Phantom: Heterogeneous Process Migration in Java

Harit Modi

Christopher S. Serra

{*harit,csserra*}*@cs.wisc.edu*

Computer Sciences Department

University of Wisconsin-Madison

Madison, WI 53706

Abstract

The problem of heterogeneous process migration, moving running processes between hosts of different platforms, can be simplified by focusing on interpreted systems. In such systems, the interpreter can serve as a platform-independent abstraction above the level of native architecture. The Phantom system implements a heterogenerous migration system based on the portable infrastructure of the Java programming language and virtual machine. Phantom's migration mechanism is both transparent and preemptable. Its implementation, based on Sun's Java Development Kit (JDK), required minimal internal modifications to the JDK source code. Design issues and implementation details of Phantom are described in this paper.

1 Introduction

Process migration is the act of suspending a running process and relocating its code and runtime state from one *host*, or machine, to another. A host can be categorized by its *platform*, whose two components are the

operating system and underlying machine architecture. It is assumed that migration occurs in the context of loosely-coupled hosts, such that it cannot be trivially implemented using shared memory. In many cases, migration mechanisms operate through the creation of *checkpoints*, an encoding of a process' runtime state at a particular point in time. In the context of migration, a process' checkpoint can be sent to the migration *target*, or destination machine, where it is used to *restore* the process and resume its execution. Migration and checkpointing can be used to implement dynamic load balancing, resilient (fault tolerant) processes, or disconnected computing.

While the area of *homogeneous* process migration, moving processes between hosts of the same platform, has been well-researched [7] [9] [13] [14] and presents straightforward problems and solutions, mechanisms for *heterogeneous* process migration [10] [12] (between hosts of different platforms) are difficult to implement. This is mostly due to the inherent differences in native architectures, such as instruction sets, stack conventions, and register layouts. This problem generally demands a machine-independent representation for a process' code and state. Mechanisms for encoding and decoding the intermediate form for various architectures have been studied both at the level of source language ("high-level") [10] and native code ("low-level") [12]. In both cases, the implementation and runtime overheads of such mechanisms are costly.

This project was motivated by the observation that the platform-independent abstraction of the Java interpreter [6] provides an excellent infrastructure for implementing heterogeneous migration of Java processes. The Java Virtual Machine (JVM) provides an abstract architecture above the level of machine and operating system. Accordingly, the JVM's platform abstraction can be used to create a process migration mechanism that targets heterogeneous systems "for free". We have taken advantage of this fact to implement a Java-based heterogeneous migration facility named Phantom (**P**reemptable, **h**eterogeneous **an**d **t**ransparent **o**bject **m**igration).

Besides the benefits of inherent heterogeneity, we chose the JVM as a framework for migration because Java is a widely used language, and therefore a migration mechanism based on Java could potentially see real use outside of domain-specific research groups. To further this goal, we have intentionally designed a mechanism that is *transparent* and *preemptable* [13]. Transparency has the two related meanings that

processes do not know they are being migrated and that Java programmers do not need to modify their code to facilitate migration. Preemptability denotes that a process can be arbitrarily migrated at any point in its execution rather than only at explicitly specified points. This distinguishes our system from *lightweight* migration mechanisms [5] [8] that allow migration only under specific runtime conditions or require an explicit source level API.

We have implemented a migration mechanism that successfully relocates Java processes between JVMs on different native platforms. Our implementation is based on version 1.1.5 of Sun Microsystem's Java Developer Kit (JDK), and includes extensions and (minimal) modifications of the JVM source code.

Due to the time constraints of our project, we were able to implement only a very basic migration infrastructure. Most forms of *external references* are not accounted for in our mechanism, but are listed as possible areas of future work. These include file I/O, network I/O, and object synchronization, to name a few.

Section 2 of this paper presents related work in various types of process migration. Section 3 summarizes relevant details of the Java language and virtual machine. Section 4 briefly describes the major components of our migration system. Section 5 contains more rigorous discussions of the system, including design rationales and implementation specifics. Section 6 presents some performance statistics for our migration mechanism. Finally, in Sections 7 and 8 we discuss the large body of potential future work and present our conclusions.

2 Related Work

Early research in process migration focused on homogeneous systems. DEMOS/MP [9] was one of the first systems to successfully implement a process migration mechanism. The V-System [13] introduced the mechanism of "pre-copy" for minimizing migration freeze time. This issue was also addressed by the virtual memory-based mechanism of "copy on reference" used in Accent [14] . Condor [7] is an example of a load balancing system based on homogeneous process migration that is still in use today.

Heterogeneous migration systems have been categorized into four areas [12]. In *passive object* systems,

migrated objects have data but no code. These frameworks concern themselves solely with issues of data representation, and generally appear in process migration systems under the guise of heterogeneous data transfer mechanisms. *Active object, migrate when inactive* systems transfer code as well as data, but avoid the issue of execution state. One such system is ObjectSpace Voyager [8], which provides Java-based object mobility, but only when the object has no active threads.

Active object, interpreted code heterogeneous systems allow the migration of running interpreted processes. These systems, of which Phantom is one, benefit from the fact that the interpreter introduces a level of abstraction above the native architecture layer, which simplifies processes' runtime state. A similar project at the University of Tennessee - Knoxville, titled "Architecture Independent Checkpointing", is implementing Java-based migration by modifying the Kaffe virtual machine. IBM's Aglets Workbench [5] also provides Java object mobility, but at the cost of requiring explicit language level primitives; aglets must initiate and be aware of their own movement.

The last and most in-depth migration systems are *active object, native code* frameworks. These systems deal with the full complexity of translating a process' active state from one native architecture to another. They must deal with differences in architectural components such as instruction sets, frame conventions, and register files. The Tui system [12] sends intermediate representations of low-level state between multiple pre-compiled binaries of a program. By comparison, the HiCaM [10] system captures runtime state at the language-level using source code analysis.

3 The Java Environment

As was mentioned above, our system's implementation is based on the Java language and virtual machine. Accordingly, some relevant details of the Java framework must first be explained to clarify the following discussions of design and implementation. The JVM interpreter is truly a virtual machine, complete with its own (stack-based) instruction set architecture. It also specifies a uniform data representation format, effectively defining away the issue of platform endianness.

Java is a statically-typed, interpreted, object-oriented language. Every Java class automatically inherits from the top-level class `Object`, which means that any instance of any class is also an instance of `Object`. Processes in Java are encapsulated in `Thread` objects. Consequently, the problem of process migration boils down to the problem of relocating a `Thread` object from one JVM to another.

4 Design

In this section, we discuss the design goals that guided the implementation of Phantom. Creating a migration mechanism that exhibits heterogeneity was the original motivation for the entire project. However, we also wanted our migration system to be flexible and easy to use. Consequently, transparency and preemptability are provided to minimize the restrictions and programming overhead of our migration mechanism.

4.1 Heterogeneity

The primary reason for using a Java-based mechanism is that we get the "heterogeneous" part of heterogeneous process migration for free. Java was intentionally designed to provide a machine-independent programming abstraction, to promote the idea of "write once, execute anywhere" programs. At both the Java and native level, the JVM provides a wonderful infrastructure for heterogeneous systems. For instance, the Java API already provides a mechanism for dealing with the entire issue of machine-independent representations of data. Even within the native implementation of the JDK, care has been taken to encapsulate platform-specific functionality inside abstract interfaces. While the implementation of Phantom required tackling several difficult problems, heterogeneity was never an issue.

4.2 Transparency

Java provides a facility called *object serialization* [1], which consists of mechanisms for saving and restoring the state of any object in a machine-independent manner. This facility has been extended to correctly serialize the execution state of Java threads, such that a checkpoint can be created by the simple act of

serializing a `Thread` object.

Because our design has not changed the model for programming threads, and specifically does *not* require the use of new interfaces, we say that our mechanism is *code-transparent*, meaning that existing programs can be migrated without being modified or even recompiled. This distinguishes our system from one such as Aglets [5], which requires that programs be modified to use a product-specific API. Likewise, we say that Phantom migration is *process-transparent* in that the threads are unaware that they are being migrated. From the thread's perspective, migration is semantically equivalent to being suspended and then resumed.

4.3 Preemption

The term "preemption points" [12], meaning places in a process' execution at which migration is possible, has paved the way for the concept of *preemption granularity* in migration systems. A *fully* preemptable migration system is one in which a process can preempted at *any* time. While full preemption was our goal, our system realizes preemption at a slightly coarser granularity.

The JVM provides functionality for *native* method invocation, which introduces all of the complications associated with native heterogeneous migration systems [12]. In order to avoid these issues, Phantom's preemption points are actually defined to be at the granularity of Java instructions[1]. Therefore, if migration is attempted on a thread while it is executing a native method, the act of migration is suspended until that native method returns.

While it is not fully preemptable, Phantom still provides a high degree of preemptability. A thread can be migrated at any point in its *interpreted* execution. This functionality does come at a cost - it makes the problem of migration more difficult. This is because the state of a running thread is complicated by structures such as the JVM's execution and operand stacks. This is discussed in greater depth in Section 5.3.

[1] By comparison, the Voyager mobile object system [8] requires that all of an object's executing threads exit before the object can be moved.

5 Implementation

This section highlights some of the major points of Phantom's implementation. It starts with a walkthrough of the steps that make up a Phantom migration. This is followed by a discussion of Java object serialization, which was a key leverage point for Phantom's checkpoint facility. Next, we enumerate the elements of runtime state that are necessary to migrate a Java thread. We describe the three biggest problems encountered in the course of the project (bytecode access, operand types, and native methods), as well as our implementation solutions. Finally, we describe how our solutions fits with the next release of the JDK.

5.1 Migration Overview

This section presents a high-level overview of how migration proceeds in Phantom. The migration for a given thread proceeds as follows: (i) the thread is suspended; (ii) the thread is serialized to form a checkpoint, (iii) the serialized thread, encoded as a byte stream, is transferred from the source JVM to the target JVM; (iv) the thread is recreated (in its suspended state) on the target via deserialization; (v) finally the thread is resumed on the target and execution continues.

One important aside is that Phantom does not try to address the issue of migration *policy*. Phantom is simply a mechanism that provides the functionality to migrate a thread. The question of who migrates threads and when they are migrated is up to the designer of the higher-level migration *manager*. We envision that these decisions would be encoded in migration daemons, which could be implemented either at the user-level or within the JVM itself. Also note that most migration *pools* consist of several migration managers acting in concert.

5.2 Object Serialization

Java's object serialization facility provides functionality for encoding and decoding an object's runtime state. Serialization and deserialization operate in a *stream*-based medium, where a stream can be anything from a byte array to a file descriptor or network socket. Thus, a serialized object can be sent across the network or

saved to secondary storage. Because serialized objects are stored in a platform-independent manner, they can be deserialized inside any JVM.

Because the serialization facility is defined in terms of the generic class `Object`, it can be applied to instances of *any* class. Serializability for a class is enabled by implementing the Java interface `Serializable`. This involves nothing more than adding the words "`implements Serializable`" to the declaration of a class as the `Serializable` interface contains no methods: it is used only to "turn on" serialization for a class. However, the default implementation of serialization only captures "state" as it is represented by an object's data members. Because much of a thread's effective runtime state is hidden away in the native code of the interpreter, serialization is, by default, not enabled for threads. Luckily, Java provides hooks for overriding the default serialization mechanism so that users may provide custom implementations. These hooks involve writing custom implementations of serialization and deserialization for a class, in methods named `writeObject` and `readObject` respectively.

We felt that the logical correspondence between serialized objects and checkpointed threads made a strong case for providing checkpointing syntax in terms of serializability. Therefore, we extended the `Thread` class to implement the `Serializable` interface. To account for the unique nature of `Thread` objects, we have provided implementations of the `Thread` class methods `writeObject` and `readObject` so that serialization of a thread object now captures its *entire* runtime state[2]. The syntax for creating a thread checkpoint is now the same as the syntax for serializing an object. Another consequence of this design is that our checkpointing mechanism can be applied to any form of Java threads, including user-defined subclasses of `Thread`.

5.3 Thread State

A Java thread's state can be broken down into three components: code, data, and stacks. Code refers to the Java *bytecode* that implements a thread's class and the classes of its members. Data denotes an object's source-level runtime data, meaning the values of its members. This particular state component required no work in the implementation of Phantom, as it is exactly the sort of data that the default serialization facility

[2]The components of a thread's state are discussed in depth in Section 5.3.

is designed to handle. Finally, a running thread has state associated with its execution. This takes the form of execution and operand stacks that are stored in native form inside the JVM.

5.3.1 Class Bytecode

One problem faced in migrating an object from one JVM to another is that the two JVMs may not share a consistent *class-space*. For example, if the thread being migrated is actually an instance of a user-defined subclass of `Thread`, that subclass may not exist on the target JVM. An unfortunate consequence of this is that instances of an undefined class cannot be deserialized. This forces Phantom to migrate code in addition to data.

In Java, new classes are created through `ClassLoader` objects. Class loaders operate on a class' compiled bytecode, which is usually found in a `.class` file located somewhere in the local system directory. The API for loaders unfortunately presents asymmetric functionality. Mechanisms do exist for defining a new class based on a byte stream that contains the class' compiled bytecode. However, there is no complementary mechanism for obtaining the bytecode that defines a given class. We were forced to slightly *modify*[3] the JDK internals to provide this functionality.

Because the operation of creating of a new class is parameterized by the class' bytecode, we simply store a copy of the bytecode, before it is parsed, at the time of creation. We modified the native JDK class structure (`Classjava_lang_Class`) to hold the bytecode, and provided access to it by adding the native method `getBytecode()` to the `Thread` class. While we wished to avoid modifying the Java `Thread` API, overriding serialization for threads demanded this step regardless.

Phantom's actual mechanism for code migration is implemented by subclasses of the `ObjectOutputStream` and `ObjectInputStream` classes, which are the fundamental streams of the object serialization facility. These streams provide convenient hooks for performing per-class operations during serialization and deserialization. An *annotation* for a class can be passed along with the stream containing instances of the class. In

[3]Every other aspect of this project was implemented as an *extension* to the JDK's native functionality. Providing access to class bytecode was the one case in which we actually modified a portion of the existing implementation.

Phantom, the annotation for a class is simply its class bytecode, which, based on the documentation, appears to correlate with the intended use of annotations [2]. At deserialization, a class' annotation is *resolved* (used to define the class) before any instances of the class are deserialized.

5.3.2 Execution State

The third and most complicated component of a thread's state is its execution and operand stacks (see Figure 1). These can be thought of abstractly as constituting the runtime state of the virtual machine. Just as a native architecture would have low-level runtime stacks, the virtual machine maintains one *execution stack* per thread, each of which contains one stack frame for every layer of (interpreted) method invocation. In addition, each stack frame has a corresponding *operand stack*, which can potentially contain references to Java objects. Because the execution stack and the operand stacks are implemented as native structures within the JVM, they must be traversed and analyzed at the native level.

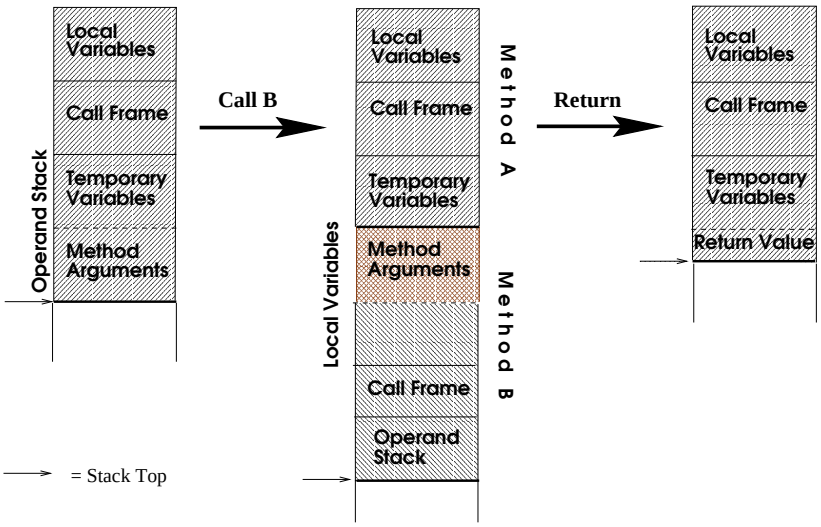

Figure 1: JDK Stack Conventions

One of the greater challenges of this project was decoding the JDK stack frame structure and calling conventions. These are illustrated in Figure 1. The formal behavior of method invocation is that a method's arguments must be on the top of the operand stack at the point of invocation. Inside the callee method, the input arguments are defined to reside in the first n local variable slots. This operation is optimized in

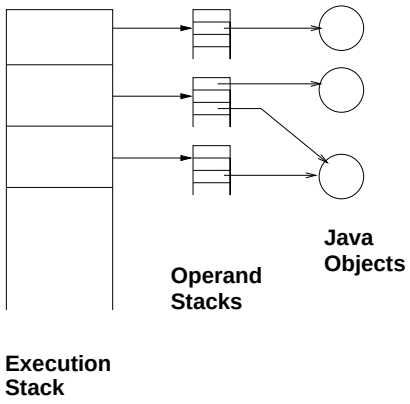

Figure 2: Object References

the JDK so that the caller's opstack is rolled back and the callee's frame is defined around the preexisting input arguments. The effect is analogous to the Sparc architecture's register windows, in that the method arguments never actually move; they obtain new logical handles (as local variables of the callee's frame) without changing physical location.

Because the JVM is a stack-based architecture, it uses operand stacks in place of a register file. Each execution stack frame has one operand stack or "opstack", which is modified in some form by almost every Java instruction. Unfortunately, the JDK implements opstacks as a buffer of generic data words, with no type information whatsoever. The reason for this is that all bytecode is analyzed by the JVM at load-time to verify that its instructions execute in a correct and type-consistent manner, after which high-level type information is no longer needed. The lack of type information posed a significant problem for our implementation, since Phantom must be able to decompose opstacks at any arbitrary point in time.

To generate the opstack type information required for serialization, we perform static **dataflow analysis** of JDK bytecode. While the JVM does not maintain dynamic opstack types, the contents of the opstack at a specific point in a method's execution can be inferred from the method's bytecode instructions. Our dataflow mechanism starts at a method's first instruction and performs an efficient[4] depth-first traversal of all the control paths in the method until the desired instruction (the current PC) is reached. The mechanism keeps track of opstack "side effects" that are incurred by instructions encountered along the way, such that

[4]Each instruction is visited at most once.

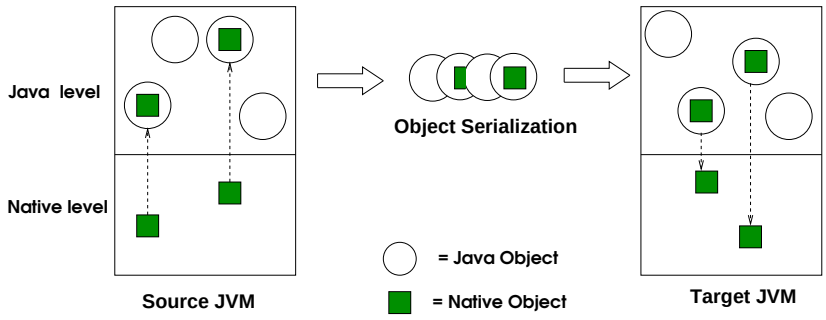

Figure 3: Temporary Object Migration

when the search terminates, it returns a complete type description of every element on the opstack.

This mechanism is based on an important bytecode constraint that is defined in the JVM specification [6]. Specifically, when control reaches a particular bytecode instruction, the opstack is required to have the same number and types of elements regardless of which control path it followed to reach that instruction. In other words, each instruction in a method has exactly one corresponding opstack type signature.

One thing to note here is that our use of dataflow is really a static analysis technique being applied on a dynamic basis, which is suboptimal. Another note is that our dataflow mechanism operates on "quicked" JDK bytecode. As an optimization, Sun's JDK actually rewrites some bytecode instructions the first time they are executed [6], replacing them with non-standard "quick" versions. Quick instructions will often perform very basic optimizations, such as replacing a table index with a direct memory offset to reduce the overhead of indirection[5]. While the official Java instruction set is standardized, the JDK quick instruction set is not. However, adding dataflow support for future additional JDK instructions would be trivial as the stack side-effects of instructions are well defined.

Once the types of the operand stack have been determined via dataflow analysis, the operands and other frame components are repackaged as Java objects and passed back up to the Java layer for serialization, as described in the next section.

[5]Note that quicked bytecode is used only for type analysis. Code migration serializes a class' original (pre-quicked) bytecode.

5.3.3 Object References

Operand stacks are also problematic because they can contain references to Java objects. This is further complicated by the fact that opstacks are only accessible from native code, in the context of which object references are no more than pointers. This begs the question of how to recreate references in a target (recreated) stack, given that: (i) the address space layouts on different hosts are almost guaranteed to be different; (ii) multiple references to the same object can exist and should be recreated as such (as shown in Figure 2); (iii) some objects, such as temporary variables, exist and are allocated purely within the context of native code, such that they never exist in a visible way at the Java level.

We anticipated that this issue would require a complicated mechanism for reference hashing, with address space mappings and so forth. Our actual solution was much more elegant. The reason that multiple references are not an issue for objects created within Java code is that the object serialization facility recursively grabs the entire tree of objects (members, members of members, etc.) underneath the top-level "serialized" object. Serialization already takes care of multiple references and address space layouts, as well as providing a machine-independent intermediate representation. Accordingly, our solution to the problem of object references in the stack was to pass them up to the Java level where they are serialized, as shown in Figure 3. At the target, the stack references are passed back down to the native layer, where they are reintegrated into the target stack as pointers. This is somewhat nonintuitive because we are making full-fledged Java-level objects out of structures that previously existed only at the native level.

5.4 Native Execution

The JVM provides functionality for invoking *native* methods, meaning a function that was natively compiled, during the execution of which the interpreter essentially relinquishes control to the underlying host architecture. Native methods can be explicitly written and linked by users or they can be generated dynamically by just-in-time (JIT) compilers. For example, a Java program can call a function that was written in C and compiled locally. Unfortunately, as soon as a native method is invoked in a thread, that thread's runtime

state consists not only of Java code, data, and JVM stacks, but also the platform-specific architectural elements of low-level stacks, registers, and machine instructions. This throws us straight into the mess of issues associated with heterogeneous native migration schemes, which is exactly what we had intended to avoid with a heterogeneous interpreted system [12].

To avoid handling migration during native execution, we use the breakpoint functionality that is provided by native code in the JVM and is exposed to user code in the JVM debugger interface [4] of versions 1.2 and later of the JDK [3]. When serialization (checkpointing) is attempted on a suspended thread, Phantom checks to see if the thread is currently inside a native method, a fact which is represented by a Java `invoke` instruction. If so, we insert a breakpoint at the next Java instruction, the one immediately following the `invoke` instruction, and then resume the thread. When the native method completes, control is returned to the interpreter and the breakpoint is encountered in the next instruction. Our breakpoint handler then re-suspends the thread and proceeds with serialization. Intuitively, `suspend` operations used in migration have the effect of "falling through" native methods, to stick at the first possible preemption point, namely the next Java instruction.

5.5 The JVM Debugger Interface

As part of their JDK 1.2 beta releases, JavaSoft started distributing the new JVM Debugger Interface (JVMDI) [4]. This interface provides native access to many of the low-level structures of the JVM, and is intended to provide the core functionality necessary to implement a Java debugger. We noticed at the start of our project that many of our anticipated functionality requirements were either already provided by the JVMDI or were to be included in future versions. Specifically, support for accessing operand stack elements was commented as "to be implemented". In the hopes that our project might see some real use as part of a future JDK release, we offered to contribute to the continued implementation of the JVMDI. As a result of this, much of Phantom's development took the ironic path of first expanding the JVMDI by exposing key functionality, and then writing code using the newly created interfaces.

In the same vein, we have made our implementation as "non-intrusive" as possible, meaning that we

have intentionally minimized the modifications that Phantom makes to the existing JDK implementation. With one exception, we have been successful in implementing our migration facility solely by extending the existing native functionality of the JVM. In other words, the functionality that we have added to the JVMDI does not modify any of the internal workings of the JVM, it merely adds code to its implementation.

6 Results

Unfortunately, we were not able to get all of Phantom's components working properly in time for this paper. Specfically, the code for reinstantiating a thread's execution state on the target JVM proved to be more complicated than we had anticipated. This is largely due to main loop of the Java interpreter, which caches key elements of state in local variables. While we thought we could work around this by manipulating bytecode instructions, we were not able to implement our solution in time. We believe this is not a fundamental design problem, merely one that requires more time to get right. In short, while the majority of the Phantom's components work correctly, the final step eluded us and therefore we were never able to get a complete migration mechanism functioning.

Accordingly, we provide some rough performance data based on isolated measurements of Phantom's working components, operating on a very simply thread with three levels of method invocation. We believe that these measurements will be roughly equivalent in the final working version of Phantom, which we hope to complete on our own. The time to encode a thread's execution state is roughly 110 ms, with 8 ms (on average) spent synchronizing on the next Java instruction boundary, and approximately 100 ms spent performing dataflow analysis and walking the JVM stacks. Recreating the stacks takes anywhere from 130 to 180 ms. Because the system was never completely assembled, we have no information on the amount of time necessary for serialization.

7 Future Work

In this section, we explore possible extensions to our work. These were beyond the scope of this project because of the time constraints involved. As such, the main focus for our work was providing a working infrastructure for process migration. Turning Phantom into more of a production load balancing system would first require investigating the issues of migration policies and distributed process management, as well as a careful design of such subsystems to make them perform efficiently. We intentionally ignored these issues in our implementation as they were beyond the scope of our time-constrained project. Beyond questions of performance, future work on Phantom would have to address the problem of external references. Phantom's current lack of support for external references renders it unusable for several classes of applications.

7.1 Simple Migration

The most obvious area of future work is to get the original Phantom mechanism working. The authors are continuing to pursue this goal on their own time.

7.2 External References

Phantom provides no support for maintaining consistent *external references* in migrated threads. An external reference is defined simply as any system state that transcends the runtime state of an individual thread. External references can exist between a thread and the JVM, such as file descriptors, or between two threads, such as a network connection or synchronization object.

7.2.1 I/O

File and network references could be accounted for using a *shadow* process scheme similar to Condor [7]. Such a mechanism would leave a degenerate process running on the source host in place of the migrated thread. This shadow thread would serve as a "home base" to facilitate process interactions that arrive for the thread at its previous location, as well as providing a means for a migrated thread to invoke environment-specific

operations from its original execution context.

7.2.2 Naming

The possibility of maintaining network I/O paths across migration would also force Phantom to address the issue of naming. Some sort of scheme for uniquely naming a thread, independent of its current location, would need to be implemented. The notion of Java *thread groups* brings up the same issue. A thread can be logically associated with a group of other threads at the time of its creation. The group relationship, and any operations on that group, should ideally remain consistent even in the face of migration. Again this hints at a need for some form of distributed thread management and identification.

7.2.3 Synchronization

Perhaps the most complicated form of external references to take into consideration is synchronization. Java provides very flexible synchronization semantics, whereby any object can be used as a synchronization lock or monitor. Consequently, a migrating thread could possess several forms of synchronization dependencies. The most straightforward problem scenario is one in which a thread is holding a lock that another thread is waiting on. When the first thread is migrated the second thread should remain blocked. Likewise, when the first thread finally does relinquish the lock (at its new location), that fact should propagate back to the second thread such that it then acquires the lock.

7.2.4 Shared Memory

Another complicated form of external references is shared memory, where two threads each contain a reference to a common object. Research from the area of distributed shared memory (DSM) [11] could likely be applied to this scenario.

7.2.5 Dynamic Libraries

Yet another type of external reference that would pose a problem is the use of dynamically loaded native libraries. Users can define their own native methods whose implementations are obtained by linking in a natively compiled library at runtime. This is in fact how the JVMDI is accessed. However, this presents a problem for migration as it assumes not only that the library is present on the local machine, but that it is compiled for the local architecture, meaning that the system cannot always just fetch the corresponding library from a thread's original host. A variety of mechanisms could be used to handle this scenario. One is to simply implement a native heterogeneous system which accounts for native code differences. Another is to allow threads to migrate only between hosts of the same platform, effectively restricting it to homogeneous migration. Yet another is to temporarily re-migrate the thread back to its original host for the duration of the native execution. All of these solutions have problems that would need to be resolved by additional research.

Native code produced by a just-in-time (JIT) compiler could be handled by falling back on bytecode. While a method can be be compiled natively for faster execution, the original bytecode can always be refound and substituted in at migration time. In terms of preemptability, JIT compiled code would be equivalent to native methods, in that migration could only occur after such a method returned.

8 Conclusion

Using the Java virtual machine as an infrastructure, we have implemented the Phantom heterogeneous migration system. Phantom successfully migrates Java threads between JVMs running on different platforms. We laid out details of our implementation, and showed that our mechanism is transparent with respect to code and runtime behavior. We further described that Phantom migration is preemptable at the granularity of individual Java instructions. Migrating thread state requires handling potentially complicated object references both at the language and native levels, which we solved by leveraging off Java's object serialization facility. Interesting by-products of the Phantom project include: (i) a mechanism for Java thread checkpoint-

ing, (ii) a dataflow analysis tool for "quicked" JDK 1.1 bytecode, and (iii) extensions to the JVM debugger interface. In this paper, we described in significant detail how Java's features could be exploited to create a migration mechanism like Phantom. Having done this work in cooperation with JavaSoft's engineers, we hope to see parts of our project reused in a future version of Java.

References

[1] "Java Object Serialization Specification", Sun Microsystems, Inc., October 1997.

[2] "Java Platform 1.1.5 Core API Specification", Sun Microsystems, Inc., 1997.

[3] "Java Platform 1.2 Beta 2 API Specification", Sun Microsystems, Inc., 1997.

[4] "Java Virtual Machine Debugger Interface Reference", Sun Microsystems, Inc., December 1997.

[5] D.B. Lange and D.T. Chang, "Programming Mobile Agents in Java", IBM Corporation white paper, September 1996.

[6] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification", Sun Microsystems, Inc., 1996.

[7] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpointing and Migration of UNIX Processes in the Condor Distributed System", University of Wisconsin-Madison, Computer Sciences TR #1346, April 1997.

[8] "ObjectSpace Voyager Core Package Technical Overview", ObjectSpace Inc. white paper, December 1997.

[9] M.L. Powell and B.P. Miller, "Process Migration in DEMOS/MP" *9th Symposium on Operating Systems Principles*, Bretton Woods, NH, October 1983, pp. 110-119.

[10] T. Redhead, "A High-level Process Checkpointing and Migration Scheme for Heterogeneous distributed Systems", CRC for Distributed Systems Technology, University of Queensland.

[11] D.J.Scales and K.Gharachorloo, "Towards Transparent and Efficient Software Distributed Shared Memory" *16th Symposium on Operating Systems Principles*, Saint Malo, France, October 1997, pp. 157-169.

[12] P. Smith and N.C. Hutchinson, "Heterogeneous Process Migration: The Tui System", University of British Columbia, Computer Sciences TR #96-04, February 1996.

[13] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facility for the V-System" *10th Symposium on Operating Systems Principles*, Orcas Island, WA, December 1985, pp. 2-12.

[14] E. Zayas, "Attacking the Process Migration Bottleneck" *11th Symposium on Operating Systems Principles*, Austin, TX, November 1987, pp. 13-24.