An Optimizing Translation Framework for Strongly Mobile Java

Arvind Saini and Gerald Baumgartner

Division of Computer Science and Engineering
School of Electrical Engineering and Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA

Abstract. Strongly mobile agents provide a convenient abstraction mechanism for migrating applications to the location of their data or as a container for deploying computational tasks in a cloud computing environment. They are difficult to implement on a stock Java VM, however, since it does not allow the computation state to be captured. We describe an implementation approach that translates strongly mobile Java into weakly mobile Java in which the generated code maintains serializable run-time stacks for the agent threads at all times. We discuss the optimizations that are needed for generating efficient weakly mobile code.

1 Introduction

For certain distributed applications, mobile agents (or mobile objects) provide a more convenient programming abstraction than remote method invocation (RMI). If an application needs to process large amounts of remote data, it may be less communication intensive to ship the computation in the form of a mobile agent to the location of the data than to use RMI calls to get the data and perform the computation locally. Mobile agents are also less affected by network connectivity. While the mobile agent is computing at a remote site, the home machine does not need to remain connected to the internet, which is especially useful if the home machine is a mobile device.

In mobile agent applications, agents typically operate autonomously using one or more threads that conceptually run within the agent. Existing mobile agent libraries for Java, such as Aglets [11, 10] or ProActive [2], however, only provide support for *weak mobility*, which allows migrating the agent object but requires that all threads are terminated before migration. However, *Strong Mobility*, which allows an agent to migrate seamlessly with running threads, would be the preferable programming abstraction. It allows a more natural programming style, since the logic for how and when an agent should migrate can be expressed procedurally and since it does not require the programmer to manually terminate all threads before migration and restart them at the destination. It also separates the migration mechanism from the application logic. Strong mobility, unfortunately, is difficult to implement since the Java Virtual Machine (VM) does not provide access to the run-time stacks of threads.

In previous research, we implemented support for strong mobility as a source-to-source translator from strongly mobile Java into weakly mobile Java [6, 20]. We also

demonstrated that strongly mobile agents can be used as containers for deploying applications on a desktop grid [4, 5] or in the cloud [14]. They allow migrating an application that is encapsulated within the agent without the application programmer having to be aware of the migration.

Our mobility translator generates weakly mobile code by implementing the run-time stack of a thread as a serializable Java data structure. Compared to other approaches to strong mobility this has the advantage that it allows multi-threaded strongly mobile agents without modifying the Java VM. The disadvantage, however, is that it results in very inefficient code. Since a run-time stack is modified by the thread that owns it as well as by a thread that wants to migrate the agent, a locking mechanism is required to protect the integrity of the stacks. With fine-grained locking, this results in a high run-time overhead.

In this paper, we describe an optimization framework for our mobility translator. We present measurements for comparing the cost of different locking mechanisms. We also present a translation approach that can improve the performance of the generated code in exchange for a higher latency for migrations. Finally, we outline how standard compiler optimization techniques can be used for further optimizing the code.

In the next section, we provide more background on strong vs. weak mobility. Section 3 discusses related work on strong mobility. We explain our language and API design in Section 4 and the details of our mobility translator in Section 5. Section 6 presents experimental results on the potential speed improvements for mobile agents and Section 7 provides concluding remarks.

2 Background

Mobile agents and remote method invocation have the same expressive power. Any agent program can be translated into an equivalent RMI program and vice versa. In fact, either mechanism can be implemented on top of the other. Similar to loops and recursion, however, some problems are more naturally expressed in one of these programming styles.

In actual implementations, RMI is implemented on top of TCP together with object serialization to allow objects to be sent as arguments to remote methods. An agent migration is then implemented by the agent environment on the home machine performing a remote method invocation on the agent environment of the destination machine and passing the agent itself as argument to the remote method. In the case of weak mobility, only the agent object is sent to the destination. For strongly mobile agents, the execution state must be transferred as well.

A language with support for strong mobility provides an simple mental model for writing mobile agents. As an example, consider a network broadcast agent that prompts the user for input, relaying the input message to a number of other host machines. Using a Java-like language supporting strong mobility the solution is straightforward:

```
public void broadcast(String hosts[]) {
    System.out.println("Enter message:");
    String message = System.in.readln();
```

```
    for(int i = 0; i < hosts.length; i++) {
        try {
            dispatch(hosts[i]);
            System.out.println(message);
        }
        catch(Exception exc) {}
    }
    dispose();
}
```

Weak mobility does not allow migration of the execution state of methods (i.e., local variables and program counters). The dispatch operation simply does not return. Instead, the framework allows the developer to tie code to certain mobility-related events. E.g., in IBM's Aglets framework, the developer can provide callback code that will execute when an object is first created, just before an object is dispatched, just after an object arrives at a site, etc. Consider the above application written in an Aglets-like framework:

```
private String hosts[];
private int i = 0;
private String message;

public void onCreation(String hosts) {
    this.hosts = hosts;
    System.out.println("Enter message:");
    message = System.in.readln();
}

public void onArrival() {
    System.out.println(message);
}

public void run() {
    if(i == hosts.length)
        dispose();
    dispatch(hosts[i++]);
}
```

Because weak mobility does not allow the execution state to be transferred, programmers must manually store the execution state in agent fields (which are transferred) and must reconstruct the information of where the agent is and what it needs to do next using the event handling methods. This scatters the logic for how the agent moves from host to host across multiple methods and, therefore, results in an unnatural and difficult programming style.

While weak mobility is a conceptually simple mechanism and relatively straightforward to implement, it results in complex mobile agent code that may have to be written by expert programmers. By contrast, strong mobility provides a simple programming paradigm but it is more difficult to implement, e.g., to ensure freedom of race conditions and deadlocks.

3 Related Work

There are two main techniques for implementing strong mobility: modifying the Java VM or via translation of either source code or bytecode.

Java Threads [3] , D'Agents [8], Sumatra [1], Merpati [17], and Ara [13] extend the Sun JVM. CIA [9] modifies the Java Platform Debugger Architecture. JavaThread, CIA, and Sumatra to not support *forced migration*, i.e., the ability of an outside thread or agent dispatching an agent. Also, D'Agents, Sumatra, Ara, and CIA do not support the migration of multi-threaded agents. NOMADS [18] uses a customized virtual machine called Aroma that supports forced mobility and multi-threaded agent migration. The drawback of all these approaches is that relying on a modified or customized VM make it difficult to port and deploy agent applications. NOMADS and Java Threads are only compatible with JDK 1.2.2 and below, D'Agents needs the modified Java 1.0 VM, and Merpati and Sumatra are no longer supported. Furthermore, NOMADS, Sumatra, and Merpati do not support just-in-time compilation.

WASP [7] and JavaGo [16] implement strong mobility in a source-to-source translator that constructs a serializable stack just before the migration using the exception handling mechanism. Neither system is able to support forced mobility. Also, JavaGo does not support multi-threaded agent migration and does not preserve locks on migration. Correlate [19] and JavaGoX [15] are implemented using byte code translation. While they support forced mobility, they do not support multi-threaded agent migration.

Instead of using a source-to-source or bytecode translator for creating a serializable stack before migration like the previous translation approaches, in our approach a source-to-source translator ensures that serializable stacks are maintained at all times [6, 20]. This allows both forced migration and multi-threaded agent migration. Also, our approach better maintains the Java semantics, e.g., by preserving synchronization locks across migrations.

4 Language and API Design

Unlike a weak mobility library, which requires several event handlers and utility classes to simplify programming of itineraries, strong mobility can be supported with a very simple API. Our original support for strong mobility consisted simply of the interface `Mobile` and the two classes `MobileObject` and `ContextInfo`. While the design looks like a library API, it is really a language extension, since our proposed translation mechanism compiles away the interface `Mobile` and the class `MobileObject`.

4.1 Basic Mobility Support

Every mobile agent must (directly or indirectly) implement the interface `Mobile`. Similar to Java RMI, a client of an agent must access the agent through an interface 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 java.io.IOException,
               com.ibm.aglet.RequestRefusedException;
}
```

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.

As explained in Section 5 below, we used the IBM Aglets library for implementing our support for strong mobility. This is currently reflected in the list of exceptions that can be thrown by `go()`. In a future version, we will add our own exception class(es) so that the surface language is independent of the implementation.

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.

```
public class MobileObject implements Mobile {
    private native ContextInfo realGetContextInfo();
    private native void realGo(java.net.URL dest)
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException;
    protected ContextInfo getContextInfo() {
        return realGetContextInfo();
    }
    public void go(java.net.URL dest)
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException {
        realGo(dest);
    }
}
```

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, including the host URL and any system objects or resources that the host wants to make accessible to a mobile agent.

The method `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. If `go()` is called from within an agent method `foo()`, the instruction following the call to `go()` is executed on the destination host. Typically, an agent would call `getContextInfo()` after a call to `go()` to get access to any system resources at the destination.

A mobile agent class could then simply be defined as a subclass of class `MobileObject` and would typically contain a thread that carries out the agent actions and moves to remote machines when needed.

4.2 Language Extensions

In a non-mobile applications, static fields of a class are shared between all the instances of that class. I.e., only one copy of a static field exists in the application. In a mobile

application, when the code is migrated to a remote machine together with an agent, a new copy of a static field will be created at the destination. Depending on the use of the static field this may or may not be the desirable behavior. If a static field is used, say, to count the number of instances of a class, it may be preferable to have a globally unique field. Similar to static fields, we propose a declaration for global variables,

```
global int n;
```

such that an agent always accesses global variable on the home machine, not on the machine the agent currently resides on.

We also propose a new language construct `immobile`, both as a modifier for methods and as a block of code,

```
immobile int foo() { ... }
int bar() { ... \immobile{ ... } ... }
```

that inhibits migration of an agent while it is executing immobile code. This gives programmers better control of mobility for multi-threaded agents, e.g., by postponing a migration until a large intermediate data structure has been deallocated.

4.3 Mobile Threads and Thread Pools

In our original implementation of strong mobility for multi-threaded agents, we used `Thread` and `ThreadGroup` objects in the strongly mobile code and generated the wrapper classes `MobileThread` and `MobileThreadGroup` as part of the weakly mobile code. These mobile threads were not available to programmers, though. We are exploring a redesign of the API that would use either thread pools or executors instead of thread groups and that would make a mobile thread API available to the programmer. This would allow programmers to use serializable mobile threads standalone without using agents.

Migration for mobile agents is a similar problem to checkpointing a high performance computing application. With checkpointing, a thread is serialized and written to disk. If the processor fails, the thread is read back in and restarted. This is the same mechanism needed for thread migration with the disk acting as a very slow communication link. Providing an API for mobile threads and thread pools would allow our mobility translator to be used for automatically generating checkpointing code.

5 Translation from Strong to Weak Mobility

5.1 Single-Threaded Agents

For efficiency reasons it would be desirable to provide virtual machine support for strong mobility. However, a preprocessor or compiler implementation has the advantage that the generated code can run on any Java VM, and that it is easier to implement and to experiment with the language design.

For our initial prototype, we chose to design the translation mechanism for a preprocessor that translates strongly mobile code into weakly mobile code that uses the

Aglets library. For our current reimplementation, we will generate code for the ProActive library [2].

For implementing strong mobility in a preprocessor, it is necessary to save the state of a computation before moving an agent so it can be recovered afterwards. Fünfrocken describes a translation mechanism that inserts code for saving local variables just before moving the agent [7]. This has the disadvantage that the `go()` method cannot be called from arbitrary points outside the agent.

Our translation approach is to maintain a serializable version of the computation state at all times by letting the agent implement its own run-time stack. This increases the cost of regular computation as compared to Fünfrocken's approach, but it simplifies restarting the agent at the remote site.

5.2 Translation of Methods

For making the local state of a method serializable, we implement activation records of agent methods as objects. For each agent method, the preprocessor generates a class whose instances represent the activation records for this method.

An activation record class for a method is a subclass of the abstract class `Frame`:

```
public abstract class Frame
    implements Cloneable, java.io.Serializable {
    public Object clone() { ... }
    abstract void run();
}
```

Activation records must be cloneable for implementing recursion as explained below. The translated method code will be generated in method `run()`.

For example, given an agent class `C` with a method `foo` of the form

```
void foo(int x) throws AgletsException {
    int y = x + 1;
    go(new URL(dest));
    System.out.println(y);
}
```

(and ignoring exception handling and synchronization for simplicity) we might generate a class `Foo` of activation records for `foo` of the form

```
class Foo extends Frame {
    C This;
    int x;
    int y;
    int pc = 0;          // program counter

    Foo(C t) { this.This = t; }
    void setArgs(int x) { this.x = x; }
    void run() {
        if (pc == 0) { pc++;  y = x + 1; }
        if (pc == 1) { pc++;
```

```
                        go(new URL(This.dest)); This.run1(); }
            if (pc == 2) { pc++;  System.out.println(y); }
        }
    }
```

The parameter and the local variable of method `foo()` became fields of class `Foo`. In addition, we introduced a program counter field `pc` and a variable `This` for accessing fields in the agent object.

The method `run()` contains the original code of `foo()` together with code for incrementing the program counter and for allowing `run()` to resume computation after moving. Calls of agent methods are broken up into a call of the generated method followed by `This.run1()`, as explained below. For allowing the agent to be dispatched by code outside the agent class, the program counter increment and the following instruction must be performed atomically, which requires additional synchronization code.

For efficiency, the preprocessor could group multiple statements into a single statement and only allow the agent to be moved at certain strategic locations.

5.3 Translation of Agent Classes

An agent now must carry along its own run-time stack and method dispatch table. The generated agent class contains a `Frame` array as a method table and a `Stack` of `Frame`s as the run-time stack. When calling a method, the appropriate entry from the method table is cloned and put on the stack. After passing the arguments, the `run` method executes the body of the original method `foo` while updating the program counter.

Suppose we have an agent class `AgentImpl` of the form

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

Since this class indirectly implements interface `Mobile`, the preprocessor translates it into the following code:

```
public class AgentImpl extends Aglet {
    int a;
    Frame[] vtable = { new Foo(this) };
    final int _foo = 0;
    Stack stack = new Stack();

    public void onCreation (Object init) {
        /* initialization code */
    }

    public void foo(int x) {
        Foo frame = (Foo) (vtable[_foo].clone());
```

```
        stack.push(frame);
        frame.setArgs(x);
    }

    public void run1() {
        Frame frame = (Frame) stack.peek();
        frame.run();
        stack.pop();
    }

    class Foo extends Frame { /* as described above */ }
}
```

The preprocessor eliminates interface `Mobile` and class `MobileObject` and lets the agent class extend class `Aglets`.

For implementing method dispatch, the agent includes a method table `vtable` of type `Frame[]`. The constant `_foo` is the index into the method table for method `foo`. The field `stack` implements the run-time stack.

The constructor of class `AgentImpl` is translated into the method `onCreation`. Since Aglets only allows a single `Object` as argument of `onCreation()`, any original constructor arguments must be packaged in an array or vector by the preprocessor.

As described above, the original agent method `foo()` gets translated into a local class `Foo` of activation records. The method `foo()` in the generated code implements the call sequence: it allocates an activation record on the stack and passes the arguments. The code for executing the method on the top of the stack and for popping the activation record in method `run1()` is shared between all methods. A client must first call `foo()` followed by a call to `run1()`.

For resuming execution after arriving at the destination, we must also generate a method `run()` inside class `AgentImpl`:

```
public void run() {
    while (! stack.empty())
        run1();
}
```

5.4 Protection of Agent Stacks

It is imperative that an agent cannot be dispatched by another thread between incrementing the program counter and executing the following statement. If the program counter increment and the following statement were not executed atomically, a thread could be dispatched after the program counter increment and incorrectly miss execution of the statement upon arrival. Since by definition this type of synchronization need not be maintained across VM boundaries, standard Java synchronization techniques are used. For a single-threaded agent, we simply synchronize on the agent object itself. For method calls, we only need to protect the call to set up the activation record. The actual execution of `run1()` does not need to be synchronized since by then a new activation record with its own `pc` will be on top of the stack:

```
synchronized(This) { pc++; go(new URL(This.dest)); }
This.run1();
```

For preventing the agent from being dispatched between the program counter increment and the next instruction, the call of `realGo()` in `MobileObject.go()` must also be synchronized on the agent object.

If two agents try to dispatch one another, this synchronization code could lead to a deadlock. For executing the statement `b.go(dest)`, Agent `a` would first synchronize on itself. Then a synchronization on `b` would be required to protect the integrity of `b`'s stack. If similarly `b` would execute `a.go(dest)`, a deadlock would result. To prevent this, the call of `realGo()` is synchronized on the agent context instead of on the caller.

```
public class MobileObject implements Mobile {
    public void go(java.net.URL dest)
        throws IOException, RequestRefusedException {
        synchronized(TheAgentContext) {
            synchronized(this) { realGo(dest); }}
    }
```

The only time any thread synchronizes on two objects is now in the call of `realGo()`, in which case the first synchronization is on the agent context. Deadlocks are, therefore, prevented.

This synchronization mechanism ensures that only one agent can migrate at a time. If two agents `a` and `b` try to dispatch one another, the first one, say `a`, will succeed. By the time `b` tries to dispatch `a`, `a` is already on a different host. The call to `a.go()` will, therefore, throw an exception that must be handled by `b`.

5.5 Multi-Threaded Agents

Our mobility translator supports migration of multithreaded agents. Unfortunately, the Java library classes `Thread` and `ThreadGroup` are not serializable. Therefore, for each use of the classes `Thread` and `ThreadGroup` we need to generate a serializable wrapper of classes `MobileThread` and `MobileThreadGroup`, respectively. The `go()` method on an agent can be invoked by another agent in the system or by a thread within the agent itself. The `go()` method calls the `realgo()` method to check whether the agent is already on the move. If so, a `MoveInterrupt` exception is thrown. Otherwise, each `MobileThread` calls the `interrupt()` method of the underlying`Thread` class. This terminates any `wait()`, `join()`, or `move()` functions if they are being executed. The time remaining to completely execute these function calls is saved so that the function can resume execution at the destination from the point where it had been interrupted.

The next step is to call the `packUp()` method of the `main` agent wrapper of the thread group. This in turn calls the `packUp()` methods of the wrappers for all the threads and the thread groups. The underlying state of execution of each thread and thread group is saved to the corresponding wrappers. All the threads are forced to halt any further executions and subsequently the agent is shipped to the destination by the `dispatch()` call. At the destination, the `reinit()` method of the `main` agent

thread group wrapper is invoked. This method calls the `reinit()` method of each wrapper. The called `reinit()` methods create `Thread` or `ThreadGroup` objects from their corresponding wrappers and the execution states of the threads are restored.

After the restoration of the execution states, the `start()` method of the `main` thread group wrapper is called. This method invokes the `start()` methods of all the `MobileThread` wrappers. Then `start()` method of the underlying thread is called, which then calls the `run()` method of the `MobileThread` wrapper. The `run()` method checks the stack of the `MobileThread` wrapper. If the stack is empty, then the `run()` method of the `Runnable` target is called. Otherwise, the activation records in the stack are executed.

5.6 Synchronization in Multi-Threaded Agents

An agent should not be shipped to the destination while a thread is in the middle of executing a statement. To prevent this from happening, the program counter update and a statement execution should be performed atomically. Neither should any two agents dispatch each other at the same time nor should two threads within the same agent try to move the agent simultaneously. For example, each statement in the thread is protected by a lock mechanism as shown below:

```
Acquire lock;
Program counter update;
Statement execution;
Release lock;
```

The problem of lock synchronization for multi-threaded agents is comparable to the readers-write problem with writers priority. Each thread in the agent is assigned a lock. The threads that are executing statements are considered to be readers and the thread that invokes the `go()` method to move the agent is considered to be the writer. After the reader thread is done executing the statement, the lock is released and acquired by the writer thread. When the writer thread has acquired the locks of all the readers, only then can the agent be allowed to relocate.

The drawback by having a locking mechanism around each program counter update and statement, is that it incurs a large overhead. On the other hand, synchronizing on an entire agent instance reduces the degree of parallelism in the system.

5.7 Optimizations

Our translation mechanism introduces several sources of inefficiencies. Migration of a strongly mobile agent is slower than that of a handwritten weakly mobile agent, because the run-time stacks need to be serialized and shipped along with the agent. However, since the expected behavior of mobile agents is that they spend a significantly larger amount of time computing than migrating, the overhead imposed on regular computation is of much more concern. The computation overhead comes from three sources: the locking mechanism for protecting the run-time stacks, the frequency of locking and the associated overhead of testing and incrementing the program counter, and pushing activation records onto the run-time stacks.

A straight-forward optimization is to combine multiple consecutive statements, e.g., multiple assignments, into a single block without releasing and re-acquiring the lock after each statement. This increases the latency slightly until a call to `go()` is honored and the agent can migrate, but given the infrequency and cost of migration, even a latency of up to 1 second would likely not be a problem for most applications.

Much of the locking overhead itself comes from insuring that writers (i.e., threads that want to move an agent) do not starve. A readers-writer lock with reader priority would be significantly cheaper but it could not insure freedom of starvation for writers. Since writers occur very infrequently, it is possible to keep the stack locked for readers by default and only allow a writer to proceed if one is pending. E.g., instead of releasing and re-acquiring the lock, we could use

```
if (Writer is present) { Release lock;  Acquire lock; }
```

using an atomic Boolean or atomic integer to test for the presence of writers.

Such a locking-scheme then allows a different code structure. Instead of having lock-unlock pairs around statements or consecutive groups of statements, it would be possible to have these if-conditions with unlock-lock pairs only in a few strategic places in the code. Again, this would increase the latency until a migration can take place, but it has the potential to drastically improve performance.

In addition, it would be possible to use standard compiler optimizations to further reduce the run-time overhead. The overhead of maintaining the program counter for a loop can be reduced by unrolling the loop. Inlining of methods can be used to eliminate the expensive method call sequence. Methods that do not contain loops may not need to be translated at all. Finally, with worst-case execution time analysis, it would be possible to give a bound on the run-time of a method or code fragment and only generate locking code to test for the presence of writers if the worst-case execution time is more than the acceptable migration latency.

6 Experiments

To indicate the overhead of our translation mechanism and the potential for optimizations, we first present the results of manual optimizations and measurements that had been performed in prior work [6]. These measurements were made on a quad-core UltraSparc-II 296MHz processor with 1GB of memory running Solaris and using the Sun JDK 1.4.0 Hotspot VM.

For these measurements, 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 benchmarking code were made to avoid method calls inside expressions, since the preprocessor did not 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 acquiring and releasing of locks only every 10,000 simple statements for a loop; and the inlining of calls to simple methods that in turn do not contain method calls.

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. A major contributor to the poor running times of the recursive benchmark programs is the garbage collector that runs several times a second during their execution.

Table 1. Execution time of strongly mobile agents compared to corresponding Aglets code.

Benchmark	Translated Code	Optimized Code
Crypt (array size: 3,000,000, no threads)	5.61X	1.23X
Crypt (array size: 3,000,000, 1 thread)	5.96X	1.30X
Crypt(array size: 3,000,000, 2 threads)	6.00X	1.41X
Crypt(array size, 3,000,000, 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 1–100, 10,000 passes)	68.27X	60.75X

We performed further optimzations on the Linpack benchmark, a matrix multiplication implementation. The inner-most loop of Linpack is inside a dot-product method. 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 2.

Table 2. Potential performance improvements for inner loop transformations of strongly mobile Linpack code relative to Aglets.

Linpack Version	Untranslated	Unrolled 2X	Unrolled 10X
Linpack (500 X 500)	1.02X	1.21X	0.75X
Linpack (1,000 X 1,000)	1.02X	1.15X	0.76X

For finding the cheapest locking mechanisms, we performed micro-measurements of lock-unlock pairs for several different locking mechanisms as well as using atomic integers or Booleans as guards for a lock. These measurements were performed on a quad-core, 2.4GHz Xeon workstation running Linux. Since all code is sequential and to make the measurements more predictable, we disabled multi-core support, hyper-threading, Intel Turbo Boost (overclocking), and Intel Speed Step (CPU throttling), and turned off all network interfaces, the X window system, and unnecessary background processes. The Java Version 1.7.0_21 and ran the measurements on the Java server VM with the command line options `-XX:CICompilerCount=1` and `-Xbatch` to ensure that the measurements are not distorted by background compilation. We took 100 measurements of 10,000 lock-unlock pairs each in a 10X-unrolled loop. The average times are shown in Table 3.

Table 3. Average execution time for one lock-unlock pair.

Locking Mechanism	Time (ns)
Semaphore	18.32
ReentrantLock	16.41
ReentrantReadWriteLock (Read Lock)	26.77
ReentrantReadWriteLock (Write Lock)	23.84
AtomicBool (as guard for lock)	16.09
AtomicInt (as guard for lock)	15.92

As our measurements show, the cheapest combination would be to use an atomic integer or Boolean (the difference between them is not statistically significant) as a guard for a `ReentrantLock` instead of our original counting `Semaphore`. With guarded locks it would be possible to generate code that unlocks and re-acquire the lock less frequently. This, together with compiler optimizations such as not translating inner loops or methods without loops, inlining, and loop unrolling has the potential to reduce the overhead to less than 20% for non-recursive applications, which would be acceptable.

7 Conclusion

We have presented a framework for translating strongly mobile Java code into weakly mobile code. Compared to existing approaches to strong mobility, our approach has the advantages that it allows multi-treaded agents and forced mobility, accurately maintains the Java semantics, and can run on a stock Java VM. The disadvantage is that without further optimizations, the run-time overhead would be prohibitively large.

The main contribution of this paper is that it presents an optimization framework for improving the performance of the generated weakly mobile code. Preliminary measurements show that with a combination of a cheaper locking mechanism and a code structure that trades off migration latency for performance, the overhead can become acceptably small. Finally, standard compiler optimization techniques can be used to further improve the performance of the generated code. We are currently working on a reimplementation of our mobility translator in the Polyglot compiler framework [12].

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek, editor, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer-Verlag, 1996.
2. F. Baude, D. Caromel, F. Huet, and J. Vayssìere. Communicating mobile active objects in Java. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643. Springer Verlag, May 2000.
3. S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. In *Software — Practice and Experience*, pages 355–394, 2002.

4. A. J. Chakravarti and G. Baumgartner. Self-organizing scheduling on the Organic Grid. *Int. Journal on High Performance Computing Applications*, 20(1):115–130, 2006.
5. A. J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-organizing computation on a peer-to-peer network. *Trans. Sys. Man Cyber. Part A*, 35(3):373–384, May 2005.
6. A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. In *Proceedings of the International Conference on Parallel Processing*, pages 321–330. IEEE Computer Society, Oct. 2003.
7. S. Fünfrocken. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, September 1998. Springer-Verlag.
8. R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.
9. T. Illmann, T. Krueger, 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*, MA '01, pages 198–212, London, UK, 2002. Springer-Verlag.
10. D. B. Lange and M. Oshima. Mobile agents with Java: the Aglets API. *World Wide Web Journal*, 1998.
11. D. B. Lange and M. Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.
12. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, Apr. 2003.
13. H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In R. Popescu-Zeletin and K. Rothermel, editors, *First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, Apr. 1997. Springer Verlag.
14. B. Peterson, G. Baumgartner, and Q. Wang. A hybrid cloud framework for scientific computing. In *8th IEEE International Conference on Cloud Computing, CLOUD 2015*, pages 373–380, New York, NY, June 2015.
15. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Springer Verlag Lecture Notes in Comuter Science*, 2000.
16. T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *In Proceedings of the 3 rd Intl. Conference on Coordination Models and Languages*, 1999.
17. T. Suezawa. Persistent execution state of a Java virtual machine. In *Java Grande*, pages 160–167, 2000.
18. N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, and T. S. Mitrovich. An overview of the NOMADS mobile agent system. In C. Bryce, editor, *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, 13 June 2000.
19. 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*, pages 29–43, Zurich, Switzerland, September 2000. Springer-Verlag.
20. X. Wang, J. Hallstrom, and G. Baumgartner. Reliability through strong mobility. In *Proc. of the 7th ECOOP Workshop on Mobile Object Systems: Development of Robust and High Confidence Agent Applications (MOS '01)*, pages 1–13, Budapest, Hungary, June 2001.