EXECUTION CONTEXT MIGRATION WITHIN A STANDARD JAVA VIRTUAL MACHINE ENVIRONMENT

ANDREW DORMAN

Supervisor: DR. CASPAR RYAN

Honours Thesis

School of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

October, 2003

Abstract

This paper proposes and evaluates a mechanism for preserving and restoring the execution state of Java applications. A framework is developed, utilising the functionality provided in the Java Platform Debugging Architecture to locate the exact instruction of computation, at the time of program suspension. With the aid of bytecode modification, processing is resumed at the point of suspension, without repeating executed statements. This framework exhibits a low object code growth rate, between 4% and 17%, a substantial reduction in comparison to existing systems. Additionally, the execution overhead placed upon running applications is a minimal 7%. This is in contrast to the cost of state capture and restoration, which incurs a substantial performance penalty of approximately one second per stack frame, and thus requires future work.

Contents

List of Figures

1 Introduction

Computing in today's society is no longer strictly confined to the office. Employees who are travelling to meet with customers or performing field work often switch between desktop, laptop and personal digital assistant (PDA). The boundaries of computing are further blurred as personal organisers merge with mobile phones and fridges come with Internet capabilities [LG03]. The *Next-Generation Internet (NGI)* [MU01] aims to integrate the Internet and its resources into everyday life, making it as common place as plumbing and electricity [IBM03]. Through the use of mobile and non-mobile devices, NGI aims at providing access to services and information wherever a user may be.

While NGI sees mobile devices playing an active role in the delivery of web resources, there is little reason why this shouldn't be extended to standard applications. Moving from desktop to laptop or PDA should be no more limiting to the current session of a program then switching between graphical windows [BC95]. Mobile Objects hold the potential to deliver this flexibility, transporting an application from host to host without the loss of the current session. Such a system would be required to operate across multiple forms of hardware, with varying processing power and instruction sets. It is unreasonable financially and in terms of production time for developers to produce versions of software for every possible hardware type that may be encountered. A language that can operate on multiple platforms such as Java is required.

The Java programming language has been designed with portability in mind. Source code is compiled into an intermediate form called *bytecode*, unlike C and C++ which are transformed into native machine code. The bytecode is then interpreted into an executable form by a *Java Virtual Machine (JVM)*, in which all Java programs are run. This allows developers to write and compile their applications once, and have them execute on any platform implementing a JVM. This ability to operate across heterogeneous environments, makes Java highly suitable for implementing distributed or mobile applications.

The JVM is designed to act as a layer of abstraction between the application and the host operating system, defining a standard set of operations and functions available to a Java application. Modifying the JVM to incorporate additional functionality compromises the portability of the Java framework. Systems that require a *non-standard* JVM may only be deployed on hosts implementing the vendor specific interpreter, removing an applications ability to operate in a heterogeneous environment.

While Java provides support for object mobility through serialization[1], dynamic class loading and Remote Method Invocation (RMI), it does not provide a framework for the migration of execution state. Current implementations of the Java Virtual Machine do not allow the serialization of threads [Mic03b][FO01], leaving programmers to implement their own software solutions for state preservation.

This project investigates using the Java Platform Debugging Architecture (JPDA) (section 2.4) to capture and restore the execution state in the strong migration[2] of mobile objects. Due to Java's native support of transport protocols (eg. sockets and RMI), this project does not investigate frameworks for the transfer of objects, but focuses on the migration of the execution context.

This paper is organised as follows. Section Two explores object mobility, reviewing existing mobility systems, and outlining the Java Platform Debugging Architecture. Section Four provides details of

[1] For more details on Serialization see section 2.2

[2] Strong Migration transfers the mobile object and its execution state. See section 2.1 for a full description.

the methodology for the implemented framework. Section Five investigates the performance of the framework, while sections Six and Seven provide a summary and areas for future work respectively.

2 Literature Review

This section provides background material in relation to the work of this thesis. It details the key elements of Object Mobility and the constraints within the Java language. It defines the concepts of strong and weak migration, and the distinction between Mobile Objects and Mobile Agents (section 2.1). The examination of object transportation and marshalling in a Java environment (section 2.2), illustrates some of the core features making Java a desirable platform for object mobility.

This section additionally provides a review of existing Java mobility systems (section 2.3), identifying existing contributes and limitations that form the impetus for this research. An investigation into the Java Platform Debugging Architecture (section 2.4) is also carried out, outlining the underlying architecture of the proposed framework.

2.1 Overview

Mobile Objects are *Object Oriented Language* objects with the ability to have their execution suspended, migrate to a new host and resume processing. They provide a flexible approach to distributed computing, modularising systems and adaptive applications.

There exists a fine distinction between a Mobile Object and a *Mobile Agent*. Mobile Agents act *autonomously* to achieve a specific *goal* on the behalf of a person or system [WJ95]. Agents possess an agenda in their own right, moving between hosts in order to complete their task if they believe it will assist in accomplishing their goal. Mobile Objects, however, do not posses the autonomous task orientation, rather they are components of a larger system in which they participate. Lacking in self-determination, Mobile Objects are invoked on a needs basis, providing the services requested of them.

Object mobility falls into two categories: that of *Weak Mobility* and *Strong Mobility* [FPV98]. Weak mobility transfers the object and its data to the target host, while strong mobility additionally transfers the execution state. As the execution state is required for the migration of applications without loss of session, this project focuses on strong mobility.

Objects migrated using weak mobility must restart their execution afresh on the destination host, as there is no way to determine their point of execution before they were transferred. This approach may be suitable for maintaining fault tolerant systems [RH97], where it is acceptable for processes to restart and continue, but is not desirable for session dependant applications where execution state is important. Moving applications between hosts is of little benefit if users are required to re-establish the state of a program once transferred.

The implications for strong migration are not restricted to moving applications between devices. The ability to capture the execution state of threads allows for application check-pointing. Each checkpoint may be kept in non volatile storage and restored in the event of an application or host failure. Similarly, Mobile Objects are well suited to dynamic load balancing systems, moving as system demand and resources vary [DR98].

In order to implement strong mobility, the following must be obtainable: access to the object to be migrated, the object's state, the method call stack and the program counter [Fun98]. As Java is an interpreted language, programs are executed in the JVM. Stack frames and program counters for currently running threads are maintained in the JVM, which runs as one process on the host operating system. For this reason, it is not possible to obtain state information of a Java thread by accessing operating system processes, as this data is within the JVM. Java disallows dynamic inspection of the stack, preventing retrieval of the program counter, and does not currently support a mechanism for storing the execution point of a program.

2.2 Object Marshalling

Implementing either strong or weak mobility requires a means of marshalling objects together in a form which can be transmitted across a wire. Developers using C or C++ must write their own functions to perform such tasks due to the lack of support within the language. Java inherently provides such a mechanism in the form of object *Serialization*.

Serialization is a means whereby objects are written to a byte stream for storage or transportation. Primarily introduced to support the transportation of objects in RMI, it is now used for a variety of preservation applications. As part of the Java security framework, objects to be serialized must implement the *java.io.Serializable* interface, forcing the developer to explicitly allow serialization.

In an Object Oriented language, each object can be considered to possess an *object graph* of references [Har99]. For example, object x contains references to objects a, b and c, object a contains references to objects d, e and f and so on. During serialization, the graph is traversed, writing each object's fields to the byte stream. Should an object in the graph not implement the *Serializable* interface, serialization will fail. Such instances may be avoided by demarking object references or fields that are not to be serialized as *transient*, thus omitting them from the serialization procedure.

Upon deserialization, all references within an object must be valid and accessible. Serializaion attempts to achieve this by additionally writing the object graph to the stream. However, references to host bound resources such as streams and I/O devices may not be captured and transported, and for this reason require a more advanced method of adaptation. Proposed solutions include updating the binding to reference a remote location, or rebinding to an equivalent resource on the new host [BCM03]. As these issues fall beyond the scope of this paper, the standard serialization routine shall be used for this project. Further information on resource adaptation may be found in Section 6.

From the developer's view, the strength of serialization is not only found in writing an object to a byte stream, but also the ability to reconstruct objects with little programmer intervention. Java handles the creation and initialisation of objects and their fields as they are read from the byte stream, ready to continue operation.

While serialization captures the fields of an object, it does not obtain the values of local variables, nor the point of execution. A further constraint is that objects must be *passive* in order for serialization to take place. That is, an object may not be mid-method or processing instructions while being written. A notable exception to this rule is when an object serializes itself.

Java Object Mobility systems require objects to be marshalled, and this is achieved using serialization. The difference between various existing object mobility frameworks arise in capturing the point of

execution and storing local method variables. Section 2.3 outlines existing strong migration systems and examines the techniques employed.

2.3 Java Frameworks for Object Mobility

There are many Java Object Mobility systems which have implemented strong migration. Due to the lack of thread serialization and execution state capture in the Java language, these frameworks have implemented their own mechanisms for supporting execution state transfer. Some of these techniques include:

- Using a pre-processor to insert variables to *checkpoint* program execution. These variables are then evaluated on the target host to reconstruct execution [FO01][WTV02].
- Throwing Java Exceptions at migration time to capture execution state. The Exception Object is then transferred to the target host along with the object to reconstruct execution context [SMY99][Fun98][SSY00].
- Modifying the Java Virtual Machine to support the migration of execution context [BH99] [ARS97] [SM99].

While these mechanisms achieve strong migration, they suffer in terms of performance and or portability.

This section provides a review of three strong migration systems, demonstrating each of the categories listed above. These approaches have been selected for discussion due to the clarity of the documented structure, and sound evaluation of their respective frameworks. Section 2.3.1 explores JavaGo's exception handling technique, section 2.3.2 reviews AdJava's variable insertion and section 2.3.3 discusses Inria's JVM modification. In doing so, this section acknowledges the contributions of these systems, and identifies opportunities for improvement.

2.3.1 Exception Handling

With the aid of a pre-processor and code insertion, Sekiguchi's JavaGo [SMY99] utilises Java's exception handling capabilities to preserve execution state during migration. Methods capable of initiating object transfer are demarcated with the primitive *migratory*, and have *try* - *catch*[3] blocks inserted, capable of responding to *NotifyMigration* exceptions. Additionally, a variable acting as an artificial program counter is included and continuously updated during the program's session. For each method within the application, the pre-processor creates a *state object*, which is used to store local variable values at migration time.

Upon the decision to migrate, a *NotifyMigration* exception is thrown and caught within the current method, allowing the saving of local variables to the *state object*. The exception is propagated to the calling method, repeating the state saving process for each method on the stack and chaining subsequent state objects together.

[3] See [Mic03a] for more details on Java's Exception Handling

To facilitate state restoration on the destination host, the pre-processor transforms method signatures to accept a state object as an extra parameter, and is used to initialise local variables upon reconstruction. Methods are also transformed to include *switch-case* blocks encompassing individual statements, which upon evaluation of the execution point variable, can resume the program at the correct location. Control structures such as *for* and *while* require a more complex code insertion technique involving code *unfolding* and duplication. This results in an increased method size of $O(n^2)$ [SMY99, p. 219], where *n* is the depth of nested loops.

An extension to JavaGo deviated from source code transformation to that of bytecode modification [SSY00]. A similar framework of propagating an exception object is used during state capture, from which values are extracted for state restoration. Bytecode is inserted to support the population of the exception object as the object is loaded into the JVM. After migration, as the object is loaded, bytecode is inserted at the head of each method to extract local variable details from the transported exception object. This approach reduces execution time in comparison to JavaGo, but still places considerable performance overhead on the system, while achieving minimal reduction in code growth.

Sekiguchi's technique of state preservation incurs a heavy spatial and temporal performance reduction on the application, due to the pre-processor's code insertion. The approach described in section 4 of this project, removes the requirement for large amounts of tracking and restoration code.

2.3.2 Variable Insertion

The Automatic Distribution of Java Applications (AdJava) [FO01] system, developed at the University of Adelaide, relies upon code insertion to maintain execution state. By pre-processing the source code of a thread, a *level* variable is inserted and incremented after each statement in the *run* method, acting as an artificial program counter. Upon migration this variable is serialized with the object data, and transferred to the destination host. To resume execution, the *level* variable is evaluated through a series of *switch-case* statements, jumping to the current point of execution.

The variable insertion technique used by AdJava causes a substantial increase in code size, an approximate ratio of `1:3` [FO01, p. 74]. This increase is responsible for higher memory consumption as well as longer transfer and execution time.

A further limitation is that the artificial program counter is only inserted in the *run* method of the thread. Should the thread be suspended whilst in a called method, all computations completed since leaving *run* will be lost. This poses performance issues should migration be frequent, or if method calls are deeply nested, as more time is required to repeat calculations hence slowing program progression.

In using the JPDA to capture the state of a Mobile Object, this project aims to eliminate the requirement for large amounts of code insertion. Furthermore, this project intends to increase the granularity of state preservation, removing the need to repeat previously executed commands upon reconstruction.

2.3.3 JVM Modification

Systems based on exception handling and code insertion have all worked within the current confines of the JVM to transfer the execution state. However, researchers at INRIA's SIRAC Project [BH99]

and the University of Maryland's Sumatra Project [ARS97] have both modified the JVM to allow replication of, and access to, internal data structures, making thread state capture and transfer possible. INRIA's implementation provides a more recent and complete set of Java interfaces for development, and therefore is discussed in this section as an example of a state capture and restoration system.

To support the changes made to the JVM, SIRAC provides two new classes, *MobileThread* (which extends *Thread*) and *ExecutionEnvironment*. MobileThread calls *extractExecEnv* at migration time, storing the current execution state in a variable of type ExecutionEnvironment. *transferExecEnv* is called to perform the migration to the destination host, where *integrateExecEnv* creates a new thread initialised with the ExecutionEnvironment.

For security reasons, all methods and data of ExecutionEnvironment are private (with the exception of the array of class names, which is required by the class loader), and may only be accessed by MobileThread's *integrateExecEnv*. This avoids the possibility of illegal access to objects on the stack, whilst still providing for the needs of migration.

The advantages associated with JVM modification are primarily performance based, due to the lack of code insertion and complex restoration procedures seen in other implementations. Benchmarking performed by SIRAC shows that JVM modification can perform up to 13 times faster than Funfrocken's code insertion system Wasp [Fun98]. This is achieved at the sacrifice of portability, as discussed earlier.

By utilising the JPDA (section 2.4), this project intends to produce a framework that operates on a *standard* JVM, delivering performance comparable to JVM modification. While a reasonable assumption would expect there to be considerable overhead in running an application through a debugger, the JPDA uses *Full Speed Debugging* [Mic02]. This allows applications to run at normal speed, using only the interpreter. When an event of interest occurs, the debugger is activated allowing for further program querying (eg. viewing stack frames, local variables etc.).

2.4 Java Platform Debugging Architecture

While previous work has sought to overcome Java's state capture shortcomings by way of JVM modification or code insertion, this project uses the Java Platform Debugging Architecture to control the suspension and resumption of Mobile Objects, as well as obtaining the current point of execution and local variable information.

The Java Platform Debugging Architecture [Mic00] is a *'request-event'* based framework for the development of debugging applications. Debuggers are able to *'hook into'* programs running in the JVM by requesting notification of events fired by the framework, eg. variable modification, exception, stepping, method entry and method exit events. These events are placed in an *'EventQueue'*, from which the debugger may consume and further query the running program.

Once an event has been removed from the queue and delegated within the debugger, the JPDA permits the debugger to access information not normally available from Java applications. In particular the ability to obtain stack frames, variable values and the point at which execution was suspended are of particular interest to this project. The execution point obtained from the JPDA is in the form of a *code index*, which represents the location of an instruction in bytes from the beginning of the code segment.

While the JPDA provides access to information not normally available in standard Java applications,

it also limits access in other areas. In querying the JVM for running objects, the debugger is not able to obtain direct object references, limiting the ability to use an object in the same manner as a conventional program. All references and values returned by the JVM are *mirrors* of the actual values. For the most part this poses few problems, as it is possible to both *get* and *set* local instance variables. A notable exception is that of the current execution point within a stack frame. There exists a *location()* method for the retrieval of the execution point, but in an effort to enforce security constraints within the Java environment there is no accompanying *setLocation()*.

The JPDA implements a *redefineClasses()* method, allowing the definition of a class to be altered after it has been loaded into the JVM. This feature allows Integrated Development Environments to reflect changes made to classes immediately during a debugging session. There are however, limitations to the functionality of the redefinition. To ensure stable execution, current stack frames do not adopt any alterations to their method until they have been popped off the stack. Subsequent invocations of amended methods adhere to the new definition. Additionally, the redefinition does not allow for changes to the class schema (eg. added variables, object references, altered method signatures etc).

Figure 1 shows how the JPDA is logically divided into a *Back End* and a *Front End*, connected via a *Communication Channel*. These layers are implemented using the *Java Virtual Machine Debug Interface (JVMDI)*, the *Java Debug Interface (JDI)* and the *Java Debug Wire Protocol (JDWP)* respectively. The program being debugged is termed a *Debuggee*, while the application performing the debugging is the *Debugger*.

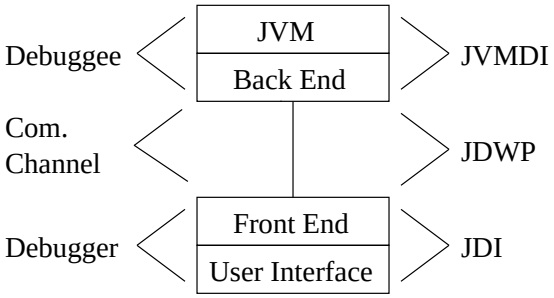

Figure 1: JPDA Overview [Mic00]

The JVMDI defines the methods and functionality a Virtual Machine provides in the JPDA. It contains a native code implementation of these methods, which developers with specialised debugging needs may choose to call directly from other native code applications.

The JDWP specifies the format for communications between the debuggee (back end) and the debugger (front end). It does not specify the exchange mechanism, only the format of the data.

The JDI contains high level interfaces, written in Java, that provide developers with convenient methods for debugger creation. The JDI utilises the lower layers of JPDA (JDWP and JVMDI), in order to obtain the required data and event notification.

The division of the JPDA makes remote debugging of applications possible, as the front and back ends may be on separate hosts. In addition the debugger and debuggee may be running different vendor's Virtual Machines and/or on different platforms.

Developers may choose to implement their product at any level of the JPDA. Debuggers not written in Java may connect directly to the JDWP for remote access to the JVM, or the JVMDI for local access

only. While it is recommended that applications use the JDI, developing at the JVMDI level may provide better support for more specific debugging requirements.

The JPDA is designed to be compatible with classes created with the standard options of a Java compiler, however not all debugging features may be available. Information such as variable names, source file and code line numbers are not included in classes by default. The compiler must be instructed to enclose these details for debugging purposes. Under Sun's JVM 1.4.1_02 compiler used for testing in this thesis, this is achieved by including *-g* as a command line argument, allowing the developer to specifying which information is to be included in the compiled class.

3 Project Aims

Based upon limitations identified in reviewed literature, this thesis aims to develop an Execution Context Migration Framework for the capture and restoration of the execution state of running applications. By utilising the Java Platform Debugging Architecture and making no modifications to JVM, this system shall be portable to all standard JVMs, allowing for operation in heterogeneous environments. In producing this framework, this project will answer the following questions:

1. Is it possible to capture and restore execution of running application using the JPDA?
2. What temporal impact does such a system have upon executing objects?
3. What are the temporal costs of state capture and restoration?
4. What is the magnitude of code growth of objects within such a framework?

4 Methodology

As outlined in previous sections, Java, from its inception has been designed to operate across multiple platforms, and is well suited for mobile applications due to its inherent support of RMI and sockets. However, the absence of a means to capture the execution state considerably restricts the development of strong Mobile Object systems. This limitation requires indirect solutions, which impedes efficiency in terms of execution time and portability.

This project has addressed these shortcomings by developing a framework that allows the suspension and reconstruction of Mobile Objects, without the need to insert large amounts of source code or modify the JVM. The JPDA allows the exact location of execution to be accessed and subsequently stored, along with the local method variables. The Mobile Object is then able to be rebuilt and resumed at the precise instruction before migration, continuing processing as normal. The framework allows Mobile Objects to operate in heterogeneous environments and utilise the advantages of strong migration. As this research is concerned with capturing and rebuilding the execution context of running objects, and not the particular means of transportation itself, such migration mechanisms have not been developed as part of this project.

The methodology is divided as follows: section 4.1 provides a high level overview of the framework, outlining the key components and functionality of the system. More in depth details of the

state capture and stack traversal process are given in section 4.2, with state restoration and bytecode modification explored in section 4.3.

4.1 System Overview

The developed system primarily consists of two threads of execution, an *External Monitor* utilising JDI debugging methods to control a Mobile Object, and a *Monitored Thread* in which Mobile Objects run. Additionally an *Internal Monitor* is required to operate in the Monitored Thread, providing support tasks that may not be accomplished via the External Monitor (see section 4.3 for more details). These entities are not generic object types, but rather implementation specific instance objects. Figure 2 details the relationships of the following system entities:

External Monitor Launched as a debugger, utilises JDI methods to control the loading, execution state capture and restoration of Mobile Object.

Monitored Thread The thread of execution containing the Internal Monitor and Mobile Object.

Internal Monitor Loads the Mobile Object and provides static methods for state restoration.

Mobile Object The Mobile Object and associated objects that are to be migrated.

Stack List A list of stack frames in existence at the time of migration. List entry information includes the execution point, local variables, current method name and signature.

The system begins with the External Monitor loading the Internal Monitor into the Monitored Thread, which in turn loads and activates the Mobile Object. When the External Monitor determines that it is time for migration, it suspends and serializes the Mobile Object. At this point it captures its execution state by querying the JVM for a list of stack frames, which are processed and stored in a *Stack List*. Together the Stack List and Mobile Object are migrated to a new host, where the External Monitor again loads the Internal Monitor. Here the Mobile Object is deserialized and the stack rebuilt, using the information contained within the Stack List.

In order for the serialization to take place, the Mobile Object must implement a *move()* method, which performs the serialization of itself. Calls to *move()* must be placed by the programmer at strategic locations throughout the code.

Previous works have restricted migration to occur at specific points, as determined by the programmer [SMY99][Fun98][SSY00]. Such implementations require the Mobile Object to move without choice at these locations. This project allows for migration to be initiated at any time, objects are not forced to relocate at each invocation of *move()*, rather the External Monitor initiates migration at its discretion. Ideally this would be achieved by the External Monitor calling *move()* via the JDI's *invokeMethod()*.

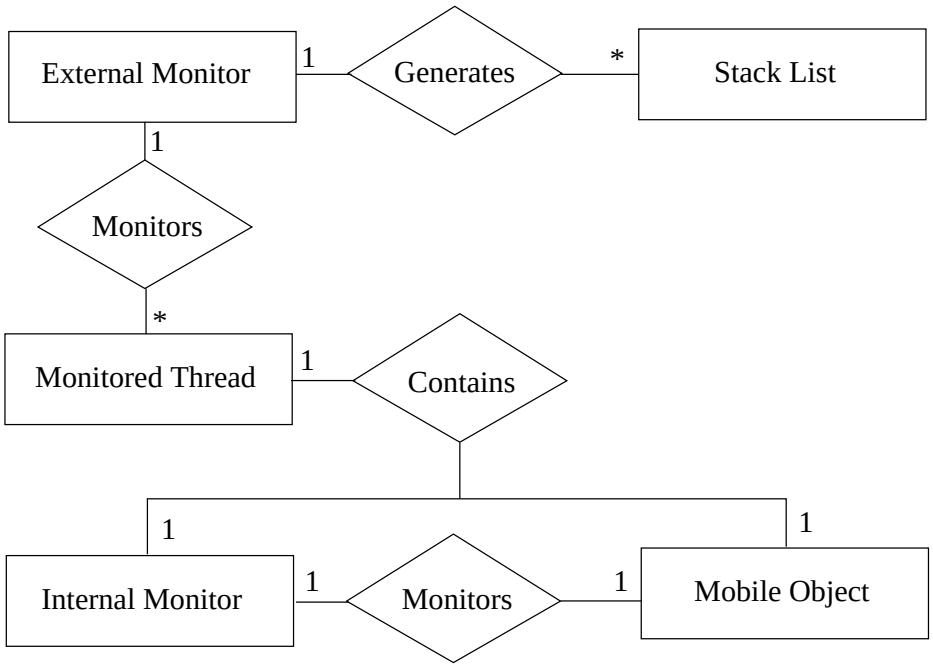

Figure 2: System Entity Relationship

As the Mobile Object is suspended by the External Monitor using JDI breakpoints mid-method, Mobile Object is not passivated, and is still considered to be active. As detailed in section 2.2, Java does not allow for the serialization of active objects, rather they must be passive, or actively executing the serialization themselves.

To circumvent this shortcoming two near identical versions of the Mobile Object are required. The first has no instructions contained within its *move()* method, while the second has a fully implemented set of serialization commands. During normal execution the empty version of *move()* is invoked, allowing the application to progress unhindered. To instigate migration the External Monitor utilises the JDI's *redefineClasses()* method, changing the definition for the MobileObject class to contain serialization instructions. At the next invocation of *move()*, the Mobile Object is preserved.

Positioning calls to *move()* should be done with great care. It is undesirable for long periods of time or code segments to pass between the External Monitor initiating migration and the next invocation of *move()*. Conversely calling *move()* too frequently would impede the application performance. Although not implemented as part of this project, a pre-processor may be used to add calls to *move()* at strategic locations, using heuristics for optimal placement as discussed in section 6. Currently it is the responsibility of the developer to place such method invocations, and thus determining the movement granularity.

In addition to *move()*, the Mobile Object must also implement *objectStart()*. This method is the first to be called by Internal Monitor, and is similar in nature to *run()* in a Java *Thread*. Its purpose is to begin the processing of the Mobile Object, and is used during state restoration to delay the onset of execution, as described in section 4.3.

4.2 Capturing Execution State

To implement strong migration both the object state and execution state must be captured and transferred to the destination host. As outlined in section 2.2, object serialization will only capture the object state. To achieve preservation of the execution state and local method variables, a technique where the method call stack is traversed, extracting variables and the code index for each frame is employed. This access to the run time method call stack is only available to debugging applications. The data collected from the stack is placed into the Stack List and migrated along with the Mobile Object to the destination host for reconstruction. Figure 3 outlines the process of creating the Stack List:

```
- Retrieve a list of stack frames for the current thread
- Create a new StackList
- For each stack frame
    - Create a new stackListItem
    - Extract and save the current code index to stackListItem
    - Extract and save the current method, method signature
      and class name to stackListItem
    - Create a new stackVariableList
    - For each local variable
        - Create a new stackVariable
        - Extract and save the variable name, value and type
          to stackVariable
        - Add stackVariable to stackVariableList
    - Add stackVariableList to stackListItem
    - Add stackListItem to StackList
- Store StackList
```

Figure 3: Pseudo Code for Stack Preservation

To remove the possibility of inconsistent state information, both the object serialization and stack preservation must occur without the Mobile Object continuing to execute instructions. It would be logical for the External Monitor to initiate migration by redefining the Mobile Object to contain a fully implemented *move()* method, setting a break point within *move()* allowing for the creation of Stack List, followed by the resumption and serialiation of the Mobile Object. Unfortunately, this approach is not feasible. Once a class has been redefined its local variables within the stack frames are not accessible, as the frame has been deemed *'obsolete'* by the JVM.

The alternate solution implemented in this project has Mobile Object invoking *move()* twice, as shown in figure 4 and on lines eight and twelve of figure 5. When the External Monitor initiates migration, it sets a break point for the currently empty *move()* method. Once this has been reached, the External Monitor creates a Stack List and proceeds to redefine Mobile Object's class. The current stack frame is popped off the method call stack using the JDI's *'pop()'*, exiting the empty move() method, followed by the resumption of the Mobile Object. The next instruction, the second call to *move()*, executes the serialization of Mobile Object ready for migration. This provides a means for both the object and execution state to be preserved without the risk of inconsistent state information. Figure 4 further illustrates the sequence of events during the execution state capture.

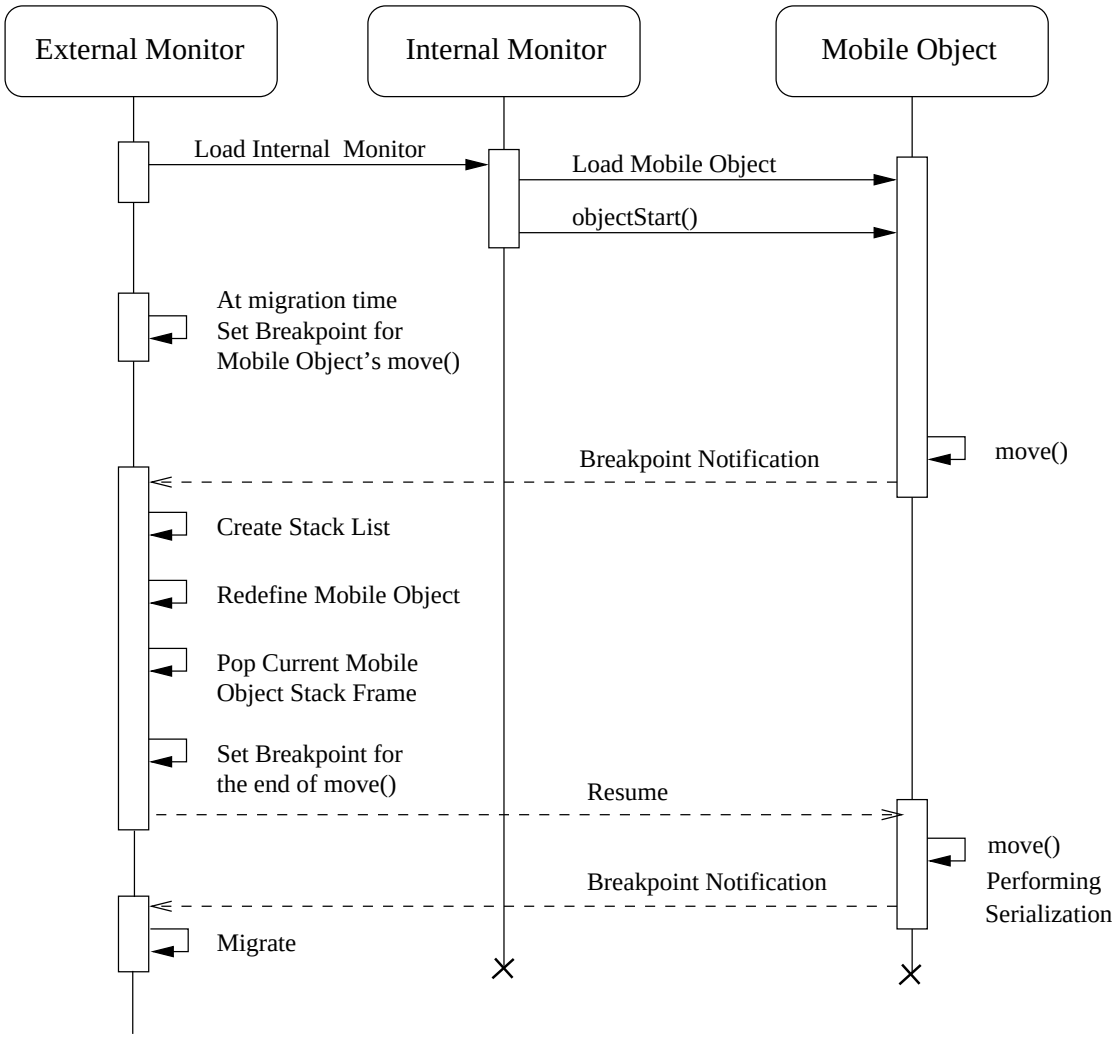

Figure 4: Execution State Capture Sequence

Figure 5 details an example of a Mobile Object. Features of interest include line one with the mandatory implementation of *Serializable*, lines eight and twelve illustrating the two calls to *move()*, and line seven demarking the end of variable declaration and initialisation by invoking *InternalMonitor.finishedVariables()*. Further details regarding this method call are provided in section 4.3.

4.3 State Restoration

In the absence of a JPDA method to set the point of execution within a stack frame, this project uses bytecode modification to insert a *'goto'* instruction at the opcode level, skipping previously executed instructions. Further details of the bytecode modification can be found in section 4.3.1.

As shown in figure 6, restoration of the execution environment begins with the External Monitor setting a break point in the Internal Monitor, causing its execution to be suspended after having deserialized the Mobile Object. From the information contained within the Stack List, the next method to

```
01  public class MyMobileObject implements Serializable {
02     MyObject myObj = null;
03     public void doMethod() {
04        int i = 0;
05        int j = 0;
06        myObj = new MyObject();
07        InternalMonitor.finishedVariables();
08        move(); move();
09        for (i = 0; i < 100; i++) {
10           j = j * i;
11           System.out.println("This is j = "+j);
12           move(); move();
13        }
14     }
15  ...
```

Figure 5: Sample Mobile Object Method

be called is determined and modified to contain the goto command, pointing to the correct location of execution. The method's class is redefined, loading the amended code into the JVM. A break point is added for the same location to which the goto refers, suspending execution once previously executed code has been skipped. The Mobile Object is resumed, allowing the method to be invoked and declare all its variables. Once suspended at the break point, all of the method's local variables are set to the values they were before migration. Again the next method to be invoked is determined and modified. This process continues iterating until the method call stack is completely rebuilt.

Upon completion of the method call stack restoration, all loaded classes are once more redefined to their original form. This removes all goto statements from the method definitions, allowing subsequent invocations to operate as normal.

It is important to note the role of the Internal Monitor during the state restoration process. The Internal Monitor is responsible for deserializing the Mobile Object into the Monitored Thread. It is not possible for the External Monitor to accomplish this, as debugging applications are unable to obtain direct references to objects, as detailed in section 2.4. Furthermore, the Internal Monitor provides a means to delay the commencement of the Mobile Object after deserialization, until its definition is revised by the External Monitor. It is imperative that execution be suspended before the method is called, for modifications to the bytecode are not adopted once a frame is placed upon the stack (see section 2.4 for details on class redefinition). In comparison to figure 4, figure 6 shows the activities required between loading the Mobile Object and invoking *objectStart()* during state restoration.

4.3.1 Bytecode Modification

For each frame in the method call stack at the time of migration, an entry exists in the Stack List detailing the code index of the next instruction to execute. To skip previously completed commands, a *goto* pointing to the code index is inserted into each method. As Java does not support the use of *goto* statements at the source code level, this project modifies classes by inserting them at the bytecode

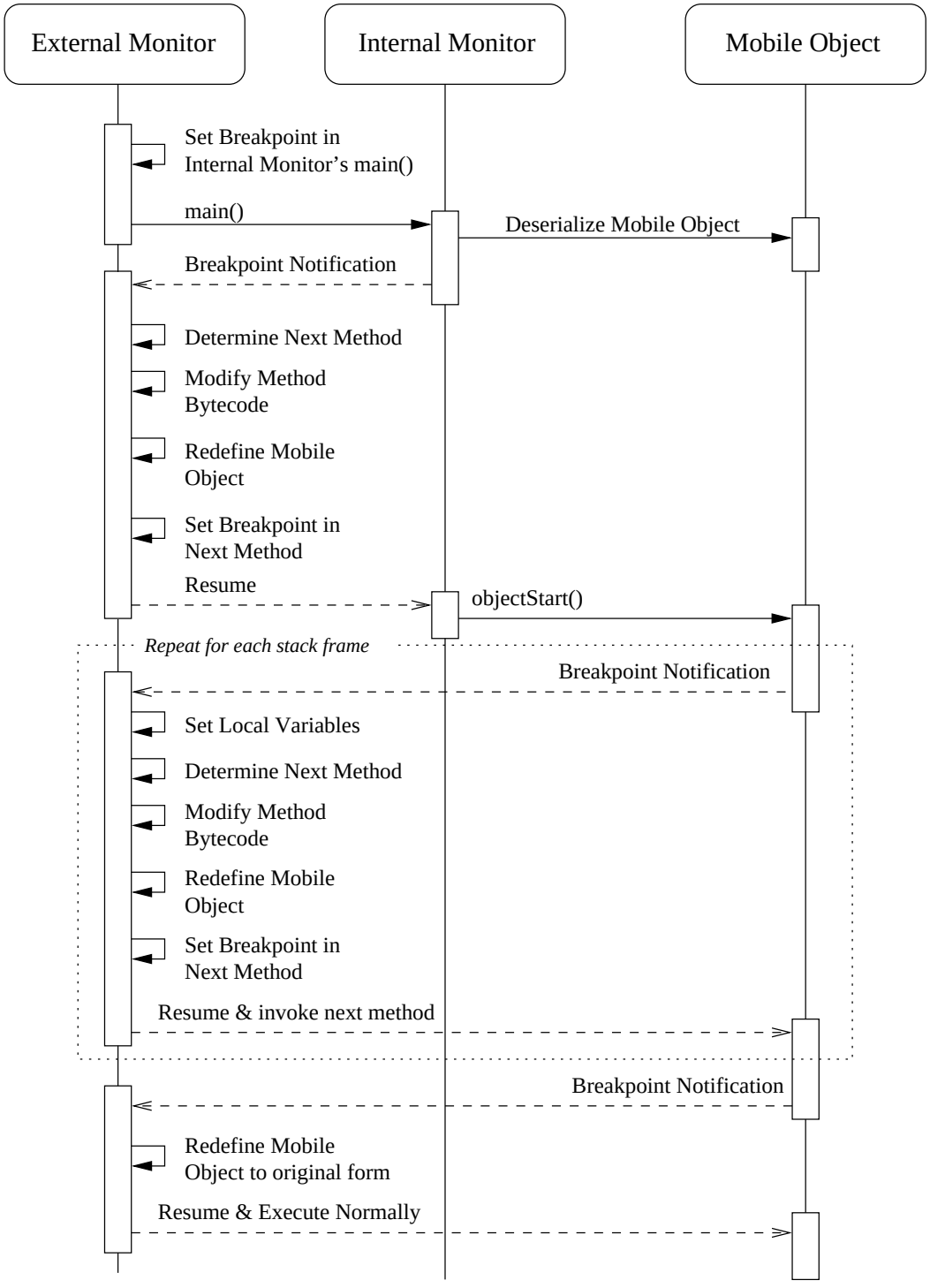

Figure 6: Execution State Restoration Sequence

level. *goto*s contain a reference to the next instruction to execute known as a branch index. Branch indices are position independent byte locations, counted from the beginning of the code segment.

In order to achieve the code insertion, this project utilises the Byte Code Engineering Library (BCEL) [Dah99] [Dah02]. This library provides convenient methods for the reading, writing and alteration of classes. Additionally, after the insertion or deletion of code, the BCEL automatically updates branch indices of instructions. While it is possible to achieve more specialised and efficient results via a custom implementation, this project has capitalised on the available API due to time constraints.

Locating the correct position to insert additional bytecode is imperative. If inserted too early, vital variable declarations will be skipped when the method executes. At run time, this would result in the JVM trying to assign values to variables which haven't been declared. As part of Java's security framework, classes are parsed through the *Bytecode Verifier* at load time to ensure they are well formed [Sch02][Mic99]. Classes with invalid bytecode, such as assigning values to undeclared variables, fail verification and are not loaded, preventing unsafe situations. Incorrect insertion will result in the Mobile Object not being loaded, thus terminating its execution.

To assist locating the appropriate insertion point, a call to the Internal Monitor's static method *finishedVariables()* is made at the end of a method's variable declarations, as shown on line seven of figure 5. *finishedVariables()* contains no instructions of its own, rather acting as a reference point within a method's bytecode. At this point the *goto* command may be inserted without fear of adverse effects on the validity and stability of the class.

Figure 7 details a simple example of the code insertion. In this scenario, execution has been suspended at the end of byte eleven. During reconstruction, a *goto* pointing to the next instruction at byte twelve is inserted after byte five (byte five is the invocation of *InternalMonitor.finishedVariables()*). Due to the *goto* being an instruction itself, the code indices have been incremented, and subsequently the *goto* references byte fifteen.

Further implementation details of the bytecode modification may be found in Appendix A - Bytecode Modification on page 29.

5 System Evaluation

In order to evaluate the developed Execution Context Migration Framework, attention has been paid to aspects directly influencing system performance. Section 5.1 details the temporal testing performed, in particular the overhead placed upon executing objects (section 5.1.1) and the time required for state capture and restoration (section 5.1.2). Section 5.2 explores the growth of object code in the system, and section 5.3 details the limitations of the current implementation.

All tests on this framework have been performed using a single CPU Intel Pentium 4 1.7 GHz machine, with 512 Mb of RAM, running Windows 2000 Professional and Sun's JVM version 1.4.1_02.

Original	Modified
0 iconst_5	0 iconst_5
1 istore_1	1 istore_1
2 bipush 9	2 bipush 9
4 istore_2	4 istore_2
5 invokestatic #2	5 invokestatic #2
8 iload_2	8 **goto 15**
9 iload_1	11 iload_2
10 iadd	12 iload_1
11 istore_2	13 iadd
12 iload_2	14 istore_2
13 iload_1	15 iload_2
14 imul	16 iload_1
15 istore_1	17 imul
16 iinc 2 by 1	18 istore_1
19 iinc 1 by 255	19 iinc 2 by 1
22 return	22 iinc 1 by 255
	25 return

Figure 7: Bytecode Modification Example

5.1 Temporal Evaluation

Temporal testing of the Execution Context Migration Framework utilises two algorithms, the *Quicksort* of one million integers and the non-optimised[4] *Fibonnaci* sequence of 45. The recursive nature of these algorithms makes them ideal for creating method call stacks of various sizes, and keeping the size of method data constant for each invocation. This provides greater control and reliability of performance measurements during the capture and restoration of execution state. Additionally, such tests have been carried out by other Mobile Object systems [SMY99][SSY00], allowing for closer comparison.

5.1.1 Execution Overhead

Strong Mobility frameworks developed in Java have traditionally imposed a performance overhead on running applications. This has primarily been due to additional instructions required for the check pointing and restoration of the execution state. To evaluate this system's overhead, addressing the second project aim of section 3, both the Fibonnaci and Quicksort algorithms were run in the standard JVM, as well as the implemented framework. During execution, no state capture or rebuilding were carried out, allowing programs to continue unhindered. The results of this experiment are shown in figure 8.

As illustrated in figure 8, the overhead experienced by the Quicksort routine was negligible, with

[4]Some implementation of the Fibonnaci sequence store results of previous values for use in subsequent calculations. This *non-optimised* implementation recomputes all values for each number in the sequence and does not store progressive results.

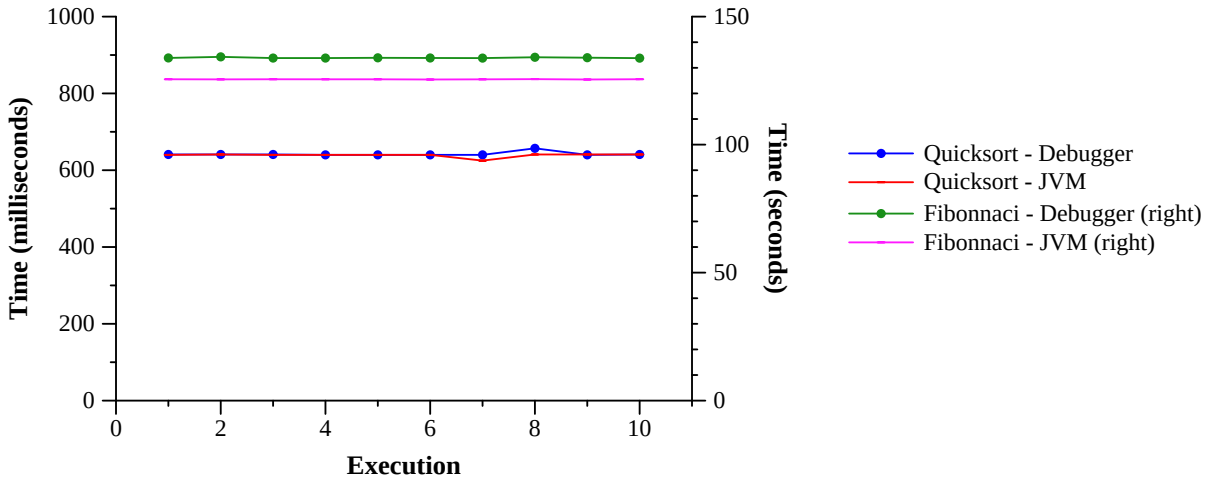

Figure 8: Performance overhead of applications in the Debugger versus the JVM

running times matching closely. The Fibonnaci algorithm however, required 7% more time to execute in the developed framework than the standard JVM. The increased burden upon the application is a result of the External Monitor checking its event queue for notification of fired events. It is not clear from these tests why the increase did not manifest in the Quicksort, although not crucial to the system, it remains an area that requires further study.

While ideally the Mobile Object framework should not introduce operational overhead to applications, as highlighted in figure 9, 7% is minimal in comparison to Sekiguchi's JavaGo [SMY99][SSY00] which experiences additional computation cost of between 16% and 183%. Additionally, Inria's modified JVM [BH99] results in applications running ten times slower than with a standard JVM.

This assessment of the developed framework allows the second project aim of section 3 to be answered, as this system only imposes a 7% overhead on running applications.

Program	JavaGo	JavaGo Extension	JPDA Debugger
Fibonnaci	183%	56%	7%
Quicksort	27%	16%	0%

Figure 9: Performance Overhead Comparison

5.1.2 State Capture and Restoration

In addition to the overhead placed on running applications, the time required to capture and restore the execution environment is another important aspect of a Mobile Object System. This section investigates different execution conditions, and their influence upon the framework.

As detailed in section 4.2, capturing the execution state is broken into two main steps, traversal of the method call stack and the serialization of Mobile Objects. To evaluate the dependency between stack depth and state preservation time, both the Quicksort and Fibinnaci algorithms were run to varying levels of method calls before migration was invoked. This allowed for the performance of

each component of the framework to be measured and assessed.

No. Frames	Fibonnaci	Quicksort
10	255	280
15	278	356
20	302	405
25	336	483
30	409	545
35	413	627

Figure 10: State Capture Times (ms)

The results for the state preservation of this experiment, as listed in figure 10, highlight the dependency of time upon the number of stack frames. The Fibonnaci algorithm required 413ms to save 35 frames, and only 255ms to store 10. The relationship between time and stack size is not shared with Inria's system based upon a modified JVM, which can perform state capture for 5 frames in 8ms, and requires only 9ms for 50 frames. This is primarily due to the mechanisms in which the state information is obtained. Inria, having direct access to the internals of the JVM, is able to take a snapshot of an application without iterating through the stack. In contrast, the framework described in this thesis must invoke methods upon the JVM to obtain every stack frame and variable, placing an additional burden upon the process. It must be noted however, that while Inria's system may capture and restore the execution environment in less time, it also places a far greater overhead upon its applications, up to 1000% [BH99]. Furthermore, the stack sizes used in this experiment are not typical of common business applications, and are used solely as a performance metric.

The investigation of stack dependency was also carried out for the state restoration process, as illustrated in figures 11 and 12. Again there appears a proportional relationship between the number of frames and required time. More importantly however, is the magnitude of time taken to rebuild the execution environment, between 9.4 and 27.9 seconds in comparison to 255 to 413 milliseconds of state capture for the Fibonnaci algorithm. These results answer the third project aim of section 3, indicating that while state capture falls within acceptable limits, state restoration imposes a substantial performance penalty.

No. Frames	Fibonnaci	Quicksort
10	9.4	8.5
15	12.9	12.2
20	17.3	15.6
25	21.3	19.3
30	24.4	22.7
35	27.9	26.5

Figure 11: State Restoration Times (seconds)

Rebuilding the execution environment, as detailed in section 4.3, consists of four main tasks: resetting local variable values, determining the next method to be invoked, bytecode modification and class redefinition. Further examination of this sequence of events reveals the bytecode modification to be responsible for the experienced delays, with the execution times for the remaining three steps being

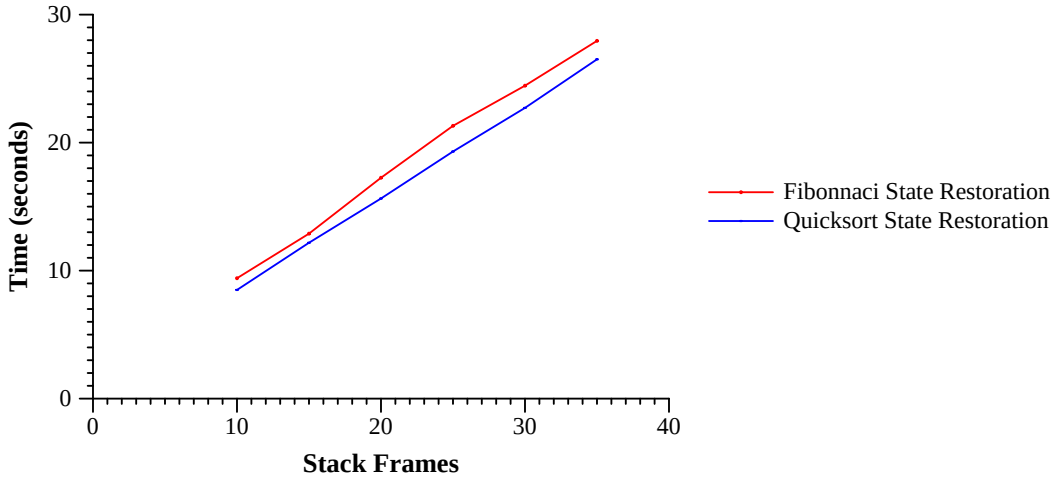

Figure 12: State Restoration for Varying Stack Sizes

comparably negligible. For each method on the call stack, the BCEL (section 4.3.1) reads in the appropriate class, parsing its various components to create a detailed object structure ready for developer manipulation. The repeated loading and parsing of classes severely impedes the performance of this system, and constitutes up to 98% of the frame restoration time as shown in figure 13. As described in section 6, further work may be carried out to optimise this aspect of the system, incorporating such measures as a custom implementation and or class caching.

Total Time	Bytecode Modification	Other Processing	Modification Percentage
719	688	31	96%
609	578	31	95%
609	594	15	98%
625	609	16	97%
672	656	16	98%
688	656	32	95%
656	641	15	98%

Figure 13: State Restoration Time Breakdown Per Stack Frame (ms)

Further analysis has been conducted to evaluate the effect of class size on the duration of bytecode modification. Figure 14 confirms the assumption that modification time is proportional to the size of the class being operated upon. Additionally it confirms the temporal expense of the process, further emphasising the need for an optimsed solution to the code modification.

5.2 Code Growth

Previous sections of this paper have investigated the temporal performance of the developed Mobile Object framework. This section concentrates upon the spatial aspects, in particular object growth, addressing the fourth project aim of section 3.

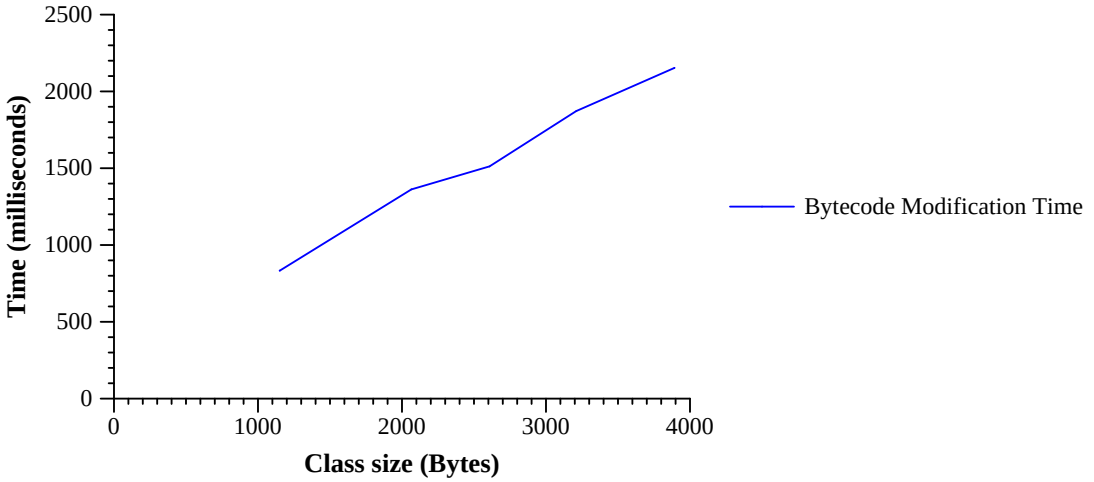

Figure 14: Class Code Size and Bytecode Modification Time

Standard compilation of a Java class does not contain the required information to perform all debugging functionality. As outlined in section 2.4, developers must explicitly request its addition from the compiler. Issuing the *-g* switch will result in the inclusion of all debugging information, namely the source file name, line numbers and variable names. The External Monitor (utilising debugging methods) requires the names of variables during the state capture and reconstruction. To investigate the growth of object code, a selection of compilation options have been applied to different sized classes. Figure 15 illustrates that increases between 24% and 28% are not uncommon with the inclusion of all debugging data.

Class	Standard	Variables	All Data	Variables Increase	All Data Increase
Quicksort	1 115	1 164	1 387	4%	24%
Host Thread	6 019	7 059	7 708	17%	28%
Execution State	2 351	2 558	2 919	9%	24%
Packed Variable	601	625	7 50	4%	25%
Client Thread	7 022	8 088	8 715	15%	24%
Transport Manager	20 485	22 640	25 511	11%	25%

Figure 15: Class Compilation Size Comparison

The External Monitor, however, does not require all of the debugging information, only the names of the local variables. These may be solely included using the *-g:var* switch. This selectivity reduces the code increase to between 4% and 17%. These results address the fourth project aim of section 3, indicating the low rate of object code growth.

In comparison to other strong mobility systems in Java, a 17% expansion in code is of little impact. AdJava [FO01] suffers from increases of up to 300%, while Sekiguchi's JavaGo [SMY99][SSY00] exhibits growths of 130% to 330%. These increases can result in longer object transfer times, and higher memory consumption. It should be noted, that unlike other systems, the addition data causing the growth is not executable code, rather only support information for the debugger.

During the bytecode modification, a *goto* command is inserted into each method on the call stack.

Although this command is executed only once, and is present only during the state restoration process, it does temporarily raise the size of the method. This increase is guaranteed not to exceed five bytes per method, as this is the largest *goto* currently supported within the JVM [Mic99].

Mobile Objects running in this framework also experience a more subtle form of code growth. The inclusion of *move()* invocations throughout the application inevitably increase the size of the class file. The extent to which this occurs is largely in the hands of the developer. If inserted after every statement, considerable increases may be witnessed. Selected and strategic placement will reduce the code expansion, and may improve system performance. It is left to the developer to determine the trade off between object size and migratory locations. Alternatively, a pre-processor may be developed for the insertion of *move()* statements within the source code (as discussed in section 6). It could possess various developer settings for the optimal insertion locations based on execution time or class size requirements.

5.3 Limitations

As outlined in section 2.2, object serialization does not capture local method variables or objects with method scope. During the execution of the Mobile Object, it is highly likely that migration will occur part way through a method. For this reason, all objects that are used within the Mobile Object (and subsequent objects) must have all object references declared with class scope. This enables the serialization to capture the state of all objects currently in use, ready for reconstruction at the destination host. Lines two and six of figure 5 illustrate how the object reference is declared with class scope, and instantiated within the required method. Primitive method variables are captured during stack processing as detailed in section 4.2.

An additional limitation applies to the declaration and initialisation of method variables. As outlined in section 4.3, should a *goto* instruction be inserted before a variable declaration, the class may fail bytecode verification. All variable declarations must be moved to the head of a method, followed by a call to *InternalMonitor.finishedVariables()*. Furthermore, all variables must be initialised before invoking *finishedVaraibles()*. During compilation, the maximum number of local variables for each method is calculated and stored in the resulting class file. When a method is called at run time, an array is created in the JVM of length equal to the maximum number of variables. Each variable resides in an array element, to which type information is not known until a variable has been assigned. Should it be possible to jump to a location and assign a value to a declared but uninitialised variable, the bytecode verifier will fail the class. For this reason, proper initialisation of variables must also be moved to the head of a method. Figure 16 illustrates the valid initialisation of variables.

Invalid	Valid
int j;	int j = 0;
int k;	int k = 5;

Figure 16: Variable Initialisation

There exist cases whereby a mobility system may wish to relocate only one or two operating objects, leaving the rest unaltered (eg. load balancing). The Execution Context Migration Framework currently provides for the capture and migration of *all* objects within the object graph of a particular Mobile Object. This limitation results from not handling the adaptation of references and resources

contained within objects, and reliance on the standard Java serialization. Additionally, the current implementation of the framework does not cater for multi-threaded applications. This feature is to be developed as part of the future work for this project (see section 6).

6 Future Work

Although this thesis has demonstrated the potential effectiveness of the Execution Context Migration Framework, the following have been identified as areas for future research:

Bytecode Modification Optimisation In the current implementation, the BCEL receives the class name for each method, retrieves the file and parses it. As the BCEL has been designed to provide generic bytecode services to developers in an Object Oriented manner, the overhead involved in loading a class is considerably high. The process of bytecode modification must be optimised in order to reduce the cost of state restoration. By developing a custom implementation, overhead of loading and modifying classes may be reduced. Class caching could also be implemented to further refine execution times. By storing previously loaded classes, classes with multiple methods on the call stack would not need to be read in more than once. This could cut the amount of processing required for parsing and file I/O.

Pre-Processing As discussed in section 4.1, the insertion of calls to *move()* are currently performed by the programmer. To create a more transparent and seamless approach, a pre-processor may be developed to carry out the invocation insertions. Not only does this free the developer to concentrate on the specifics of the application, but will also allow for heuristic placement optimisation to take place. Heuristic rules for different configurations could allow developers to specify an emphasis on file size, execution time, or even particular methods optimisation. Such a process could greatly improve system performance over random programmer placements.

Multi-Threading The current implementation only handles single threaded Mobile Objects. This system is extensible to cater for multi-thread Mobile Objects.

Resource Adaptation While Mobile Objects themselves may be transported between hosts, this is not always true of the reference they hold. Object references such as streams, printers and input devices require a more advanced solution, as they may not be sent along the wire [Per03]. One possible approach proposes the wrapping of resources via proxy mechanisms [YMH+00].

7 Summary and Conclusion

In response to the weaknesses of existing strong migration systems, and in an effort to address the four project aims of section 3, this project implemented the Execution Context Migration Framework. The framework supports strong migration by utilising the JPDA, obtaining the exact location of execution at the time of suspension. Through the insertion of *goto* statements via bytecode modification, execution is resumed at the same point it was suspended. Furthermore, Mobile Objects running in the system are not required to migrate at each invocation of *move()*. The External Monitor determines the appropriate time to instigate migration, providing an adaptive approach to object mobility.

The spatial evaluation of the Execution Context Migration Framework illustrates its ability to keep code sizes low in comparison to other systems, between 4% and 17%, whilst still allowing full state reconstruction. A further strength of the framework is the minimal overhead of only 7% placed upon running applications.

While the running time for preserving the execution state falls within acceptable limits, state reconstruction imposes a sizeable performance penalty, due to the bytecode modification process. It is believed that further research may optimise this routine by creating a custom implementation, delivering a more acceptable performance. However, as the framework stands, it is envisaged that it will be of benefit to applications which are required to operate with little to no overhead, and infrequent migration. In such a scenario, the comparably slow transportation of the application will be outweighed by its ability to deliver faster execution to application users.

As the JPDA is part of the standard JVM, the developed framework is portable across multiple platforms, allowing it to operate in a heterogeneous environment. While the state capture and restoration is performed faster in some modified JVMs, the sacrifice of portability is too great in today's computing society. The nature of the Next Generation Internet, with computing occuring on varying types of devices, requires that distributed applications be adaptable to a range of host configurations.

It is hoped that future versions of Java provide stronger support for the capture and restoration of execution state of running applications. Alternate solutions such as the Execution Context Migration Framework are an interim measure, designed to allow strong mobility in a Java environment until such time as inherent support is provided within the Java framework.

8 Acknowledgements

I would like to thank my supervisor, Dr Caspar Ryan, for his advice and guidance throughout the year. Also my family, for their endless support, as well as Steve Perry, Rebecca Mifsud and Simon Duff for many valuable discussions.

I would like to thank the Inria research team, the Byte Code Engineering Library and the Sun Development team for the use of their systems and material.

References

[ARS97] A. Acharya, M Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130. Springer-Verlag, Heidelberg, Germany, 1997.

[BC95] K. A. Bharat and L. Cardelli. Migratory Applications. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag: Heidelberg, Germany, 1995.

[BCM03] P. Bellavista, A. Corradi, and R. Montanari. Dynamic Binding in Mobile Applications : A Middleware Approach. *IEEE Internet Computing*, 7(2):34 – 42, 2003.

[BH99] S. Bouchenak and D. Hagimont. Pickling threads state in the Java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99),*, Madeira Island, Portugal, 1999.

[Dah99] M. Dahm. Byte Code Engineering. In Clemens Cap, editor, *Java-Informations-Tage (JIT'99)*, 1999.

[Dah02] M. Dahm. Byte Code Engineering Library, 04/12/2002 2002. `http://bcel.sourceforge.net/`.

[DR98] B. Dimitrov and V. Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–482, 1998.

[FO01] M. Fuad and M. Oudshoorn. AdJava - Automatic Distribution of Java Applications. In Michael Oudshoorn, editor, *Twenty-Fifth Australian Computer Science Conference*, volume 4, pages 65 – 75, Melbourne, Australia, 2001. Australian Computer Society.

[FPV98] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342 – 361, 1998.

[Fun98] S. Funfrocken. Transparent Migration of Java-based Mobile Agents. In F. Hohl K. Rothermel, editor, *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, pages 26–37, Stuttgart, Germany, 1998. Springer-Verlag.

[Har99] E. R. Harold. *Java I/O*. O'Reilly, 1999.

[IBM03] IBM. Next Generation Internet, 2/10/2003 2003. `http://www.ngi.ibm.com`.

[LG03] LG. LG Internet Family, 2/10/2003 2003. `http://www.lginternetfamily.co.uk/fridge.asp`.

[Mic99] Sun Microsystems. The JavaTM Virtual Machine Specification - Second Edition, 1999. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.ht%ml`.

[Mic00] Sun Microsystems. Java Platform Debugger Architecture Overview, 2000. `http://java.sun.com/j2se/1.3/docs/guide/jpda/jpda.html`.

[Mic02] Sun Microsystems. The Java HotSpot Virtual Machine v1.4.1 White Paper, 2002. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.%1/Java_HSpot_WP_v1.4.1_1002_3.html#full`.

[Mic03a] Sun Microsystems. Handling Errors with Exceptions, 2003. `http://java.sun.com/docs/books/tutorial/essential/exceptions/`.

[Mic03b] Sun Microsystems. Object Serialization: Frequently Asked Questions, 2003. `http://java.sun.com/products/jdk/serialization/faq`.

[MU01] S. Moyer and A. Umar. The Impact of Network Convergence on Telecommunications Software. *IEEE Comunications*, pages 78–84, 2001.

[Per03] S. Perry. Yet to be published: Object Transport and Reference Management For an Adaptive Next Generation Internet Framework, 2003.

[RH97] M. Richmond and M. Hitchens. A New Process Migration Algorithm. *ACM SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[Sch02] M. Schnefeld. Security Aspects in Java Bytecode Engineering, August 1st - 2nd 2002. `http://www.illegalaccess.org/blackhat/blackhat.pdf`.

[SM99] K. Shudo and Y. Muraoka. Noncooperative Migration of Execution Context in Java Virtual Machines. In *Proc. of the First Annual Workshop on Java for High-Performance Computing (in conjunction with ACM ICS99)*, Rhodes, Greece, 1999.

[SMY99] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and Its Portable Implementation. In *Coordination Models and Languages*, pages 211–226. 1999.

[SSY00] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Joint Sym- posium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 16–28, 2000.

[WJ95] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115 – 152, 1995.

[WTV02] D. Weyns, E. Truyen, and P. Verbaeten. Distributed Threads in Java. In *International Symposium on Distributed and Parallel Computing (ISDPC)*, 2002.

[YMH+00] H. Yamauchi, H. Masuhara, D. Hoshina, T. Sekiguchi, and A. Yonezawa. Wrapping Class Libraries for Migration-Transparent Resource Access by Using Compile-Time Reflection. In *Workshop on Reflective Middleware*, 2000.

A Bytecode Modification

Locating the correct positing to resume execution requires careful analysis of the bytecode surrounding the code index returned by the JPDA. This section details the particularities of ensuing the correct path of execution is followed.

Java's bytecode consists of strongly typed instructions to operate the JVM. For example, to load an integer onto the operand stack[5] the *iload* command is used. Similarly, loading floats, longs and object references require the use of *fload, lload* and *aload* respectively.

The strong typing of instructions is not restricted to the loading of variables, as method invocations also require different commands depending upon their type. Static methods are invoked with *invokestatic*, object constructors with *invokespecial* and standard object methods with *invokevirtual*. In addition to enforcing security constraints at the bytecode level, these instructions provide for the different semantics required for the operations.

It is possible to maintain many references to objects of the same type within the same scope. Therefore, to invoke a method of an object, the JVM must be informed of which instance to use. For this reason, *invokevirtual* is preceded by loading an object reference onto the operand stack. This is shown in figure 17, as the object reference is loaded at byte thirteen, the method parameter is loaded at byte fourteen, and the method invoked at byte fifteen. A static method however, does not require an object to be instantiated before it is invoked, consequently the instruction may be issued without loading an object reference. Note that in figure 17 byte ten invokes the static method without loading an object reference.

Source Code		Bytecode	
	`public void myMethod()`	1	`0 iconst_0`
	`{`		`1 istore_1`
1	`int i = 0;`	2	`2 new #2`
2	`MyObject obj = new MyObject();`		`5 dup`
3	`InternalMonitor.finishedVariables();`		`6 invokespecial #3`
4	`obj.methodA(i);`		`9 astore_2`
	`}`	3	`10 invokestatic #4`
		4	`13 aload_2`
			`14 iload_1`
			`15 invokevirtual #5`
			`18 return`

Figure 17: Java Bytecode Example

The method call stack is rebuilt during state restoration by invoking methods in the same order they were before suspension. The execution point of each method is stored in the Stack List as a code index, which details the *next* instruction that *was* to be processed. However, to invoke methods in the same sequence the *previous* instruction to the code index (ie. the method call) must be issued. This is not a simple matter of reducing the code index by one command, as different types of method

[5]The operand stack provides a location for storing and manipulating values and references during calculations and operations. For more information see [Mic99].

calls possess varying semantics. For example, in figure 17 had *methodA* of *obj* been called as part of the call stack before migration, the code index would be pointing to the return statement at byte 18. Decrementing this by one instruction to byte 15 will reference *invokevirtual*, but will fail to load the required object reference at byte 13, and the method parameter at byte 14.

To handle the varying types of method invocations, the code index is initially reduced by one instruction. At this point the call type is evaluated along with the method signature to determine the number of parameters. If the instruction is *'invokestatic'*, the code index is left unaltered. If the instruction is *'invokevirtual'*, the next instance of *aload* (loading an object reference) is found by searching backwards through the method. Should there be method parameters in either case, the code index is decremented further in order to allow them to be loaded onto the operand stack. This is achieved with the assistance of the number and type of parameters extracted from the method signature.