` and `MobileThreadGroup` objects are created, they create new `Thread` and `ThreadGroup` objects to perform the actual execution. `MobileThread` thus contains the information about its underlying thread that is needed to reconstruct the state of that thread after a move. `MobileThreadGroup` acts similarly with respect to `ThreadGroup`. Only the wrappers are moved when an agent moves to a new site. At the destination, these wrappers create new `Thread` and `ThreadGroup` objects and set their state so that execution can continue. Each `MobileThread` also belongs to a particular `MobileThreadGroup`, and when a `MobileThread` object recreates a thread of execution, that `Thread` is also assigned to the same `ThreadGroup` as at the source location.

The class `MobileThread` contains a `start()` method which is called to begin execution of a `MobileThread`. This can happen after it has been created for the first time or when the agent starts up all the threads after moving to a new site. This method calls the `start()` method of the underlying `Thread`, which then calls the `run()` method of its target, the `MobileThread` wrapper. The `run()` method checks the `MobileThread` stack. If the stack is empty, it means that the `MobileThread` is a newly created one, and has to call the `run()` method of its `Runnable` target. The `MobileThread's` stack not being empty means that the activation records already on the stack need to be executed.

The preprocessor translates the strongly mobile agent code to weakly mobile code, as explained in Section 4. Furthermore, the preprocessor replaces every occurrence of `Thread` and `ThreadGroup` in the original code with `MobileThread` and `MobileThreadGroup`. In this manner, every reference to a `Thread` or `ThreadGroup` object in the original code is now translated to a reference to a `MobileThread` or `MobileThreadGroup` object. We thus ensure that every original operation on a `Thread` or `ThreadGroup` in user code is now made to go through their wrappers.

The mobility translator translates every occurrence of the word `Thread` in the source code with the word `MobileThread`. This ensures that the calls to the methods of `Thread` go through the serializable wrapper, and that the `run()` method of a multi-threaded Agent now executes as activation records on the stack of the thread wrapper.

6.2 Static Methods of java.lang.Thread

When `MobileThread.currentThread()` is called, it calls `Thread.currentThread()` in turn. This returns a reference to the currently executing `Thread`. A reference to the `MobileThread` wrapper over this `Thread` object now needs to be returned. The solution is to maintain a static Hashtable that contains a mapping of `Thread`s to their corresponding `MobileThread`s. In this way, `MobileThread.currentThread()` returns the correct `MobileThread` object.

Similarly, the other static methods of `MobileThread` (`sleep()`, `enumerate()`, etc.) use `ThreadTable`, where necessary, to return the appropriate `MobileThread` references.

6.3 Relocating a Multi-threaded Agent

The `go()` method is called on a multi-threaded agent. This calls the `realGo()` method, which first checks whether this agent is already being moved or not. If the agent is being moved, a `MoveRefusedException` is thrown. Otherwise, the thread that wishes to move the agent acquires locks such that every `Thread` within the agent blocks and comes to a standstill. Each `MobileThread` makes an `interrupt()` call to its `Thread`, thus terminating any `wait()`, `join()` or `sleep()` operations. If any of these are timed, the time remaining for them to finish is saved such that they can be completed at the destination.

The `packUp()` method of the `main` agent threadgroup wrapper is called. From here, the `packUp()` method of each threadgroup and thread wrapper under `main` is called, and the state of its underlying threadgroup and thread saved. The system threads are then forced to terminate and the agent is relocated by using the Aglets `dispatch()` method. The `java.lang.Thread` `API` does not permit direct termination of a thread. Section 7 explains how we accomplish this.

On arrival at the destination, the `reinit()` method of the main threadgroup wrapper is called. This method then creates a new `ThreadGroup`, sets its state, and then calls the `reinit()` method of each threadgroup and thread wrapper under `main`. Each wrapper's `reinit()` method creates a new `Thread` or `ThreadGroup`, and sets its state. The `start()` method of the `main` threadgroup wrapper is called, resulting in calls to the `start()` methods of all `MobileThread`s to begin execution of their threads.

7 Synchronization Issues

There are three major issues that need to be handled correctly for the synchronization control of a multi-threaded agent - preserving the atomicity of a logical instruction; preventing deadlock when agents dispatch one another, or when multiple threads attempt to dispatch the agent; preserving the semantics of Java `synchronized` blocks across a migration.

7.1 Protection of Agent Stacks

An agent should not be moved while it is executing a statement. It is necessary to protect every program counter increment and its following statement. Synchronizing on the agent will reduce parallelism dramatically. The problem can be reduced to a basic *readers/writers conflict*, where the increment of the program counter, and the execution of the following statement by each thread, acts as a *reader*; the *writer* being the thread that calls `go()`. This problem is solved by using a variant of the solution in [14]. *Lock*s are maintained by each agent. The predicate they represent is 'OK to execute statements?'. The number of locks equals the number of executing threads within the agent. Reader threads acquire and release locks before and after executing logical statements, by `request_read()` and `read_accomplished()` calls.

```
AgentImpl.this.request_read(); if(pc==15) {pc++;
stmt;} AgentImpl.this.read_accomplished();
```

When a thread makes a call to `go()`, it is designated as a writer. The writer thread attempts to acquire all the agent locks. Once it makes this attempt, no reader can acquire a lock. The writer then calls `interruptForMove()` on all currently executing `MobileThread`s. If a thread is carrying out a `wait()`, `join()` or `sleep()`, the wrapper repeatedly interrupts it until it stops the operation. The method whose execution was interrupted, checks whether the wrapper interrupted the thread. If so, the program counter is decremented so that the interrupted operation will resume at the destination. If the interrupted operation was timed, like `wait(long)`, the time remaining for the operation to complete is saved by the `MobileThread`. An `InterruptedException` is thrown if the wrapper did not cause the interrupt. A count of the number of currently active readers is maintained. This count is incremented when a reader requests the lock by calling `request_read()`, and decremented when the lock is released by a `read_accomplished()` call. As the readers only relinquish their locks at this stage, depending on whether the writer is an internal or an external thread, the count must go down to one or zero. The writer then calls the `packUp()` method of the `main` threadgroup wrapper. This results in each `MobileThread` saving the state of its `Thread`, setting the program counter of the activation records on its stack to -1, and disallowing the popping of activation records. At this point, the writer releases its locks.

All the waiting readers are released and are free to continue execution. The program counters have all been set to a negative value, however, and so no further statements can be executed. The reader threads run through to completion and terminate. None of the activation records are popped during this step.

It is necessary to ensure that all the threads that were executing `wait()` and `join()` operations at the source are restored to their original condition at the destination before the other agent threads are restarted.

Extending the guarantee of transparent interruption and restoration of long-running operations to library code, is non-trivial. Libraries may implement guarded `wait()`s, `sleep()`s or `join()`s by using loops with condition checks around these operations; an approach similar to that described in [21]. When a `MobileThread` interrupts its `Thread`, its held locks would not be released immediately in this case. Agent relocation would be delayed, perhaps for an unacceptable amount of time.

We believe that most calls to the standard Java library will terminate within an acceptable amount of time. In the absence of a mechanism that can save the state of a `Thread` executing a library call, the best option is for the compiler to flag library calls that could lead to potentially long operations and indicate that no guarantee about the immediate migration of an agent is possible. A message could be printed out to the programmer and the decision would have to be taken by him/her as to whether the delay in migration would be acceptable. Should the programmer desire a finer granularity of control, he/she should pass the library through the mobility translator. Another possibility would be to implement native code wrapper methods around `wait()`, `sleep()` and `join()`, thus allowing a `MobileThread` to detect and interrupt its `Thread`'s long operations. This would have to be at the bytecode level.

If two agents try to dispatch one another, the synchronization technique we have adopted could lead to a deadlock. Agent `a` would synchronize on itself for executing the statement `b.go(dest)`, which would require synchronization on `b` to protect the integrity of `b`'s stacks. If similarly, `b` would execute `a.go(dest)`, a deadlock would result. To prevent this, the call of `dispatch()` within `realGo()` is synchronized on the agent context instead of on the caller.

When two agents try to move one another, and `a` executes `b.go(dest)` and `b` executes `a.go(dest)`, each Aglet sends a `go` message to the other. If `a`'s `go()` method synchronizes on the agent context first, every thread inside `a` is interrupted before a move. This includes the thread that is attempting to move `b`. All of `a`'s threads get interrupted, `a`'s state is saved, and `a` is moved to its destination. Since `a`'s attempt to move `b` is interrupted, `b` does not move.

If multiple threads within the same agent attempt to move the agent, deadlock could still result. More than one thread could call `go()`. Only one of them will actually synchronize on the agent context. Now, when this writer thread attempts to acquire all the locks, it will not be able to. This is because the other threads that are attempting to move the agent will be blocked, waiting to acquire a lock on the agent context. An additional level of synchronization is introduced in order to avoid this. Every agent maintains a condition variable in the agent context. This indicates whether the agent is currently being moved or not. The first writer thread will acquire a lock on this variable, test and set it, and then release the lock. Subsequent threads will acquire the lock, test the variable, and then release the lock by throwing a `MoveRefusedException`.

7.2 Synchronization Blocks

The Java semantics for `synchronized` blocks or methods are that the locks acquired by a thread on entering them are released when the agent is migrated. When users use synchronization to protect the agents' internal data structure, this protection must extend across machine boundaries.

For weakly mobile languages, synchronized blocks are a non-issue since code never executes beyond the call to `go()`. In strongly mobile systems, however, a call to `go()` may appear at any point within a synchronized block.

Difficulties stem from the fact that object locks are not stored within the object during serialization, but are hidden within the virtual machine. To tackle this problem we introduce serializable locks in place of standard Java object locks. Client programmers use the standard `synchronized` keyword to enforce synchronization constraints. During the translation phase, an object of class `MobileMutex` is introduced for each object that requires synchronization. Whenever a programmer requests object locking through the use of the Java `synchronized` keyword, the lock is actually taken out and released via calls to `lock()` and `unlock()` on the associated `MobileMutex` object. In this way, synchronized blocks and methods are eliminated from the original source, and re-implemented using the new locking mechanism. The overhead is minimal, and synchronization semantics are preserved across a move.

`synchronized` blocks are often used in conjunction with `wait()` and `notify()` operations. These are translated such that their semantics are preserved even after the translation of `synchronized` blocks.

If synchronized blocks are to be made transparent across moves, a `MobileMutex` object needs to be added to the object on which synchronization is desired. In our current implementation, this is only possible if the programmer has access to the source code of that object, if the object is itself an agent, or if the programmer has source access to

every synchronization on the object. In the next version of the translator, we will address this issue by associating a `MobileMutex` object with every `java.lang.Object`.

8 Performance

8.1 Optimizations

The translation mechanism discussed do far is overly conservative and thus inefficient. We have identified some optimizations for the above translation algorithms that are simple enough to be done automatically by a compiler:

- If a method is not recursive, or if it is tail-recursive and the compiler can determine that the execution time is bounded, it should not be translated into a class.
- To reduce the overhead of synchronization and program counter update, statements should be grouped to form logical, atomic statements.
- If the number of statements executed inside a loop is sufficiently small, and the statements are simple, i.e., no method calls or loops, a lock acquire and release could be made every *N* (say 10,000) simple statements. This would mean that in the case of a `go()` call, upto *N* statements would need to be executed before the move actually takes place.
- Loop unrolling and method inlining could reduce overhead.
- If a local variable is limited in scope to only one logical statement, it should remain a local variable, and should not be translated into a field of the generated class.
- Code that checkpoints every *N* simple statements, or every *N* milliseconds could be generated.

8.2 Measurements

Measurements were taken to estimate the cost of the described translation mechanism for agents. Standard Java benchmarks were rewritten in the form of both strongly mobile agents and Aglets. This did not involve changing the timed code significantly. The only changes that needed to be made to the original benchmarks' code were made to avoid method calls inside expressions. This is because the preprocessor does not as yet handle these.

The strongly mobile agents were passed through the translator. We then used simple manual optimization techniques to improve the performance of the translated agents. These are - the grouping of simple statements to form logical, atomic statements; the obtaining and releasing of locks every 10,000 simple statements for a loop; the inlining of method calls to simple methods that in turn, do not contain method calls.

Benchmark	Translated Code	Optimized Code
Crypt(array size - 3000000)	5.61X	1.23X
Crypt(array size - 3000000) multi-threaded version - 1 thread	5.96X	1.30X
Crypt(array size - 3000000) multi-threaded version - 2 threads	6.00X	1.41X
Crypt(array size - 3000000) multi-threaded version - 5 threads	5.60X	1.31X
Linpack(500 X 500)	10.00X	1.75X
Linpack(1000 X 1000)	9.48X	1.65X
Tak(100 passes)	245.30X	220.83X
Tak(10 passes)	247.00X	213.60X
Simple Recursion (sum of first 100 natural nos. - 10000 passes)	68.27X	60.75X

Table 1. Execution time of Strongly Mobile Agents compared to corresponding Aglets

Benchmark	Translated Code	Optimized Code	Aglet
Crypt	32.10	30.69	30.44
Crypt - multi-threaded 1 thread	32.54	30.82	30.35
Crypt - multi-threaded 2 threads	32.56	30.82	30.35
Crypt - multi-threaded 5 threads	32.54	30.83	30.38
Linpack(500 X 500)	31.02	30.02	28.34
Linpack(1000 X 1000)	58.27	52.94	51.24
Tak(100 passes)	22.04	21.99	20.98
Tak(10 passes)	22.05	22.02	20.98
Simple Recursion	22.03	21.82	21.02

Table 2. Memory utilization of Strongly Mobile Agents and Aglets (MB)

The running times and memory footprints of the translated agents and the manually optimized agents were compared with the equivalent weakly mobile Aglets. The results have been presented in table 1, and in table 2. A major contributor to the poor running times of the recursive benchmark programs is the Garbage Collector which runs several times a second.

We performed some further optimzations on the Linpack benchmark. The time taken by Linpack depends to a great extent on a particular method call inside a double-nested loop. This method contains another loop. We manually inlined this method, and measured execution time with the inner-most loop untranslated, and with the translated loop unrolled. The running time comparisons are presented in table 3, and the memory footprint results are in table 4. A user could obtain a performance improvement by including annotations in the code to inform the preprocessor how to optimally translate certain code portions.

A comparison of the code sizes of the agent code out-

Linpack Optimizations	Inner Loop Untranslated	Inner Loop Unrolled 2 times	Inner Loop Unrolled 10 times
Linpack (500 X 500)	1.02X	1.21X	0.75X
Linpack (1000 X 1000)	1.02X	1.15X	0.76X

Table 3. Execution time of Optimized Strongly Mobile Agents compared to corresponding Aglets for Linpack

Linpack Optimizations	Inner Loop Untranslated	Inner Loop Unrolled 2 times	Inner Loop Unrolled 10 times
Linpack (500 X 500)	29.9	30.19	30.48
Linpack (1000 X 1000)	52.8	53.12	53.40

Table 4. Memory utilization of Optimized Strongly Mobile Agents for Linpack (MB)

Number of stack frames	Agent pack time	Agent dispatch time	Aglets dispatch time
1	6	2436	1750
2	5	5421	1875
3	5	5410	1830

Table 5. Migration Time for Single-threaded Strongly Mobile Agents and Aglets (ms) - Linpack

Number of threads	Agent pack time	Agent dispatch time	Aglets dispatch time
1	9	5266	1782
2	10	5133	1860
5	16	5126	1803

Table 6. Migration Time for Multi-threaded Strongly Mobile Agents and Aglets (ms) - 5 frames on main thread stack, 2 frames on other threads' stacks - Multi-threaded Crypt

put by the preprocessor, and that of the corresponding simple Aglets, was made for the benchmarks discussed above. This was done by comparing their `.class` files. For the benchmarks discussed previously, the translated agents are between 6 and 14 times the sizes of the simple Aglets.

The overhead of migrating agents depends on the amount of state that the agent requires to carry along with itself. This is dependent on the number of threads within the agent, and on the number of frames on the runtime stack of the threads. The migration costs of moving a single threaded agent with different numbers of frames on the stack have two components - the time required to pack up the agent state, and the time to move the agent. The latter is the time required for the translated agent to execute the Aglets dispatch method. We compare this against the time required for the transfer of the simple benchmark Aglet. Agents and Aglets are transferred between ports on the same machine, in order to obtain a meaningful comparison that is unaffected by network delay. The results for different stack sizes are shown in table 5.

Similarly, the dependence of the migration cost of a multi-threaded agent, on the number of threads is shown in table 6.

The measurements were taken on a Sun Enterprise 450 (4 X UltraSPARC-II 296MHz), with 1GB of main memory, running Solaris. We used the Sun JDK 1.4.0_01 HotSpot VM in mixed mode execution, with the heap size limited to 120MB.

9 Conclusions

We have argued that strong mobility is an important abstraction for developing Grid Computing applications, and have outlined a source translation scheme that translates strongly mobile code into weakly mobile code by using a preprocessor. The API for the strongly mobile code and the translation mechanism are designed to give programmers full flexibility in using multi-threaded agents, and in dealing with any synchronization problems.

We are able to handle almost the entire Java programming language. Time constraints mean that the translation of some constructs like inner classes, and `try` blocks have not yet been implemented. If an agent uses library code that contains guarded `wait`, `sleep` or `join` calls, rapid termination before a move cannot be guaranteed. `synchronized` blocks that synchronize on an untranslated Object in user code cannot be transparently migrated. In both these situations, the translator is designed to display a warning for the programmer. Some resources need to be accessed on the machine where the agent originated, and should be declared `global` by the programmer. An RMI server to do this needs to be implemented. Timed operations, like open network connections, are not preserved across a relocation. Mobile agents need to be prevented from sharing objects with one another, or non-mobile objects. We will investigate using Isolates [7] for this purpose.

Source code, rather than bytecode translation, does not involve decompilation, and is more convenient. The performance measurements indicate that our approach to achieving strong mobility for Java is practical. In future, we will use analysis techniques to automate the generation of optimized source code. Measurements also indicate that performance can be improved further by allowing programmers to make annotations to source code.

Our preprocessor currently generates Java code that uses IBM's Aglets library. In future versions of our translator, we will instead target the *ProActive* weak mobility system, or RMI directly.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[2] L. Bettini and R. D. Nicola. Translating Strong Mobility into Weak Mobility. In *Mobile Agents*, pages 182–197, 2001.

[3] S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer. Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. Technical Report RR-4662, INRIA, December 2002.

[4] J. Bradshaw, N. Suri, A. J. Caas, R. Davis, K. M. Ford, R. R. Hoffman, R. Jeffers, and T. Reichherzer. Terraforming Cyberspace. In *Computer*, volume 34(7). IEEE, July 2001.

[5] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner. Implementation of Strong Mobility for Multi-Threaded Agents in Java. Technical Report OSU-CISRC-2/03-TR06, Department of Computer and Information Science, The Ohio State University, February 2003.

[6] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[7] G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Tampa, FL, Oct. 2001.

[8] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems*, Berkeley, CA, February 2003.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3), 2001.

[10] S. Fünfrocken. Transparent Migration of Java-based Mobile Agents: Capturing and Reestablishing the State of Java Programs. In *Proceedings of the Second International Workshop on Mobile Agents*, Stuttgart, Germany, September 1998.

[11] Gnutella. http://www.gnutella.com.

[12] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software— Practice and Experience*, 32(6), May 2002.

[13] Grid Physics Network. http://www.griphyn.org.

[14] A. Holub. Reader/writer locks. *Java World*, April 1999. http://www.javaworld.com/javaworld/jw-04-1999/jw-04-toolbox-p3.html.

[15] A. Iamnitchi, I. Foster, and D. Nurmi. A Peer-to-peer Approach to Resource Discovery in Grid Environments. In *11th Symposium on High Performance Distributed Computing*, Edinburgh, UK, August 2002.

[16] T. Illmann, T. Krüger, F. Kargl, and M. Weber. Transparent Migration of Mobile Agents using the Java Platform Debugger Architecture. In *Proceedings of the 5th International Conference on Mobile Agents*, Atlanta, GA, December 2001.

[17] H. Jiang and V. Chaudhary. Compile/Run-time Support for Thread Migration. In *16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[18] H. Jiang and V. Chaudhary. On Improving Thread Migration: Safety and Performance. In *9th International Conference on High Performance Computing*, Dec. 2002.

[19] D. Kotz, R. Gray, and D. Rus. Future Directions for Mobile-Agent Research. *IEEE Distributed Systems Online*, 3(8), August 2002. http://dsonline.computer.org/0208/f/kot.htm.

[20] D. B. Lange and M. Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.

[21] D. Lea. *Concurrent Programming in Java[tm]: Design Principles and Patterns*. The Java Series. Addison Wesley, 2nd edition, 1999.

[22] B. Overeinder, N. Wijngaards, M. van Steen, and F. Brazier. Multi-Agent Support for Internet-Scale Grid Management. In *AISB'02 Symposium on AI and Grid Computing*, April 2002.

[23] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *First International Workshop on Mobile Agents*, Berlin, Germany, Apr. 1997.

[24] O. Rana and D. Walker. The Agent Grid: Agent-Based Resource Integration in PSEs. In *16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, August 2000.

[25] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of Agent Systems, Mobile Agents, and Applications*, 2000.

[26] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Models and Languages*, 1999.

[27] SETI@home. http://setiathome.ssl.berkeley.edu.

[28] T. Suezawa. Persistent execution state of a Java virtual machine. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[29] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, and R. Jeffers. Strong Mobility and Fine-Grained Resource Control in NOMADS. In *Proceedings of the Second International Symposium on Agent Systems and Applications / Fourth International Symposium on Mobile Agents*, Zurich, Sept. 2000.

[30] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents*, Zurich, Switzerland, September 2000.

[31] X. Wang. Translation from Strong Mobility to Weak Mobility for Java. Master's thesis, The Ohio State University, 2001.

[32] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, September 2002.