Louisiana State University
LSU Digital Commons

LSU Doctoral Dissertations | Graduate School

1-21-2018

An Optimizing Java Translation Framework for Automated Checkpointing and Strong Mobility

Arvind Kumar Saini
Louisiana State University and Agricultural and Mechanical College, asaini2@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

 Part of the Computer Sciences Commons

Recommended Citation
Saini, Arvind Kumar, "An Optimizing Java Translation Framework for Automated Checkpointing and Strong Mobility" (2018). *LSU Doctoral Dissertations*. 4195.
https://digitalcommons.lsu.edu/gradschool_dissertations/4195

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Digital Commons. For more information, please contactgradetd@lsu.edu.

AN OPTIMIZING JAVA TRANSLATION FRAMEWORK FOR AUTOMATED CHECKPOINTING AND STRONG MOBILITY

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The School of Electrical Engineering and Computer Science

by
Arvind Kumar Saini
M.S. Computer Science, Midwestern State University, 2008
May 2018

Dedicated to Aryabhata, the astronomer and mathematician of ancient India, who invented the digital zero.

Acknowledgements

First and foremost, I would like to thank my research advisor, Dr Gerald Baumgartner, for providing to me his valuable guidance and insights that contributed towards my successful completion of the dissertation thesis for the doctoral program at Louisiana State University. Being a student of Dr Baumgartner inspired me to further develop my analytical and critical thinking skills which has fostered my growth towards the path of a successful researcher and also to go deeper into the world of cutting-edge technology that has heralded the boom in information technology in the recent years. I would also like to thank my other professors at Louisiana State University and the faculty of Midwestern State University who laid the foundation of my Doctoral program. Finally, I would also like to thank my family members and friends in India and USA whose encouragement and support was invaluable during all these years of the Ph.D. program.

Table of Contents

Abstract

Long-running programs, e.g., in high-performance computing, need to write periodic checkpoints of their execution state to disk to allow them to recover from node failure. Manually adding checkpointing code to an application, however, is very tedious. The mechanisms needed for writing the execution state of a program to disk and restoring it are similar to those needed for migrating a running thread or a mobile object. We have extended a source-to-source translation scheme that allows the migration of mobile Java objects with running threads to make it more general and allow it to be used for automated checkpointing. Our translation scheme allows serializable threads to be written to disk or migrated with a mobile agent to a remote machine. The translator generates code that maintains a serializable run-time stack for each thread as a Java data structure. While this results in significant run-time overhead, it allows the checkpointing code to be generated automatically. We improved the locking mechanism that is needed to protect the run-time stack as well as the translation scheme. Our experimental results demonstrate an speedup of the generated code over the original translator and show that the approach is feasible in practice.

Chapter 1
Introduction

In the last decade, high performance computing has witnessed an exponential growth in the computing power due to multifold increase in the number of cores in high performance systems. Computation is divided into a number of tasks, with each task being executed in parallel by a thread assigned a core. Such kind of multithreaded programming on multiple cores results in faster performance when it comes to solving complex tasks. However, in the case of a core failure, the thread needs to restart execution from the beginning. This results in the delay in the computation time. The drawback can be overcome by saving the execution state of the thread on a memory disk at regular intervals. Writing of the execution state by the thread on memory storage device is known as the *checkpoint*. Whenever there is core failure on which a thread is being executed, the last checkpoint is read by the thread and execution resumes from the point where the interrupt had occurred. This results in reduced computation time as execution no longer restarts from the beginning of the thread.

It is very likely in the coming few years, High Performance Computing will advance from Petascale (10^{15} Flops) to Exascale (10^{18} Flops) technology. This implies that there will be a drastic increase in the number of processors in order to achieve a higher degree of parallel computing. Increase in the number of processors also means more core failures, resulting in the Mean-Time-Between-Failures getting reduced from hours to minute. To make a system more fault tolerant system, different techniques of checkpointing have been developed. In checkpointing, the execution state of the threads running on the cores is written or checkpointed on a memory disk, at regular intervals. In case of a processor failure, thread the last checkpoint is read and the thread starts re-executing from it was interrupted. We have developed a technique that utilizes the translation scheme to makes the task of writing the

code for checkpoint less tedious. In our method, mobile agents are deployed as serializable threads that can read and write on a storage device.

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 [16, 15] or ProActive [3], 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 because the Java Virtual Machine (VM) does not provide access to the run-time stacks of threads.

In the previous research, support for strong mobility was implemented as a source-to-source translator from strongly mobile Java into weakly mobile Java [8, 33]. It was also demonstrated that strongly mobile agents can be used as containers for deploying applications on a desktop

grid [6, 7] or in the cloud [20]. 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 dissertation, 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.

Chapter 2
Background

Exascale systems will have a larger number of cores in order to achieve the goal of increasing the speed of parallel computing. However, the increase in the number of cores will imply more frequent core failures resulting in delayed computations. In order to reduce overhead caused by core failure, fault-tolerant technique of checkpointing has been developed. For checkpointing, serializable threads are used to write the execution state of the program to the files or external storage disks at specified intervals. In case of a node failure, the thread can read the execution state from the last checkpoint and program resumes execution from where it was halted. Mobile agents can be deployed as serializable threads that can be used for checkpointing. 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 an 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 a 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 about 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.

Chapter 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 [5] , D'Agents [13], Sumatra [1], Merpati [29], and Ara [19] extend the Sun JVM. CIA [14] 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 [30] 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 [11] and JavaGo [28] 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 [31] and JavaGoX [24] 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 [8, 33]. 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.

Chapter 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 `go()` and `getContextInfo()`. 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.

4.2 Class `ContextInfo`

Class `ContextInfo` is used for an agent to access any resources on the machine it is currently running on:

```
public class ContextInfo implements java.io.Serializable {
     private java.net.URL hostURL;
     public ContextInfo (java.net.URL h) { hostURL = h; }
     public java.net.URL getHostURL() {
         return hostURL;
     }
     // ...
 }
```

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

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`. By publishing the interface to this object, agents can be written to access those resources.

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.

4.3 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 an exception of type `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.

When calling a method on an agent, an exception will be thrown if the agent is not reachable. As in Java RMI, this is expressed by declaring that the method might throw an exception. In our current design, we use the exception class `AgletException`. In a future version, we will provide our own exception class.

4.4 Checkpointing

For writing a serializable thread for manual or automated checkpointing, we define the interface `SerializableRunnable` as an extension of the `Serializable` and `Runnable` interfaces.

```
public interface SerializableRunnable extends Serializable, Runnable { }
```

A class `SerializableThread` is defined as an implementation of interface `SerializableRunnable`. `SerializableThread` class contains a `run()` method which is overridden by the extension of the class.

```
public class SerializableThread implements SerializableRunnable  {
public void run()  { }
}
```

A `Serializable` thread to be deployed as a mobile agent is an extension of the class `SerializableThread`. Translated strong mobility statements are encoded inside the `run()` method. The method `Restart()` resumes execution after reading the last checkpoint.

```
public class T extends SerializableThread {
    transient CheckPointer cp;
    transient boolean rollover = false;
    Semaphore sem = new Semaphore(1);
```

```
transient int pc = 0;
transient int i = 0;
T(CheckPointer c) throws IOException {
    cp = c;
}
void reinit(CheckPointer c) {
    cp = c;
    // ...
}

public void start() {
    run();
}

// Restarting the execution from the last checkpoint
public void restart(int i, CheckPointer c) throws IOException {
    pc = 0;
    System.out.println(i);
    cp = c;
    this.i = i + 1;
    run();
}

public int index() {
    return i;
}

// Transalted
```

```
    public void run() {......


    }
}
```

Chapter 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 [3].

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 [11]. 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();
}
```

The drawback of this technique is that agent might be invoked to relocate

5.4 New `run1()` Translation Scheme

```
public void run() throws InterruptedException {
    semaphore.acquire();
    while (!stack.empty())
    run1();
}
```

```
public void run1() throws InterruptedException {
    if (waitingWriters > 0) {
        semaphore.release();
        semaphore.acquire();
    }
    Frame frame = (Frame) stack.peek();
    frame.run();
    stack.pop();
}
```

Second method

```
public void run() throws InterruptedException {
    semaphore.acquire();
    while (!stack.empty())
    run1();
}


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

5.5 Protection of Thread 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`.

This synchronization mechanism could be extended to multi-threaded agents. In this case, each thread would have its own execution stack. Instead of on the agent, we would have to synchronize on the thread. In effect, the threads would become agents. When migrating an agent, we would have to ensure that all thread are in a safe state. This can be done by maintaining a counting semaphore whose count indicates how many thread are currently running.

5.6 Synchronization Blocks

When the method `go()` is called from inside a synchronized block or method, the semantics of the synchronized code block needs to be retained while relocating the agent. On encountering the synchronized method or block, the object gets hold of the intrinsic lock in order to ensure mutual exclusion within the block. However, the lock is lost when the agent is moved to the new destination. Due to the inherent security features of the JVM, the intrinsic locks are not saved in the objects during serialization. Instead, the locks are hidden within the JVM, thus making them inaccessible to the user.

However, the above problem can be eliminated by using `Serializable` locks. In the translation mechanism, an object of serializable `MobileMutex` class is created for an object that uses synchronization. Locking and unlocking of the synchronized code block is implemented by `lock()` and `unlock()` of the corresponding `SerializableLock` object. The `SerializableLock` class is shown below :

```
public class SerializableLock implements java.io.Serializable
boolean locked = false;
public SerializableLock() {}
public synchronized void lock() {
    if (!locked) locked = true;
```

```
        else {
            while (true) {
                try {
                    wait();
                    locked = true;
                    break;
                }
                catch(InterruptedException ex) {}
            }
        }
        public synchronized void unlock() {
            locked = false;
            notify();
        }
    }
```

5.7 Translation of Serializable Threads

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 `SerializableThread` and `SerializableThreadGroup`, 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 `SerializableThread` 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 `SerializableThread` wrappers. Then `start()` method of the underlying thread is called, which then calls the `run()` method of the `SerializableThread` wrapper. The `run()` method checks the stack of the `SerializableThread` 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.

5.8 Synchronization for Multiple Threads

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:

```
semaphore.acquire();
if (pc == i)  {
```

```
        pc++;
        stmt;
    }
    semaphore.release();
```

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.

5.9 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.

```
semaphore.acquire();
if (pc == i) {
    pc++;
    stmt1;
    stmt2;
    stmt3;
}
semaphore.release();
```

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 ensuring 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 (waitingWriters) {
    semaphore.release();
    semaphore.acquire();
}
```

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.

Chapter 6
New Constructs

6.1 Previous Translations

Translation mechanisms for the conditional and loop statements have also been implemented. In this section, translations scheme for the `if-else` and `while` statements will be explained first. The implementaion of `if-else` and `while` statements had been developed by [6].

The strongly mobile code to be translated is shown below:

```
if (cond) stmt1; else stmt2;
```

The translated weakly mobile code is shown below. thisIf, thisWhile, thisFor and thisDo are the translations for if, while, for and do-while statements respectively. Statement stmt' is translation of statement stmt in the strongly mobile code. inpc(stmt) denotes program counter value before execution of stmt and outpc(stmt) denotes program counter value after the execution of stmt.

```
if (((pc > inpc(thisIf)) && (pc < outpc(stmt1')))
       || (pc == inpc(thisIf)
       && cond) ) {
    stmt1';
}
if (pc == inpc(thisIf))
    pc = outpc(stmt1') + 1;
if ((pc >= inpc(stmt2')) && (pc < outpc(stmt2'))) {
    stmt2';
}
if ((pc == outpc(stmt1'))
    || (pc == outpc(stmt2'))) {
```

```
    pc = outpc(stmt2') + 1;
}

while (cond) { stmt' }

while ((pc >= inpc(thisWhile))
        && (pc <= outpc(stmt')))  {
    if (pc == inpc(thisWhile) && ! cond) {
        pc = outpc(stmt') + 1;
        break;
    }
    stmt';
    if (pc == outpc(stmt')) {
        pc = inpc(thisWhile);
    }
}
```

6.2 `for` Loop Translation

The translation for the `for` loop in strongly mobile code

```
for (InitialStmt; Cond; AfterStmt)  {
    stmt;
}
```

is given below

```
for (InitialStmt; (pc >= inpc(thisFor) && pc <= outpc(stmt)); AfterStmt) {
    if (pc == inpc(thisFor) && !(cond)) {
        pc = outpc(stmt) + 1;
        break;
    }
```

```
        stmt;
        if (pc == outpc(stmt)) {
            pc = inpc(thisFor);
        }
    }
```

`InitialStmt` and `AfterStmt` are from the untranslated `for` loop. `thisFor` is the translated version of the `for` loop. The loop is executed as long as the value of the program counter variable is between `inpc(thisFor)` and `outpc(stmt')` inclusive. If the condition is false, the value of the program counter after the execution of the translated statement is incremented by one and the loop is ended by the `break` statement.

6.3 `do-while` Translation

The translation for the `do-while` loop in strongly mobile code

```
do { stmt; } while (cond);
```

is given below

```
do {
    if (pc >= inpc(thisDo) && pc < inpc(thisWhile)) {
        stmt;
    }
    if (pc == inpc(thisWhile) && ! cond) {
        pc = output(stmt) + 1;
        break;
    }
    stmt;
    if (pc == outpc(stmt)) {
        pc = inpc(thisWhile);
    }
```

```
} while (( pc >= inpc ( thisWhile )) && (pc <= outpc(stmt)));
```

`thisDo` is the translation of the `do` statement. The translation ensures that the `stmt'` is executed at least once. if the condition is false after the execution of the statement for the first tim, program counter variable after statement execution is incremented by one and the loop ends due to the `break` statement. In case the condition is true, `stmt'` is executed again. The program counter variable is set to value of the program counter before the execution of `thisWhile` statement.

Chapter 7
Translation Schemes for `run1()`

The strongly and different versions of weakly mobile codes have been designed to ensure that optimization at runtime is minimal as possible. In order to achieve minimal optimization, the following

1) The value for bound integer variable n is assigned by a function call.

```
int n = bound();
```

2) The integer variables x, x1, x2...x9 are assigned values by random functions.

```
// Inititalize variables x, x1...x10 randomly.
x = random.nextInt(100);
x1 = random.nextInt(100);  x2 = random.nextInt(100);
x3 = random.nextInt(100);  x4 = random.nextInt(100);
x5 = random.nextInt(100);  x6 = random.nextInt(100);
x7 = random.nextInt(100);  x8 = random.nextInt(100);
x9 = random.nextInt(100);  x10 =  random.nextInt(100);
```

3) The variables x, x1, x2...x9 are used as a variables in an arbitrary function which is invoked after the for loop.

```
// Define a method that uses variables x, x1, x2...x10
// for some arithmetical operation.
int func(int x, int x1, int x2, int x3, int x4,
         int x5, int x6, int x7, int x8, int x9, int x10) {
    int y = x + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
    return y;
}
```

The for loops in the strongly and weakly mobile code invoke the method below:

```
//Define  method  bar() which is called from the function foo()
void bar() {
    int x, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10;
    int n;
    //Initialize variables x, x1, ...x10, n
    x = 1;  x1 = 2;  x2 = 3;  x3 = 5;  x4 = 7;  x5 = 11;
    x6 = 13;  x7= 17;  x8 = 19;  x9 = 23;  x10 = 29;
    n = 10;
    for (int i = 0; i < n; i++) { //Loop n times
        //Add variable x variables to x1...x10
        x1 = x1 + x;  x2 = x2 + x;  x3 = x3 + x;
        x4 = x4 + x;  x5 = x5 + x;  x6 = x1 + x;
        x7 = x1 + x;  x8 = x1 + x;  x9 = x1 + x;
        x10 = x10 + x;
    }
}
```

7.1 Strongly Mobile Code

The code fragment below is the strongly mobile code. The foo() function has a for loop which iterates for specific number of times. The bar() function is invoked during each iteration.

```
void foo() {
    ...
    ...
    for (int count = 0; count < 101; count++) {
        long start = System.nanoTime();
        for (int i = 0; i < n; i++) {
            // Add variable x variables to x1...x5
```

```
            x1 = x1 + x;  x2 = x2 + x;  x3 = x3 + x;
            x4 = x4 + x;  x5 = x5 + x;
            bar();   // bar() function is called.
            // Add variable x variables to x6...x10
    x6 = x1 + x;  x7 = x1 + x;  x8 = x1 + x;
            x9 = x1 + x;  x10 = x10 + x;
}
        System.out.println(System.nanoTime() - start);
        // Call function func()
        func(x, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
    }
}
```

7.2 Weakly Mobile Code 1

Code fragment below is the translated weakly mobile code for the strongly mobile code above. There is a unlocking-locking pair after every program counter update and statement execution. The bar() and run1() functions are called before the lock is released.

```
class Foo extends Frame { // foo() function translated into Foo frame.
    ...
    void run() throws InterruptedException { //Begin run function
        ...
        for (int i = 0; pc >= 0 && pc <= 10; i++) { //Beginning of for loop
            if (pc == 0 && !(i < n)) {
                pc = 11;
                break; // Loop ends
            }
            semaphore.release();
            semaphore.acquire(); //Lock-unlock pair
```

```
if (pc == 0) {
    pc++;
    x1 = x1 + x;
}
...
semaphore.release();
semaphore.acquire();
if (pc == 5) {
    pc++;
    bar(); // Push Bar frame on run time stack
    // run1() function is after lock is acquired and
    // before the lock is released
    // Pop the bar frame from run time stack and
    // execute run() function.
    run1();
}
semaphore.release();
semaphore.acquire();
...
if (pc == 9) {
    pc++;
    x10 = x10 + x;
}
if (pc == 10) {
    pc = 0;
}
semaphore.release();
semaphore.acquire();
```

```
        }
    }
}
```

7.3 Weakly Mobile Code 2

Code fragment below is the translated weakly mobile code for the strongly mobile code above. The statements are grouped into a logical block.There is a unlocking-locking pair after every program counter update and logical block execution. The bar() function is called before the lock is released and run1() function after the release.

```
class Foo extends Frame { // foo() function translated into Foo frame.
    ...
    void run() throws InterruptedException { // Begin run function
        ...
        for (int count = 0; count < 101; count++) {
            pc = 0;
            long start = System.nanoTime();
            for (int i = 0; pc >= 0 && pc <= 2; i++) {
                if (pc == 0 && !(i < n)) {
                    pc = 3;
                    break;
                }
                semaphore.release();
                semaphore.acquire();
                if (pc == 0) {
                    pc++;
                    x1 = x1 + x;  x2 = x2 + x;  x3 = x3 + x;
                    x4 = x4 + x;  x5 = x5 + x;
                    bar(); // Push Bar frame on run time stack
```

```
                }
                semaphore.release();
                // Pop  Bar frame from the stack and execute run() function
                run1();
                semaphore.acquire();
                if (pc == 1) {
                    pc++;
                    x6 = x6 + x;  x7 = x7 + x;  x8 = x8 + x;
                    x9 = x9 + x;  x10 = x10 + x;
                }
                if (pc == 2) {
                    pc = 0;
                }
                semaphore.release();
                semaphore.acquire();
            }
            System.out.println(System.nanoTime() - start);
            // Invoke func() function
            func(x, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
        }
    }
}
```

7.4 Weakly Mobile Code 3

Code fragment below is the translated weakly mobile code for the strongly mobile code above. The statements are grouped into a logical block.There is a unlocking-locking pair after every program counter update and logical block execution. The bar() function is called before the lock is released and run1() function after the release.

```
class Foo extends Frame { // Run time stack Foo
    ...
    void run() throws InterruptedException { // Begin run function
        ...
        for (int count = 0; count < 101; count++) {
            pc = 0;
            long start = System.nanoTime();
            for (int i = 0; pc >= 0 && pc <= 1; i++) {
                if (pc == 0 && !(i < n)) {
                    pc = 2;
                    break;
                }
                semaphore.release();
                semaphore.acquire();
                if (pc == 0) {
                    pc++;
                    x1 = x1 + x;  x2 = x2 + x;  x3 = x3 + x;
                    x4 = x4 + x;  x5 = x5 + x;
                    // Push Bar frame on run time stack
                    bar();
                    // Pop Bar frame from the stack and execute run()
                    run1();
                    x6 = x6 + x;  x7 = x7 + x;  x8 = x8 + x;
                    x9 = x9 + x;  x10 = x10 + x;
                }
                if (pc == 1) {
                    pc = 0;
                }
```

```
                semaphore.release();
                semaphore.acquire();
            }
            System.out.println(System.nanoTime() - start);
            func(x, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);
        }
    }
}
```

However in the above scheme, statements after the bar() method in the logical block might not get executed if the agent relocates during the execution of activation record of bar() method. In that case, after the execution of the activation record of bar() method is completed, execution in the callee method foo() resumes from the last program counter update instead.

Chapter 8
Checkpointing

8.1 Introduction To Checkpointing

In last the few years, algorithms in the Machine Learning have been proposed that need more powerful machines to perform complex computations. In order to achieve faster computational speeds, the number of nodes in the parallel computing machines has been increasing drastically. With increasing number of nodes, the likelihood of failures has increased resulting in the Mean Time Between Failures (MTBF) getting reduced from hours to minutes. It is predicted that in Exascale systems that failures will occur between 3 to 26 minutes [27]. In case of a node failure, the computation that was being performed at node needs to be restarted all over again from beginning in order to achieve correct results which results in longer time for executing a computational task. In order to overcome this drawback, various fault tolerant techniques such as redundancy and checkpointing have been proposed. Redundancy involves execution of the replica of a task on another node. The drawback of redundancy is that extra resources are required and overhead is incurred due to redundant communication. Checkpointing [10] involves writing the execution state of the executing task to a file or an external storage device at specified intervals. The executing task is interrupted to facilitate migration to the file or storage device where the execution state is saved. In case of a node failure, the task reads the execution state and resumes execution from the last checkpoint instead of restarting execution from the beginning. Another useful application for checkpointing is debugging analysis. Modern debugging tools incur large overheads when trying to detect a program error. However, the error programs can also be detected by using checkpointing tools. The use, however, checkpointing incurs overhead due to reading to and writing from the files. In the current Petascale systems, significantly large checkheads cause I/O bottlenecks resulting in 25 percent of the overhead [25]. If there are 1000 processes of

1GB RAM, that will require I TB of space that degrades the performance as well as costs more memory space. In Exascale systems, more nodes imply shorter MTBF resulting in more checkpoint overhead due to more frequent failures. Attempts are being made to reduce the checkpoint overhead and also decrease the memory space occupied by the checkpoint in the storage devices.

8.2 Checkpointing Mechanisms

In the High-Performance systems, it is imperative that there should be a consistent global checkpoint even though the different nodes might be at different stages of execution. Since the global checkpoint is composed of local checkpoints, maintaining synchronization among all the local checkpoints is a tedious task. In case of single node failure, programs running on all the other nodes also have to restart from the previous checkpoints.This "domino effect" because of single node failure causes significantly large overhead. To eliminate this drawback, three different types of checkpointing mechanisms have been developed.

1) Uncoordinated checkpointing involves checkpointing by programs running on different nodes whenever it is possible to do so. This results in a reduced overhead but obtaining a consistent global checkpoint becomes more complicated leading to a more likelihood of a domino effect.

2) Coordinated checkpointing involves synchronized checkpointing by different nodes, thus enabling a consistent global checkpoint. the advantage of this technique is that it eliminates the possibility of the domino effect, thereby incurring less overhead than uncoordinated checkpointing. However, internal synchronization required in order to maintain a consistent global checkpoint results in significant overhead. For the synchronization overhead reduction, protocols such as non-blocking checkpointing coordination protocol has been proposed. DMTCP uses checkpointing based on synchronized clocks for reducing the overhead.

3) Communication-induced checkpointing (CIC) or Message Induced Checkpointing involves local and forced checkpointing. In local checkpointing, the local state of the program is

written on a persistent storage device. In forced checkpointing, protocols force the program to perform an additional checkpointing. There is an overhead reduction because local checkpointing can be performed when size of the state is small. However frequent forced checkpointing and messages piggybacking data produce significant overhead.

8.3 Techniques To Reduce Checkpoint Overhead and Memory Space

Various checkpointing techniques have been proposed to reduce the overhead and increase the efficiency. Data compression [23] is used to reduce the checkpoint overhead. However, if there is a memory overhead if a large amount of data needs to be compressed. Additionally, time overhead is incurred due to data decompression. In concurrent checkpointing, the process continues execution while the execution state is saved on the buffer. The saved data is then transferred to an external storage device. Incremental checkpointing [21] saves only. the part of the program that has been modified since the last checkpoint resulting in a reduced overhead. Copy-on Write method involves fork of child process by the parent process.The parent process continues execution without getting interrupted while the child process does checkpointing. However, spawning of new processes results in performance degradation if the size of the checkpoint data is large. Solid State Disk(SSD) memory has also been suggested to reduce the checkpoint overhead because it has lower access times and latency cycles. However, reading and writing data cannot be done more than a specified number of times. Fusion-io ioDrive card is another technology that has been suggested to reduce the overhead. It is based upon NAND flash-based solid-state technology and provides more bandwidth. [9] developed a technique called data deduplication that reduces the memory space occupied by checkpoint.

8.4 Levels of Checkpointing

Checkpointing is performed at three different levels:

1) User-level checkpointing makes use of user libraries for checkpointing implementation. The application programs are modified, compiled and then linked to the library, hence this

technique might not transparent to the users. The drawback of the technique is that parallel programs and shell scripts might not be checkpointed due to inaccessibility to the system files. Esky, Condo and libckpt are examples of user-level checkpointing. Esky currently works under Linux 2.2 and Solaris 2.6. DMTCP (Distributed Multithreaded Checkpointing) [2] implements transparent user-level checkpointing in distributed applications. No modifications are required in the application or the operating system. System calls (fork and dump) are used for creating core dump for the restoring the application state.

2) Application level checkpointing implementation involves encapsulating the checkpointing functionality within the application code. This technique is platform independent but is not transparent to the users. The size of the checkpoint in application checkpointing can be comparatively smaller. Another drawback is that the user needs to have a good understanding of the checkpointing functionality. Also, there is performance degradation in terms of scalability. Cornell Checkpoint Compiler is based on application level checkpointing.

3) System level checkpointing is of two types, hardware and kernel. The checkpointing interval is controlled by a system parameter. It is transparent to the user and the application code does not require any modification. System-level checkpointing can also implement preemption in which Checkpointing is performed case before a node is highly likely to fail soon. Preemption results in more efficient batch scheduling as the number of idle cycles are reduced. The drawback of system level checkpointing is that kernel source code might not be available to the user.Also, the kernel level checkpointing is not platform independent. The most popular form of system-level checkpointing is Berkeley Lab Checkpoint/Restart (BLCR), a checkpoint/restart implementation designed for Linux clusters. by the Future Technologies Group at Lawrence Berkeley National Lab. CRAK and Zap are also examples of system level checkpointing. In Hardware level checkpointing, customization of the cluster for checkpointing is done using the digital hardware. This technique is user transparent.

SWICH is an example of this technique. Hardware checkpointing can also be implemented using FPGA but with an additional cost of hardware.

8.5 Diskless and Multilevel Checkpointing

The most basic form of checkpointing is the single level checkpointing in which the checkpoint data is written on a parallel file system (PFS). It is called single level checkpointing because there only one type of storage device used. However, in Exascale computing, the number of nodes will increase drastically which implies more node failures. Due to the large size of checkpoint data and limited bandwidth of the PFS, significant overhead will be incurred due to I/O [18, 26]. PFS techniques such as Panasas [34], GPFD and Lustre has higher efficiency but incur significant overhead. Diskless and multiple checkpointing techniques have been developed eliminate the shortcoming of single point checkpointing. The advantage of diskless checkpointing [22] technique is that it eliminates the need of using PFS for writing checkpointing. This is kind of checkpointing distributes the encoded redundant data across the in-storage devices like main memory and SSD. However, the redundant checkpoint data stored in in-storage devices can be lost in case of a node failure and the process cannot restart from the point where the In the multilevel checkpointing technique [12] [32], there are multiple levels of storage device where the nodes can write the checkpoint data. The most resilient but slowest nodes checkpoint to the parallel file systems whereas less re-salient but faster nodes checkpoint on external storage devices such as Flash, RAM or a disk. Frequent checkpointing external storage with lower latency devices result in lesser overhead than the single point checkpointing. [4] have integrated topology based Reed-Solomon encoding in a three level multilevel checkpointing scheme. The technique is implemented using Fault-Tolerance Interface and the encoding time is hidden using Fault-Tolerance dedicated per node. Large-scale evaluation was performed using SPECFEM3D on TSUBAME2.0 that showed FTI imposed only 8 per cent of checkpoint overhead at a speed of over 0.1 Petaflops and checkpoint time of 6 minutes. [17] developed the Scalable/Restart library to evaluate

multilevel checkpointing on a large scale system. The efficiency of the technique has been demonstrated by using a probabilistic Markovian model and the results of an application.

8.6 Our method of Checkpointing

The method developed uses the translation scheme to generate code for the checkpointing. The translator produces serializable threads which are to be executed on the processor cores. The serializable threads, along with runtime stack, can migrate at the desired intervals of time to the memory disks where the execution state can be written. In the case of a processor failure, the thread migrates to the corresponding memory disk and reads the execution state. The aim is to make the task of the programmer who wants to write checkpointing code as simple as possible. This technique is user-friendly as the programmer only has to write the code in the t strongly mobile form and the translator, instead of the programmer, produces the checkpointing functionality and features as the part of the generated code.

```
public class CheckPointer extends Thread {
    enum Type {
        MANUAL,  // manual checkpointing
        TIMED    // automated checkpointing
    }
    Type type;
    int period; // Time period
    File file;
    FileOutputStream fos;
    FileInputStream fis;
    ObjectOutputStream oos;
    ObjectInputStream ois;

    File outFile;
    FileWriter fileWriter;
```

```
BufferedWriter bufferedWriter;


double start;   // Starting  and ending time in ns.
double end;
private static Object lock = new Object();


// Checkpointer function for manual checkpointing.
CheckPointer() throws IOException {
    file = new File("a.txt");
    fos = new FileOutputStream(file);
    fis = new FileInputStream(file);
    oos = new ObjectOutputStream(fos);
    ois = new ObjectInputStream(fis);
    outFile = new File("outputFile");
    fileWriter = new FileWriter(outFile);
    bufferedWriter = new BufferedWriter(fileWriter);
    type = Type.MANUAL; // Declare type for manual checkpointing.
    period = 0;
}


//Checkpointer function for automated checkpointing.
CheckPointer(int time) throws IOException {
    file = new File("a.txt");
    fos = new FileOutputStream(file);
    fis = new FileInputStream(file);
    oos = new ObjectOutputStream(fos);
    ois = new ObjectInputStream(fis);
    outFile = new File("outputFile");
```

```
    fileWriter = new FileWriter(outFile);
    bufferedWriter = new BufferedWriter(fileWriter);
    type = Type.TIMED; // Declare type for automated checkpointing.
    period = time;
}

// Linked list of serializable threads
LinkedList < T > threads = new LinkedList();

void register(T t) {
    threads.add(t); // add t to list threads
}

void checkPoint() {
    synchronized(lock) {
        for (T t: threads) {
            try {
                start = System.nanoTime();
                oos.writeObject(t);
                System.out.println("CheckPoint written");
                //System.out.println(System.nanoTime() - start);
                //oos.close();
            } catch(IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
```

```
            }

            for (T t: threads) {
                if (t.pc == -1) {
                    try {
                        T readThread = (T) ois.readObject();
                        System.out.println("CheckPoint read");
                        readThread.restart(readThread.index(), this);
                    } catch(ClassNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    } catch(IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

The translation implementation developed is of two types

1) Manual Checkpointing Checkpointing can be inserted manually at any location within the generated checkpointing code.

2) Automated Checkpointing Checkpointing is done by a separate thread which is running in parallel with the serializable thread encapsulating the weakly mobile code. The checkpointing thread writes the execution state at specified intervals.

Chapter 9
Measurements

To indicate the overhead of the translation mechanism and the potential for optimizations, the results of manual optimizations and measurements had been performed in prior work [8]. 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. Simple manual optimization techniques were used to improve the performance of the translated agents. These were: 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 9.1.A major contributor to the poor running times of the recursive benchmark programs was the garbage collector that runs several times a second during their execution.

[8] performed further optimzations on the Linpack benchmark, a matrix multiplication implementation. The inner-most loop of Linpack is inside a dot-product method. This method was manually inlined, and measured execution time with the inner-most loop untranslated, and with the translated loop unrolled. The running time comparisons are presented in Table 9.3 .

TABLE 9.1. 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

TABLE 9.2. Memory utilization of Strongly Mobile Agents and the Aglets (MB)

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 9.3. 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

TABLE 9.4. Memory utilization of Optimized Strongly Mobile Agents for Linpack (MB)

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 9.5. Migration Time for Single-threaded Strongly Mobile Agents and Aglets (ms) — Linpack Benchmark

Number of stack frames	Agent pack time	Agent dispatch time	Aglets dispatch time
1	12	8418	1105
2	12	5200	1078
3	6	5153	1060

TABLE 9.6. Migration Time for Multi-threaded Strongly Mobile Agents and the Aglets (ms) — 5 frames on main thread stack, 2 frames on other threads' stacks

Number of threads	Agent pack time	Agent dispatch time	Aglets dispatch time
1	12	8418	1105
2	12	5200	1078
5	6	5153	1060

The overhead of migrating agents depends on the amount of state that the agent requires to carry along with itself. This was 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 was the time required for the translated agent to execute the Aglets dispatch method. compare this against the time required for the transfer of the simple benchmark Aglet. Agents and Aglets were 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 Table 9.5. Similarly, the dependence of the migration cost of a multi-threaded agent, on the number of threads is shown in table Table 9.6 .

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 (over-clocking), and Intel Speed Step (CPU throttling), and turned off all network interfaces, the X

TABLE 9.7. Average execution time for one lock-unlock pair.

Locking Mechanism	Time (ns)	Standard Deviation(ns)
Semaphore	8.25	0.009
ReentrantLock	6.39	0.062
ReentrantReadWriteLock (Read Lock)	10.90	0.085
ReentrantReadWriteLock (Write Lock)	7.31	0.076
AtomicBool (as guard for lock)	1.68	0.040
AtomicInt (as guard for lock)	1.34	0.016

window system, and unnecessary background processes. We used Java Version 1.7.0_21 and ran the measurements on the Java server VM with the command line options `-Xbatch` and `-XX:CICompilerCount=1` 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 9.7.

As our measurements show, the cheapest combination would be to use an atomic integer (the difference between them atomic integer and atomic Boolean is statistically significant) . All the error bars were less than 0.1ns (or whatever is the smallest error) with a confidence of 99%, so the differences between any pair of locking mechanisms are statistically significant. Therefore , Atomic integer is used 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.

Measurements were performed for the various run1() translation schemes and the average execution times computed are shown in table Table 9.8. The measurements show that WeaklyMobile2 scheme performs better than WeaklyMobile1. As expected, WeaklyMobile3 was efficient than WeaklyMobile1 and WeaklyMobile2.

Measurements were also performed our checkpointing scheme discussed in the paper. Average execution times were computed for both reading and writing for manual checkpointing

TABLE 9.8. Average execution time

Locking Scheme	Time	Standard Deviation
Strongly Mobile	18.54 ns	47.96 ns
WeakMobile1	309.84 ns	2.26 ns
WeakMobile2	222.99 ns	1.55 ns
WeakMobile3	211.12 ns	1.52 ns

TABLE 9.9. Average execution time for manual checkpoint scheme

Checkpoint Scheme	Time (ns)	Standard Deviation(ns)
Reading from disk	5365.99	3144.18
Writing to disk	4827.99	2721.57

(Table 9.9) and automated checkpointing Table 9.10. Linpack code was translated into weakly mobile code and the average execution times for Linpack without and with the various locking schemes which have been shownin table Table 9.11. There was a 34 per cent overhead reduction when the using the AtomicInteger as gaurd when compared to single step locking scheme. To further reduce the overhead, the number of statements executed were also taken into consideration along with the `AtomicInteger` guard. This mechanism showed a overhead reduction overhead by as much as 21 per cent. However, there was an insignificant change in the overhead irrespective of the increase in number of statements to check whether locking-unlocking needed to be performed. The reason can be attributed to the fact that comparison operation along with the AND operation is always performed.

TABLE 9.10. Average execution time for automated checkpoint scheme

Checkpoint Scheme	Time (ns)	Standard Deviation(ns)
Reading from disk	21843.73	1667.86
Writing to disk	22309.10	1319.45

TABLE 9.11. Average execution for translated Linpack (200 X 200)

Locking scheme	Time (ms)	Standard Deviation (ms)
Strongly Mobile	0.59	0.01
Weakly Mobile - Without locks	2.72	0.17
Weakly Mobile - Single Step	7.21	0.24
Weakly Mobile - Atomic Guard	4.72	0.15
Weakly Mobile - Atomic Guard with Count	3.93	0.14

Chapter 10
Conclusions

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 multithreaded 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 dissertation 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 have also developed a checkpointing techniques (both manual and automated) in which the user can generated checkpointing code in form of threads that can migrate from node to another. Our technique is user friendly since it eliminates the need for the user to have knowledge of checkpointing features and functionalities.

Chapter 11
Future Work

In our translation scheme for strongly mobile code, we have modified the syntax tree of Polyglot extension compiler so as to obtain a more optimized translated code. Our objective has been to reduce the overhead incurred due locking-unlocking. We have experimented with different locking mechanisms and schemes in order to achieve that objective. More work can be done on further optimization so as to achieve a better efficiency. A construct can be introduced which does not translate the section of a code. This eliminates the need of translating the section of code that incurs significantly less overhead than the translated one due to frequent locking and unlocking. Functions having only a few lines of code can be inlined instead of being called from another function, thus lowering the overhead incurred. An algorithm also needs to be developed that will determine where the logical block of statements will end. One suggestion is to end the block where a function with a significantly large overhead is invoked. We have also developed a technique in which the programmer can produce serializable threads that can migrate from one node to another. The serializable threads can be used for restarting an application in the Android systems when it powered on. This will reduce time to load the application on Android. The checkpointing implementation using serializable threads can also be applied to checkpointing in High Performance Computing. With the advent of Exascale Computers, the Mean Time Between Failures is likely to reduce and thus me efficient techniques of checkpointing are needed in order to reduce the overhead incurred due to reading and writing to the disk. We have developed a prototype that implements checkpointing in Java language. However, Java is not the language used in High Performance Systems. But a method similar to our technique can be used for checkpointing using commonly used High Performance System languages such as C++ and FORTRAN.

References

[1] Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In Jan Vitek, editor, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer-Verlag, 1996.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssìere. Communicating mobile active objects in Java. In Marian Bubak, Hamideh Afsarmanesh, Roy Williams, and Bob Hertzberger, editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643. Springer Verlag, May 2000.

[4] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[5] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. In *Software — Practice and Experience*, pages 355–394, 2002.

[6] Arjav J. Chakravarti and Gerald Baumgartner. Self-organizing scheduling on the Organic Grid. *Int. Journal on High Performance Computing Applications*, 20(1):115–130, 2006.

[7] Arjav J. Chakravarti, Gerald Baumgartner, and Mario 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.

[8] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald 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, October 2003.

[9] Zhengyu Chen, Jianhua Sun, and Hao Chen. Optimizing checkpoint restart with data deduplication. *Sci. Program.*, 2016:10–, June 2016.

[10] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[11] Stefan Fünfrocken. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In Kurt Rothermel and Fritz 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.

[12] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 251–255, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[13] Robert S. Gray, David Kotz, George Cybenko, and Daniela Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[14] Torsten Illmann, Tilman Krueger, Frank Kargl, and Michael 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.

[15] Danny B. Lange and Mitsuru Oshima. Mobile agents with Java: the Aglets API. *World Wide Web Journal*, 1998.

[16] Danny B. Lange and Mitsuru Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.

[17] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, MSST '07, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[20] Brian Peterson, Gerald Baumgartner, and Qingyang 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.

[21] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[22] James S. Plank, Youngbae Kim, and Jack J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 351–, Washington, DC, USA, 1995. IEEE Computer Society.

[23] James S. Plank and Kai Li. Ickp: A consistent checkpointer for multicomputers. *IEEE Parallel Distrib. Technol.*, 2(2):62–67, June 1994.

[24] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori 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.

[25] Bianca Schroeder, Garth Gibson, and Garth A. Gibson. Understanding failures in petascale computers. 2007.

[26] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[27] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[28] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori 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.

[29] Takashi Suezawa. Persistent execution state of a Java virtual machine. In *Java Grande*, pages 160–167, 2000.

[30] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An overview of the NOMADS mobile agent system. In Ciarán Bryce, editor, *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, 13 June 2000.

[31] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre 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.

[32] Nitin H. Vaidya. A case of multi-level distributed recovery schemes. Technical report, College Station, TX, USA, 2001.

[33] Xiaojin Wang, Jason Hallstrom, and Gerald 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.

[34] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

Vita

Arvind Saini is from New Delhi, India. He obtained his Master's degree in Computer Science from Midwestern State University, Texas (USA) in May, 2008. His research area at Midwestern State University was in Software Engineering. In 2010, he joined Louisiana State University for the doctoral program in Computer Science and will obtain the Ph.D. degree in May, 2018. His research interests include optimizations for mobile agents for large clusters and desktop grids and parallel computing. Currently, he is exploring the use of the mobility translator for checkpointing applications. In addition, he authored a paper titled *An Optimizing Translation Framework for Strongly Mobile Agents*, which was published in the proceedings of KPS 2015 in Vienna, Austria.