Termination and Rollback in Language-Based Systems

by

Algis Rudys

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Dan S. Wallach, Chair, Assistant Professor
Computer Science

Robert Cartwright
Robert Cartwright, Professor
Computer Science

Scott Rixner
Scott Rixner, Assistant Professor
Computer Science

HOUSTON, TEXAS

DECEMBER, 2002
Termination and Rollback in Language-Based Systems

Algis Rudys

Abstract

Language run-time systems are routinely used to host potentially buggy or malicious code in a secure environment. For this thesis, we address a single aspect of the resource-management problem, the problem of code termination, including the related issue of restarting previously terminated code. We provide formal analysis, including proofs of the correctness of our termination system for an idealized language. We also consider implementations of our systems in Java using bytecode rewriting. The performance impact of these systems ranged from relatively modest (3 to 25% overhead for soft termination), to severe, although perhaps unavoidable (6 to 23× overhead for transactional rollback). We discuss the technique of bytecode rewriting as used here to modify the semantics of a language. Our goal is to provide a mechanism based on code-to-code transformations with semantics for codelets in a run-time system similar to the semantics provided by operating systems for managing processes.
Acknowledgments

Chapter 2 is based on "Termination in Language-Based Systems," presented at the 2001 Network and Distributed Systems Security Symposium [67], and later revised for publication in ACM Transactions on Information and System Security [68]. Chapter 3 is based "Transactional Rollback in Language-Based Systems", published at the 2002 International Conference on Dependable Systems and Networks [69].

I would like to thank Dan Wallach for his guidance, support, and criticism as my advisor. Thank you to my thesis committee, Dan Wallach, Robert Cartwright, and Scott Rixner, for volunteering their time and expertise. I would also like to thank John Clements, Jiangchun “Frank” Luo, Liwei Peng, and David Price for their contributions to this project. Thank you also to Matthias Felleisen, Shriram Krishnamurthy, Natarajan Shankar, and Drew Dean for their comments, and thanks to Willy Zwaenepoel and Dave Johnson for helpful discussions. In addition, thanks to the many anonymous reviewers who contributed to improving this work. Finally, thanks to my family and friends for their support and encouragement.
# Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables ix

1 Introduction 1
  1.1 Related Work .............................................. 4
    1.1.1 Language-Based Security .............................. 4
    1.1.2 Persistent Systems .................................... 6
    1.1.3 Bytecode Rewriting .................................... 7

2 Soft Termination 9
  2.1 System design ............................................... 9
    2.1.1 Naïve Termination .................................... 9
    2.1.2 Hard Termination ..................................... 10
    2.1.3 Soft Termination ..................................... 11
    2.1.4 Soft Termination with Codelet, System, and Blocking Code .... 20
  2.2 Java Implementation ...................................... 29
    2.2.1 Termination Check Insertion ......................... 30
    2.2.2 Control Flow ......................................... 31
    2.2.3 Blocking Calls ....................................... 33
    2.2.4 Invoking Termination ................................ 34
    2.2.5 Optimizations ........................................ 35
    2.2.6 Synchronization ...................................... 36
    2.2.7 Thread Scheduling .................................... 37
2.2.8 System Code Safety ........................................... 37
2.2.9 Memory Consistency Models ................................ 37
2.2.10 Defensive Security ........................................... 37
2.3 Performance ....................................................... 38
2.3.1 Microbenchmarks .............................................. 38
2.3.2 Application Benchmarks ...................................... 40
2.3.3 Termination Check Overhead ................................ 42
2.4 Summary ............................................................ 43

3 Transactional Rollback ............................................ 44
3.1 System Design ...................................................... 44
3.1.1 Rollback and Transactions ................................... 45
3.1.2 Architecture of Transactional Rollback .................... 45
3.1.3 Design Discussion .............................................. 49
3.2 Java Implementation .............................................. 52
3.2.1 Locking Code Insertion ...................................... 53
3.2.2 Lock State Sharing ............................................ 56
3.2.3 Arrays ............................................................ 57
3.2.4 Constructors .................................................... 57
3.2.5 Native Methods ................................................. 58
3.2.6 Open Files ....................................................... 59
3.2.7 Optimizations ................................................... 59
3.2.8 Integration into a Real-World System ....................... 60
3.3 Performance ......................................................... 62
3.3.1 Microbenchmarks .............................................. 62
3.3.2 Application Benchmarks ..................................... 63
3.3.3 Optimization Profiling ....................................... 65
3.4 Summary ............................................................. 66

4 Bytecode Rewriting ............................................... 68
4.1 Bytecode Rewriting Implementations .................................. 68
  4.1.1 SAFKASI .................................................. 69
  4.1.2 Soft Termination ............................................ 70
  4.1.3 Transactional Rollback ...................................... 70
4.2 JVM Design Issues .................................................. 71
  4.2.1 JVM Internals ............................................... 71
  4.2.2 “Local” Java Semantics ..................................... 74
  4.2.3 “Global” Java Semantics .................................... 78
4.3 Optimizing Bytecode Rewriting ..................................... 82
4.4 Summary ............................................................. 83

5 Conclusions ............................................................ 85

Bibliography .............................................................. 87
## Illustrations

2.1 Simple language used for our analysis. ........................................ 12
2.2 Evaluation context for reduction of the simple language. ............... 13
2.3 An operational semantics for our language. ................................. 13
2.4 Lexical scoping rules for expanding \([V / x] M\) for any expression \(M\). .................................................. 14
2.5 The steps in evaluating a program \(P\). ........................................ 15
2.6 The soft termination transformation. ........................................... 16
2.7 An extended language for analysis. .............................................. 20
2.8 Evaluation context for reduction of this extended language. ............. 21
2.9 Operational semantics for the extended language. .......................... 21
2.10 The soft termination transformation for this extended language. ....... 22
2.11 The soft termination transformation applied to any function definition. . 30
2.12 The soft termination transformation applied to any loop. ............... 31
2.13 The soft termination transformation applied to a blocking call. ...... 32
2.14 Soft termination overhead for microbenchmarks. ........................... 39
2.15 Soft termination overhead for application benchmarks. .................. 41

3.1 Object structure before a transaction has begun operation on the object. . 50
3.2 Object structure after a transaction has read from the object. ............ 50
3.3 Object structure after a transaction has written to the object. ............ 50
3.4 Transactional rollback transformation applied to method definitions and 
method calls in Java. ......................................................... 53
3.5 Transactional rollback transformation applied to object field accesses 
(read and write). .................................................................. 54
3.6 Transactional rollback transformation applied to an arbitrary class. .... 55
3.7 Transactional rollback transformation applied to a code block which starts a transaction. .......................... 61
3.8 Transactional rollback overhead for microbenchmarks. .......................... 63
3.9 Transactional rollback overhead for application benchmarks. .......................... 64

4.1 How a Java bytecode transformation changes the process of loading a class. 69
Tables

2.1 Average total number of termination checks performed for each benchmark. 42

4.1 Summary of the three systems based on bytecode rewriting. . . . . . . . . . 72
Chapter 1

Introduction

Language run-time systems offer several advantages over operating systems running traditionally compiled programs. They enable the same program to run on different hardware and operating system platforms; any portability issues need only be addressed once, in the development of the language run-time system implementation. They also minimize context switching and inter-program communications overhead, since the programs are all running in the same operating system process and memory space.

Code written to be distributed over networks and run in language run-time system is known as mobile code. A well-known example of mobile code is Java applets. Java applets have been around since 1995, when Netscape integrated the Java virtual machine (JVM) into its Navigator 2.0 browser. In addition to its applications within web browsers, mobile code has also been touted for OS kernel extensions, active networking, extensible databases, agent-based negotiation systems, device identification and configuration, and other problem domains.

One aspect common to many of these systems is that they routinely host potentially buggy or malicious code from external sources that are not necessarily trusted. In addition, they may run code from multiple sources where the sources do not trust one another. We use the term codelet\(^1\) to describe a program from a single source and executed in conjunction with or as an internal component of a larger program. Maintaining a secure environment for such code requires some control over the execution of these codelets. At its most basic level, this means the ability to mediate between a codelet and the potentially dangerous primitives it is allowed to call, as well as the ability to stop and start codelets at will.

---

\(^1\)The term “codelet” is also used in artificial intelligence, numerical processing, XML tag processing, and PDA software, all with slightly different meanings.
The designers of Java [42, 53] recognized this need for security, and integrated language-based enforcement of security into the Java virtual machine. Java promises an environment where untrusted buggy or malicious code can run safely inside the Web browser, enhancing the user’s Web experience without jeopardizing the user’s security. Rather than using kernel-based protection, the JVM runs inside the same address space as the browser, providing protection and separation as a side effect of enforcing its type system. A combination of static and dynamic type checking serves to prevent a malicious codelet from forging a reference to an arbitrary memory location or otherwise subverting the system.

The promise of Java may be attractive, but a large number of security flaws have been discovered since its release [24, 57]. Significant strides have been made at understanding the type system [3, 74, 31, 32, 23, 17] and protecting the Java system classes from being manipulated into violating security [80, 41, 33, 34], but efforts to control resource exhaustion have lagged behind. A simple infinite loop will still freeze the latest web browsers. The most successful systems to date either run the JVMs in separate processes or machines [55, 73], surrendering any performance benefits from running the JVM together with its host application, or create a processlike abstraction inside the JVM [7, 77, 43, 8, 20]. These process abstractions either complicate memory sharing among codelets or make it completely impossible.

This thesis explores a series of new language run-time system-based mechanisms for terminating codelets. Although they are not specific to Java, they apply to Java, and we have implemented them in Java as a case study. Our designs are based on code-to-code transformations, and are therefore easy to port across languages. The basic termination mechanism, discussed in Chapter 2, is called soft termination. Soft termination is intended to be invoked either by an administrator or by a system resource monitor which has concluded that a codelet is exceeding its allotted resources and should no longer be allowed to run. It guarantees termination of a codelet by breaking any potential infinite loops in the codelet. Soft termination provides semantics similar to UNIX’s ps and kill commands, yet requires neither processlike structures nor limits on memory sharing.

Language run-time systems also suffer the dual problem of restarting terminated codelets. Language run-time systems allow and encourage unrestricted data sharing among
codelets; these shared data structures could be in an inconsistent state when the codelet is terminated. These inconsistencies can destabilize other running codelets as well as make it complicated to restart a terminated codelet.

To solve this problem, we observe that many codelet systems, such as component-based web servers (e.g., a Java Servlet web server), execute in a transactional style, spawning codelet instances to independently service requests as they arrive. We can track the changes the codelets make to any state; if a codelet is terminated, we roll back all of its changes. If the codelet is later restarted, the system is not in some intermediate state that might cause instability. Because of the nature of the rollback operation, codelets must be run as transactions in order to maintain the consistency of the state. Note, however, that we are not saving the codelet’s call stack or execution state — that is, we are not saving a continuation. When the codelet is resumed, this transient state is initialized from scratch. Only the codelet’s persistent state is saved. Similarly, transactional rollback only deals with operations on memory, and doesn’t address network- or file-based state. From the perspective of writing codelets, our design only requires the addition of transaction start and commit instructions to the system that hosts codelets. We expect these transactional instructions would be integrated in the event dispatching mechanism used in such systems as a Java Servlet-based Web server. No other changes would need to be visible to the codelet programmer. Transactional rollback is discussed in Chapter 3.

We designed both of these systems using code-to-code transformations. Code-to-code transformation is a powerful technique from language theory for designing new semantics for a language. The designs lead directly to implementations based on source code rewriting. The resulting systems are portable across implementations of a language runtime system, and the designs can be applied with minimal modifications to other languages entirely.

Bytecode rewriting is used to implement code-to-code transformations for Java. Since bytecode rewriting makes no modifications to the Java virtual machine, the resulting systems can be used on all implementations of the Java virtual machine. We used bytecode rewriting to implement both of these systems for testing purposes. Chapter 4 discusses some lessons we have learned in the design and implementation of systems based on code-
to-code transformations and bytecode rewriting. These lessons should be useful in designing and implementing other systems based on bytecode rewriting.

1.1 Related Work

This thesis inherits from existing research in a number of areas. Soft termination builds on existing work in language security. Transactional rollback, in addition to its relation to language security, also derives from previous work in persistent object systems. Finally, this thesis depends on work done in bytecode rewriting and code-to-code transformation.

1.1.1 Language-Based Security

Systems such as Smalltalk [40], Pilot [66], Cedar [76], Lisp Machines [11], and Oberon [82] have taken advantage of language-based mechanisms to provide OS-like services. At least as early as the Burroughs B5000 [13] series computers, language-based mechanisms were being used for security purposes. At least as early as the University of California at Berkeley’s Informer system [29], code rewriting has been used to enforce security properties. More recently, language-based enforcement of security has been popularized by Java, originally deployed by Netscape for its Navigator 2.0 browser in 1995 to run untrusted applets.

However, these systems provide little or no support for resource management on the programs they run. A number of projects have been developed to address this. A recent Scheme system called MrEd [38] supports thread termination and management of resources such as open files but has no way of disabling code from running in future threads. Some systems, such as PLAN [44], restrict the language to guarantee that programs will terminate.

In general, many language systems support interactive debugging, which includes the ability to interrupt a running program at any point. This can be performed with operating system services or by generating inline code to respond to an external debugger. The UNIX ptrace(2) [75] system call is an example of such an operating system service. Among other features, it allows any process to suspend another, subject to operating system permissions, and step through the target process instruction by instruction. However,
ptrace(2) depends for its functionality on the separation provided by operating system processes.

Much of the recent research in this area has been focused on the Java programming language. PERC [60], for instance, is an extension to Java that supports asynchronous exceptions while providing a mechanism for protecting system invariants in critical sections. A programmer may specify blocks with provable limits on their run-time and asynchronous exceptions are deferred while execution is in one of these blocks.

Chander et al. [14] describe a system to target specific sorts of resource exhaustion attacks via bytecode instrumentation. The general technique they present is to replace calls to sensitive methods (for instance, for setting thread priority or creating a new window) with calls to customized methods that first verify that the operation is not harmful. Although such a mechanism is effective, it is very specific, requiring a new method to be written by hand for each potentially harmful method. In addition, this system cannot prevent resource exhaustion within a user codelet (for instance, an infinite loop), and it fails to address termination.

J-Kernel [43] is a system for managing multiple Java codelets running in the same JVM. It is written entirely in Java, giving it the advantage of working with multiple JVMs with minimal adjustment. It is implemented as a transformation on Java bytecode as class files are loaded by the system. J-Kernel isolates threads to run within specific codelets; cross-domain calls are supported via message passing from one codelet thread to another or to the system. By isolating threads to their codelets, it becomes safe to arbitrarily deschedule a thread. Such a system necessarily restricts data sharing between codelets.

JRes [21] is a resource management system for Java. Bytecode rewriting is used to instrument memory allocation and object finalization in order to maintain a detailed account of memory usage. Again, termination is mentioned, but no specific details are provided.

J-SEAL2 [9] is a framework, written entirely in Java, for running Java codelets. Extended bytecode verification, combined with a limited degree of bytecode rewriting, is used to ensure that codelets do not violate protection domain boundaries. The result is that sharing among codelets is restricted. Termination is guaranteed by a bytecode transformer that effectively prevents the codelet from catching a particular type of exception.
KaffeOS [5, 6] provides an explicit processlike abstraction for Java codelets. It is implemented as a heavily customized JVM with significant changes to the underlying language run-time system and system libraries. Code termination is supported in the same manner as a traditional operating system: user codelets are strongly separated from the kernel by running in separate heaps. Memory references across heaps are heavily restricted. The multitasking virtual machine (MVM) [20] also uses JVM extensions to support separation of codelets, primarily to prevent the codelets from interfering in each other’s execution. Bernadat et al. [8] and van Doorn [79] describe similar systems that customize a JVM in order to support better memory accounting and security. van Doorn takes advantage of lightweight mechanisms provided by an underlying microkernel. These systems provide a style of termination we call hard termination (see Section 2.1.2).

1.1.2 Persistent Systems

Databases have always been capable of operating on data, generally through queries. More recently, programming languages have supported orthogonal persistence. This is the notion that all data types, whether stored persistently or temporarily, are treated equivalently in the language. PS-algol [4] and Elle [1] were among the first programming languages to support orthogonal persistence. Napier88 [26] is a more recent example.

Persistent object systems have been around nearly this long. Persistent object systems provide orthogonal persistence in object-based environments. One of the earliest is POMS [16], the Persistent Object Management System. In this system, based on PS-algol, persistent and transient data are indistinguishable. POMS was followed up by CPOMS [12], a layer written in the C programming language that also provided persistence for PS- Algol.

Thor [54] is a more recent example of a persistent object system. It is an early example of systems that were not bound to a single programming language. In addition, Thor guarantees the integrity of the object store even when used by an unsafe language such as C++. It can provide this guarantee by only allowing access to persistent objects via their methods. The persistent objects themselves are implemented in Theta, a type-safe language.
More recently, Java has been the target of persistent systems research. PJama [64] is one persistent object system that has been developed for Java. PJama maintains an object store parallel with the system heap. Objects become persistent automatically by reachability from an object which is already persistent, or by being explicitly declared persistent. As objects become persistent, they are migrated to this object store, to eventually be written to disk.

Much of the current development in persistence-related techniques is also taking place around PJama. One example is Daynès and Czajkowski’s lock state sharing mechanism [22]. Hosking et al. [45] discuss several techniques for optimizing read and write barriers, using PJama as a basis. Finally, a number of novel ideas for implementing transaction management, explicitly transient state, and checkpointing are all based on PJama.

More recently, there has been research on implementing Java persistence outside of the JVM. The goal here is to allow portability across virtual machine implementations. Kutlu and Moss [50] describe a system that uses Java reflection. They do, however, make modifications to the JVM to support more extensive reflection. Marquez et al. [56] describe a persistent object system implemented for Java using bytecode transformations. This requires no modifications to the JVM.

1.1.3 Bytecode Rewriting

Java bytecode rewriting has been applied in far too many other systems to provide a comprehensive list here. Most notably, it is used in SAFKASI [80], which is discussed in Chapter 4. It is also used in numerous other settings for a wide variety of purposes.

Access Control. By intercepting or wrapping calls to potentially dangerous Java methods, systems by Pandey and Hashii [62], Erlingsson and Schneider [35], and Chander et al. [14] can apply desired security policies to arbitrary codelets without requiring these policies to be built directly into the Java system code, as done with Java’s built-in security system.
Resource Management and Accounting. J-Kernel [43] and J-SEAL2 [9] both focus primarily on isolation of codelets. Bytecode rewriting is used to prevent codelets from interfering in each others’ operations. JRes [21] focuses more on resource accounting; bytecode rewriting is used to instrument memory allocation and object finalization sites.

Optimization. Cream [15] and BLOAT (Bytecode-Level Optimization and Analysis Tool) [61] are examples of systems which employ Java bytecode rewriting for the purpose of optimization. Cream uses side-effect analysis, and performs a number of standard optimizations, including dead code elimination and loop-invariant code motion. BLOAT uses Static Single Assignment form (SSA) [19] to implement these and several other optimizations.

Profiling. BIT (Bytecode Instrumenting Tool) [52] is a system which allows the user to build Java instrumenting tools. The instrumentation itself is done via bytecode rewriting. Other generic bytecode transformation frameworks, such as JOIE [18] and Soot [78], also have hooks to instrument Java code for profiling.

Other Semantics. Sakamoto et al. [70] describe a system for thread migration implemented using bytecode rewriting. Marquez et al. [56] describe a persistent system implemented in Java entirely using bytecode transformations at class load time. Notably, Marquez et al. also describe a framework for automatically applying bytecode transformations, although the status of this framework is unclear. Kava [81] is a reflective extension to Java. That is, it allows for run-time modification and dynamic execution of Java classes and methods.
Chapter 2

Soft Termination

In this chapter, we introduce and discuss soft termination. Section 2.1 formalizes and describes what we mean by soft termination. Section 2.2 describes our Java-based implementation of soft termination, and mentions a number of Java-specific issues that we encountered. We present performance measurements in Section 2.3. We provide a summary of this chapter in Section 2.4.

2.1 System design

A large number of possible designs exist for supporting termination in language run-time systems. We first consider the naïve solutions and explain the hard problems raised by their failings. We then discuss how operating systems perform termination and finally, describe our own system.

2.1.1 Naïve Termination

One naïve solution to termination would be to identify undesired threads and simply remove them from the thread scheduler. This technique is used by Java’s deprecated Thread.destroy() operation.\(^1\) Unfortunately, there are numerous reasons this cannot work in practice, which led Sun to deprecate this method.

Critical sections. A thread may be in a critical section of system code, holding a lock, and updating a system data structure. Descheduling the thread would either leave the system in a deadlock situation (if the lock is not released) or leave the system data

\(^1\)For more information, see http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html
structures in an undefined state, potentially breaking system invariants and destabilizing the entire system (if the lock is forcibly released).

**Boundary-crossing threads.** In an object-oriented system, a program wishing to inspect or manipulate an object invokes methods on that object. When memory sharing is unrestricted between the system and its codelets or among the codelets, these method invocations could allow a malicious codelet to hijack the thread from its caller and perhaps never release it. This is especially problematic if the thread in question is performing system functions, such as finalizing dead objects prior to garbage collection.

**Blocking calls.** Many language run-time systems have functions (especially I/O functions) which block. Blocking calls frequently involve a system call to the operating system. Descheduling a thread which is making a blocking call could cause problems if and when the corresponding system call returns, and should be avoided.

Another naïve solution is to force an asynchronous exception, as done by Java’s deprecated `Thread.stop()` operation. Although this exception will wait for blocking calls to complete, it may still occur inside a critical section of system code. In addition, blocking calls could potentially never return, resulting in a nonterminable thread. Finally, a workaround is needed to prevent user-level code from catching and ignoring the exception.

### 2.1.2 Hard Termination

Operating systems such as UNIX support termination by carefully separating the kernel from the user program. When a process is executing in user space, the kernel is free to immediately deschedule all user threads and reclaim the resources in use. We call such a mechanism a *hard termination system* because once termination is signaled, user-level code may be terminated immediately with no harmful side effects. External resources, such as data files in the file system, may be left in an inconsistent state by this termination. However, in general, these inconsistencies do not threaten stability at the system level.

If the process is executing in the kernel, termination is normally delayed; a flag is checked when the kernel is about to return control to the user process. In cases where the
kernel may perform an operation that could potentially block forever (e.g., reading from the network), the kernel may implement additional logic to interrupt the system call. System calls that complete in a guaranteed finite time need not check whether their user process has been terminated, as the kernel will handle the termination signal on the way out.

2.1.3 Soft Termination

Unlike a traditional operating system, the boundary between user and system code in a language run-time system is harder to define. Although all code within the system is generally tagged with its protection domain, there is nothing analogous to a system call boundary where termination signals can be enforced. Furthermore, because system code might call back to user code, even a thread executing user code might be unsafe to terminate.

This section introduces a design we call soft termination and describes the properties we would find desirable. We present a formal model of soft termination based on code rewriting for a simplified language and prove that all programs will terminate when signaled to do so.

Key Ideas

Soft termination is based on the idea that a codelet may be statically instrumented to check for a termination condition during the normal course of its operation. Our goal is to perform these checks as infrequently as possible—only enough to ensure that a codelet may not execute an infinite loop when termination has been signalled. Furthermore, as with the UNIX kernel, we would like the termination of a codelet not to disturb any system code it may be using at the time it is terminated, so as to preserve system correctness.

The soft termination checks are analogous to safe points, which are used in language environments to insert checks for implementing stack overflow detection, preemptive multitasking, interprocess and intertask communication, barrier synchronization, garbage collection, and debugging functions. A good discussion on safe points is provided by Feeley [36]. In Feeley's terminology, our design uses "minimal polling."
\[ P = \Gamma \ M \]
\[ \Gamma = D \ldots D \]
\[ D = (\text{define} \ (f \ x) \ M) \]
\[ M = (f \ M) \ | \ (\text{if}_0 \ M \ M \ M) \ | \ (\text{let} \ (x \ M) \ M) \ | \ (\text{try} \ M \ M) \ | \ (\text{throw}) \ | \ V \ | \ x \ | \ (\text{CheckTermination}) \]

\[ V = c \in \mathbb{N} \]
\[ f, x = \text{identifiers} \]

Figure 2.1: Simple language used for our analysis.

**Formal Design**

For our analysis, we begin with a simple programming language having natural numbers, functions, conditional expressions, and simple exceptions (see Figure 2.1). In our language, a program is a collection of function definitions (\( \Gamma \)) followed by an expression to be evaluated (\( M \)). An expression can contain function applications as well as primitive operations, conditionals, and exceptions. For simplicity, functions are designed to each take a single natural number parameter (a number of schemes exist for representing multiple natural numbers using a single natural number). Section 2.2.2 discusses our handling of the richer control flow available in Java for our implementation of soft termination.

We write the semantics of our language using the same style as Felleisen and Hieb [37]. Figure 2.2 defines \( E \), the grammar of evaluation contexts for our language. An evaluation context is simply an expression with a subexpression replaced by a “hole” ([ ]). The hole acts as a placeholder in the context; \( E[M] \) represents the result of putting expression \( M \) into the hole of evaluation context \( E \). The hole is consistently located in such a way as to enforce a left-to-right order of evaluation. The reduction rules in Figure 2.3 are applied to these contexts, defining the behavior of the language.

The semantics for our language define three possible ending states: \( V \), \textbf{error}, and \textbf{(throw)}. \( V \) represents a final state in which the program reduces to a value. The \textbf{error} state indicates that some error condition, such as a call to an undefined function, was reached in the evaluation of the program. The \textbf{(throw)} state occurs when the entire program reduces to \textbf{(throw)}; it indicates that the program terminated as the result of an uncaught exception.
\[ E = \emptyset \mid (f\ E) \mid (\text{if}_0\ E\ M\ M) \mid (\text{let}\ (x\ E)\ M) \mid (\text{try}\ E\ M) \mid \text{(CheckTermination)} \mid (\text{throw}) \mid x \]

\text{final states} = V \mid \text{error} \mid (\text{throw})

Figure 2.2: Evaluation context for reduction of the simple language.

\[
E[(f\ V)] \rightarrow \begin{cases} 
E[V_0] & \text{if } \delta(f, V) = V_0 \\
E[[V / x] M]^+ & \text{if } (\text{define}\ (f\ x)\ M) \in \Gamma \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[
E[(\text{if}_0\ V\ M_1\ M_2)] \rightarrow E[0] \text{ or } E[1] \quad \text{(depending on external conditions)}
\]

\[
E[(\text{if}_0\ V\ M_1\ M_2)] \rightarrow E[M_1]
\]

\[
E[(\text{let}\ (x\ V)\ M)] \rightarrow E[[V / x] M]^+
\]

\[
E[(\text{try}\ V\ M)] \rightarrow E[V]
\]

\[
E[(\text{try}\ (\text{throw})\ M)] \rightarrow E[M]
\]

\[
E[(f\ (\text{throw}))] \rightarrow E[\text{(throw)}]
\]

\[
E[(\text{if}_0\ (\text{throw})\ M_1\ M_2)] \rightarrow E[\text{(throw)}]
\]

\[
E[(\text{let}\ (x\ (\text{throw}))\ M)] \rightarrow E[\text{(throw)}]
\]

\[
E[x] \rightarrow \text{error} \quad \text{(unbound variables)}
\]

Figure 2.3: An operational semantics for our language.

\[\uparrow\] The expansion \( [V / x] M \) indicates that every instance of \( x \) in \( M \) should be replaced with the corresponding \( V \), according to the standard rules of lexical scope. This is defined for our language in Figure 2.4.

Although this is similar in nature to \text{error}, it is treated differently to indicate its different origin.

A (\text{CheckTermination}) expression reduces to a Boolean value (0 or 1), indicating whether termination for the current codelet has been externally (and asynchronously) requested. Since (\text{CheckTermination}) behaves as a termination signal, we model it by assuming that when evaluation begins (\text{CheckTermination}) evaluates to 0. Once (\text{CheckTermination}) becomes 1 (indicating that termination has been requested), it continues to be 1 until the codelet terminates.

The last five reductions require some explanation. The first four of these allow for the single-step reduction of (\text{throw}) expressions appearing as subexpressions in any other expression. When (\text{throw}) occurs as the body of a \text{let} or as the second or third parameter
of an if, the default reduction rules already correctly reduce the expression, so no special case is needed.

The last reduction rule reduces a variable, when put into the hole of the evaluation context, to **error**. Note that evaluation of let expressions and function applications replaces all bound variables with their values, according to the rules of lexical scope (see Figure 2.4). As a result, any attempt to evaluate a variable indicates that the variable is unbound and undefined, which is an error.

We include a delta (δ) function which maps primitive names and valid arguments to values. Note that the delta function is simply an abstraction used to represent primitives in this language, and never actually appears in a program. We assume the syntactic property that no name collisions occur; that is, no function is defined more than once, and no primitive function is redefined.

As long as (CheckTermination) remains constant, this language is deterministic. That is, for any given program, when evaluated multiple times, the sequence of reductions applied will always be the same and the program will always terminate in the same final state. The language inherits this determinism from the property that for every syntactically valid program, there is exactly one reduction in the operational semantics which can be applied
Sample program:

\[
P = (\text{define } (f_1 x) (\text{if}_0 x 1 \text{ (throw)}))
\]
\[
(\text{try } (f_1 (f_1 0))) (f_1 0))
\]
\[
\Gamma = (\text{define } (f_1 x) (\text{if}_0 x 1 \text{ (throw)}))
\]
\[
M = (\text{try } (f_1 (f_1 0))) (f_1 0))
\]

<table>
<thead>
<tr>
<th>Step</th>
<th>Expression</th>
<th>Reduction Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\text{try } (f_1 (f_1 (f_1 0)))) (f_1 0))</td>
<td>(E[(f V)] \mapsto E[[V / x] M] ) if ((\text{define } (f x) M) \in \Gamma)</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>2</td>
<td>(\text{try } (f_1 (f_1 (\text{if}_0 0 1 \text{ (throw)}))) (f_1 0))</td>
<td>(E[(\text{if}_0 0 M_1 M_2)] \mapsto E[M_1])</td>
<td>Evaluation of (\text{if}_0) conditional, where the test is 0.</td>
</tr>
<tr>
<td>3</td>
<td>(\text{try } (f_1 (f_1 1))) (f_1 0))</td>
<td>(E[(f V)] \mapsto E[[V / x] M])</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>4</td>
<td>(\text{try } (f_1 (\text{if}_0 1 1 \text{ (throw)}))) (f_1 0))</td>
<td>(E[(\text{if}_0 1 M_1 M_2)] \mapsto E[M_2])</td>
<td>Evaluation of (\text{if}_0) conditional, where the test is 1.</td>
</tr>
<tr>
<td>5</td>
<td>(\text{try } (f_1 \text{ (throw)))) (f_1 0))</td>
<td>(E[(f \text{ (throw))}] \mapsto E[(\text{throw})]) )</td>
<td>Single-step reduction of ((\text{throw})) occurring as a parameter to a function.</td>
</tr>
<tr>
<td>6</td>
<td>(\text{try } \text{ (throw)} (f_1 0))</td>
<td>(E[(\text{try } \text{ (throw)} M)] \mapsto E[M])</td>
<td>Evaluation of (\text{try}) expression catching a ((\text{throw})).</td>
</tr>
<tr>
<td>7</td>
<td>(f_1 0)</td>
<td>(E[(f V)] \mapsto E[[V / x] M])</td>
<td>Call to a user-defined function.</td>
</tr>
<tr>
<td>8</td>
<td>(\text{if}_0 0 1 \text{ (throw)})</td>
<td>(E[(\text{if}_0 0 M_1 M_2)] \mapsto E[M_1])</td>
<td>Evaluation of (\text{if}_0) conditional, where the test is 0.</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: The steps in evaluating a program \(P\).
The left column shows the actual program in various stages of reduction. The subexpression that will be placed in the hole (and operated on by the reduction rules) is boxed. The middle column shows the reduction rule, as written in the operational semantics in Figure 2.3, that applies to the program at this stage. The right column describes the reduction.
(1) $X[\Gamma M] = X[\Gamma] X[M]$
(2) $X[D_1 \ldots D_n] = X[D_1] \ldots X[D_n]$
(3) $X[(\text{define } (f x) M)] = (\text{define } (f x) X[M])$
(4) $X[(f M)] = (\text{let } (t X[M]) ((\text{if}_0 \ (\text{CheckTermination})
\quad (f t \ (\text{throw})))))$

Where the identifier $t$ occurs nowhere else in the program.

(5) $X[(\text{if}_0 M_1 M_2 M_3)] = (\text{if}_0 X[M_1] X[M_2] X[M_3])$
(6) $X[(\text{try } M_1 M_2)] = (\text{try } X[M_1] X[M_2])$
(7) $X[(\text{let } (x M_1) M_2)] = (\text{let } (x X[M_1]) X[M_2])$
(8) $X[(\text{throw})] = (\text{throw})$
(9) $X[(\text{CheckTermination})] = (\text{CheckTermination})$
(10) $X[V] = V$
(11) $X[x] = x$

Figure 2.6: The soft termination transformation.

to that program, and the result is another syntactically valid program. A formal proof of this property is beyond the scope of this thesis.

Figure 2.5 describes a step-by-step evaluation of the sample program $P$. This program illustrates both the evaluation of functions, including the handling of function parameters, and the propagation and catching of exceptions.

Steps 1, 3, and 7 show the application of a user-defined function. Note in Steps 1 and 2, the parameter of the second application of function $f_1$ is reduced to a value before the application is reduced. Step 5 shows a $(\text{throw})$ expression single-stepping out of a function call. Step 6 shows the same $(\text{throw})$ expression being caught in a $(\text{try})$ expression. In Step 9, there are no more reduction rules to be applied, so the program returns the value 1.

Proof of Result Preservation

The soft termination transformation $X$ is described in Figure 2.6. Rule 4 of this transformation inserts the check for termination before every function application. It also describes how a function's parameter is first evaluated separately, and is likewise recursively transformed by $X$ to catch any nested function applications. The other transformation rules describe how the transformation continues recursively on expressions. We first prove that
if termination has not been requested, the transformed program evaluates to the same final state as the original untransformed program.

**Theorem 2.1.** Assume the definitions in Figures 2.1 through 2.4 and 2.6. As long as 
(CheckTermination) is 0, evaluating any transformed expression \( \text{X}[M] \) always results in the same final state as evaluating the corresponding expression \( M \).

**Proof.** We can show this by a structural induction on \( M \), the grammar of expressions in our language. In the base cases,

\[
M_{\text{base}} = (\text{throw}) | (\text{CheckTermination}) | V | x .
\]

In each of these cases, \( \text{X}[M_{\text{base}}] = M_{\text{base}} \), so the behavior of the expression is preserved.

In the inductive cases,

\[
M_{\text{inductive}} = (\text{if}_0 M M M) | (\text{let} \ (x \ M) \ M) | (\text{try} \ M M) | (f \ M),
\]

where

\[
M = M_{\text{base}} | M_{\text{inductive}} .
\]

In the first case, if\(_0 \) expressions are transformed by the rule:

\[
\text{X}[(\text{if}_0 M_1 M_2 M_3)] = (\text{if}_0 \ \text{X}[M_1] \ \text{X}[M_2] \ \text{X}[M_3]).
\]

By the inductive hypothesis, evaluating each \( M_i \) has the same final state as evaluating the corresponding \( \text{X}[M_i] \) as long as (CheckTerminion) is 0. Because the result of evaluating the if\(_0 \) expression is solely dependent on these results, it is also unaffected by \( \text{X} \). The cases of let and try expressions proceed similarly.

The final case, that of function applications, is not so straightforward. According to the definition of \( \text{X} \):

\[
\text{X}[f \ M] = (\text{let} \ (t \ \text{X}[M]) \ ((\text{if}_0 \ \text{CheckTermination} \ f t) \ (\text{throw}))).
\]

By the inductive hypothesis again, evaluating \( M \) has the same final state as evaluating \( \text{X}[M] \). Any error or exception in \( M \) occurs in \( \text{X}[M] \), causing the same final state. In the absence of errors or exceptions thrown, \( M \) and \( \text{X}[M] \) evaluate to the same value \( V \). At
this point, the original expression has reduced to \((f \ V)\), while the transformed expression has reduced to:

\[
(let \ (t \ V) \ ((if_0 \ (CheckTermination) \ (f \ t) \ (throw))))
\]

which further reduces to:

\[
(if_0 \ (CheckTermination) \ (f \ V) \ (throw)).
\]

Since \((CheckTermination)\) is 0, this reduces to \((f \ V)\), the same as the original expression.

Because, for every expression, the \(\mathcal{X}\) transformation does not affect the result of evaluating the expression as long as \((CheckTermination)\) is 0, the behavior of the entire program is preserved. □

**Proof of Termination**

Proving that a transformed program terminates in a finite number of reduction steps, given that \((CheckTermination)\) is 1, is a straightforward exercise. We start with the grammar \(M\) of all possible expressions from Figure 2.1. After transforming \(M\) by \(\mathcal{X}\), we call the resulting set of expressions \(M_{lock}\); \(M_{lock}\) is a subset of \(M\). Assuming \((CheckTermination)\) is 1, we wish to show the program enters a “locked” state where termination is guaranteed.

\[
M_{lock} = (if_0 \ (CheckTermination) \ (f \ V) \ (throw)) \mid \begin{array}{ll}
(if_0 \ 1 \ (f \ V) \ (throw)) & (a) \\
(CheckTermination) & (b) \\
(if_0 \ M_{lock} \ M_{lock} \ M_{lock}) & (c) \\
(let \ (x \ M_{lock} \ M_{lock}) \ M_{lock}) & (d) \\
(try \ M_{lock} \ M_{lock}) & (e) \\
(throw) \mid \begin{array}{ll}
x & (f) \\
V & (g) \\
\end{array}
\end{array}
\]

Expressions \(c\) to \(i\) are taken directly from the definition of \(M\) in Figure 2.1. Expression \(a\) represents the final case of \(M\), function applications, as transformed by \(\mathcal{X}\). Expression \(b\) describes an intermediate step in reducing expression \(a\) when \((CheckTermination)\) is 1. Note in particular, that \(\mathcal{X}[[M]] \subseteq M_{lock}\). To prove termination, we must show that, as long as \((CheckTermination)\) is 1, then \(M_{lock}\) is closed under program stepping, and that the syntactic length of the program will be strictly decreasing.
Lemma 2.1 (The Closure Property). Suppose (CheckTermination) is 1. For arbitrary expressions $M$ and $M'$, if $M \rightarrow M'$ and $M \in M_{lock}$, then $M' \in M_{lock}$.

Proof. By inspection, for all possible expressions in $M_{lock}$, we observe that our semantics preserves closure. □

Lemma 2.2 (The Syntactic Length Property). If (CheckTermination) is always 1, then for $M, M' \in M_{lock}$ such that $M \rightarrow M'$, we have that $|M'| < |M|$.

Proof. That is, as reduction rules are applied, a program in $M_{lock}$ gets shorter. To prove this, it suffices to observe that all reductions other than function application make the program smaller, and that for any expression in $M_{lock}$, all function applications are “fenced out” by a termination check. □

Lemma 2.3. When (CheckTermination) becomes set, a program being evaluated enters $M_{lock}$ within at most two reduction steps.

Proof. When a program is in $M_{lock}$ and (CheckTermination) is set, by Lemma 2.1, the program is guaranteed to stay in $M_{lock}$. If the program is not in $M_{lock}$ when (CheckTermination) is set, then it is either evaluating an application or a conditional which will result in an application (since these are the only two expressions where $M$ and $M_{lock}$ differ). Within one or two steps, then, it will be evaluating the body of the application, $\mathcal{X}[M]$. As noted above, this is defined to be in $M_{lock}$. In other words, a transformed program that is not in $M_{lock}$ is guaranteed to enter $M_{lock}$ within two steps. □

Theorem 2.2. Any program in $M$, after being transformed by $\mathcal{X}$, is guaranteed to terminate in a finite number of steps, once the (CheckTermination) condition is raised.

Proof. By Lemma 2.3, within two steps of the (CheckTermination) condition being raised, the program is guaranteed to be in $M_{lock}$. By Lemma 2.1, the program is guaranteed to remain in $M_{lock}$. By Lemma 2.2, then, the program is guaranteed to get shorter with each reduction. Since program lengths are finite, a program transformed by $\mathcal{X}$ is guaranteed to reduce to some non-reducible expression (that is, to a single value or (throw)) in a finite number of steps, once the (CheckTermination) condition is raised. □
\[ P = \Gamma M \]
\[ \Gamma = D \ldots D \]
\[ L = \text{system} | \text{codelet} \]
\[ D = (\text{define } L (f \ x) M) | (\text{define blocking } (f_{\text{blocking}} \ x) f_{\text{nonblocking}}) | (\text{define primitive } (f \ x)) \]
\[ M = (f \ M) | (\text{if}_0 \ M \ M \ M) | (\text{let} (x \ M) \ M) | (\text{try} \ M \ M) | (\text{throw}) | V | x | (\text{CheckTermination}) \]
\[ V = c \in \mathbb{N} | \text{blocks} \]
\[ f, x = \text{identifiers} \]

Figure 2.7: An extended language for analysis.
This extended language distinguishes codelets from system code, and identifies blocking functions.

### 2.1.4 Soft Termination with Codelet, System, and Blocking Code

Although this little language provides a significant subset of most programming languages, it likewise lacks many interesting features. Two in particular are blocking functions and a distinction between system and user code. Both of these features require extensions to the language. Figures 2.7 and 2.9 introduce the syntax and semantics of the extended language, and Figure 2.8 shows the evaluation context. Section 2.2.3 discusses our handling of blocking calls in our Java implementation of soft termination.

We first discuss the extensions made to this language. We then examine the soft termination transformation as applied to the extended language. We show that, once termination has been requested, a program in this language is guaranteed to terminate.

**Language Extensions**

The distinction between system and user functions is especially relevant as a feature of mobile code; it represents a distinction between trusted and untrusted code. This trust relationship is enforced by restricting the providers of untrusted codelets to declaring functions **codelet**. Only functions that are defined as part of the run-time system can be declared **system**; we assume that some preprocessor exists to remove any **system** declarations from user code. We use the term "system code" to refer to the collection of expressions declared
\[
E = [1 | (f \ E) | (if_0 \ E \ M \ M) | (let \ (x \ E) \ M) | (try \ E \ M) |
\text{(CheckTermination)} | \text{(throw)} | x
\]

\[
\text{final states} = V | error | \text{(throw)}
\]

Figure 2.8 : Evaluation context for reduction of this extended language.

\[
E[(f \ V)] \mapsto \begin{cases} 
E[V_0] & \text{if (define primitive } (f \ x) \text{) } \in \Gamma \\
\text{and } \delta(f, V) = V_0 \\
E[[V / x] \ M] & \text{if (define } L (f \ x) \ M) \in \Gamma \\
E[V_1] & \text{(that is, } f \text{ is declared blocking)} \\
\text{and } \delta'(f, V) = V_1 \\
E[V_2] & \text{if (define blocking } (f \ x) \ g) \in \Gamma \\
\text{and } \delta'(f, V) \text{ is undefined} \\
E[blocks] & \text{if (define blocking } (g \ x) \ f) \in \Gamma \\
\text{and } \delta'(g, V) \text{ is undefined} \\
error & \text{otherwise}
\end{cases}
\]

The value of \(\delta'(f, V)\) and whether it is defined depend on external conditions.

\[
E[(\text{CheckTermination})] \mapsto E[1] \text{ or } E[0] \text{ (depending on external conditions)}
\]

\[
E[[\text{if}_0 \ 0 \ M_1 \ M_2]] \mapsto E[M_1]
\]

\[
E[[\text{if}_0 \ blocks \ M_1 \ M_2]] \mapsto error \text{ if } V \neq 0, blocks
\]

\[
E[(\text{let } (x \ V) \ M)] \mapsto E[[V / x] \ M]
\]

\[
E[(\text{try } V \ M)] \mapsto E[V]
\]

\[
E[(\text{try } (\text{throw}) \ M)] \mapsto E[M]
\]

\[
E[(f \ (\text{throw}))] \mapsto E[(\text{throw})]
\]

\[
E[[\text{if}_0 \ (\text{throw}) \ M_1 \ M_2]] \mapsto E[(\text{throw})]
\]

\[
E[(\text{let } (x \ (\text{throw})) \ M)] \mapsto E[(\text{throw})]
\]

\[
E[x] \mapsto error \text{ (unbound variables)}
\]

\[
\delta(\text{blocks}?, v) = \begin{cases} 
1 & \text{if } v = \text{blocks} \\
0 & \text{otherwise}
\end{cases}
\]

Figure 2.9 : Operational semantics for the extended language.

The extended language syntax is given in in Figure 2.7. This is largely a superset of the semantics of the original language, described in Figure 2.3.

\text{\dagger}As in the original language, the expansion \( [V / x] \ M \) indicates that every instance of \( x \) in \( M \) should be replaced with the corresponding \( V \), according to the standard rules of lexical scope. These are defined in Figure 2.4.
Figure 2.10: The soft termination transformation for this extended language.
in the bodies of all functions declared `system`, and the term "codelet" to refer to all other expressions in a program.

Because of the added number of other function types, all of which, including those behaving like primitive operations, are declared explicitly, we have also chosen to declare primitive operations explicitly using the `primitive` keyword. This is similar to how Java declares native methods.

The extended syntax greatly complicates how function applications are evaluated. If a function being applied has been declared `primitive`, the application reduces to the value of the $\delta$ function applied to that operation. If the function is declared `system` or `codelet`, the application reduces to the body of the function. Evaluation of `blocking` function applications is described below. If the function is not declared in the program, the application reduces to `error`.

**Blocking Functions.** Blocking functions are those functions that are known to block. That is, if a return value is not available, such functions may stall indefinitely waiting for the value to become available. Blocking functions generally provide some service of the underlying operating system, such as I/O operations. They are generally called through some interface provided by the operating system, and treated in a fashion similar to primitives in the language. As a result, we treat their definition and evaluation in a manner similar to how we treat primitive operations.

In particular, we define a function $\delta'$, similar to the $\delta$ function used to define primitives. $\delta'$ is a nondeterministic function indicating, based on external conditions, whether the application is defined and to what value the corresponding blocking function application should reduce. If the $\delta'$ function is undefined, this indicates that a return value is not available for the blocking function.

The `blocking` function declaration must name one additional function, which we call $f_{\text{nonblocking}}$. This function is a nonblocking version of the blocking function, which we call $f_{\text{blocking}}$. That is, when $\delta'$ is defined, applications of these two functions reduce to the same value. The behavior of these functions only differs when $\delta'$ is undefined. In this case, the expression ($f_{\text{nonblocking}} V$) reduces to the value `blocks`, while the expression ($f_{\text{blocking}} V$)
reduces to itself, and will continue to reduce to itself as long as \( \delta' \) is undefined. The third and fourth reduction rules for function applications define the behavior for applications of \( f_{\text{blocking}} \); the fifth and sixth rules define the behavior for applications of \( f_{\text{nonblocking}} \).

**Description of \( \mathcal{X}_2 \) Transformation**

Because we have extended this language, we must now extend the soft termination transformation. We call this new transformation, described in Figure 2.10, \( \mathcal{X}_2 \), to distinguish it from the \( \mathcal{X} \) transformation in Figure 2.6.

Rules 3a through 3d describe the transformation on function definitions. Rules 3a and 3b describe how the transformation continues recursively on system and codelet function definitions, and Rule 3d says primitive function declarations are unmodified. Rule 3c says a wrapper function is created for every blocking function. The wrapper uses the non-blocking function declared with each blocking function to simulate the effect of applying the blocking function.

This wrapper alternately polls the nonblocking function and checks to see if termination has been indicated. When the nonblocking function application reduces to a value other than blocks, the wrapper function application reduces to this value. If termination is ever indicated, the wrapper throws an exception. Otherwise, the wrapper function is recursively applied, and the process repeats. We use the identifier \( f_{\text{wrapper}} \) to refer to such a wrapper function generated for some blocking function \( f_{\text{blocking}} \). Note that different programming languages choose different abstractions for managing blocking I/O primitives. In Section 2.2.3, we show how this works in Java.

Rule 4 describes how function applications are handled for \( \mathcal{X}_2 \). If the function being applied is a codelet function, the termination check is added at the call site to the function, just as in the \( \mathcal{X} \) transformation. If it is a system function, however, no termination checks are added. Adding termination checks on applications of system functions may cause an unexpected (throw) to be evaluated in a critical section of a system function, as described in Section 2.1.1. Recall from Section 2.1.2 that UNIX treats calls to system code similarly.

Note that where the original system function made an up-call to a codelet, the \( \mathcal{X}_2 \) transformation may cause an exception to be thrown at the call site. System code is responsible
for catching this exception and proceeding appropriately. Because exceptions are already a valid result of applying \texttt{codelet} functions, system code must already be prepared to handle this case. As a result, this adds no new constraints on system code.

Rule 5 describes how applications of blocking functions are replaced with applications of the corresponding nonblocking wrapper function. This guarantees that no blocking function is ever applied in the transformed program, simplifying the termination proof.\footnote{Even in \texttt{system} functions, blocking function applications are replaced with applications of the corresponding nonblocking wrapper. This can cause an unexpected \texttt{throw} to be evaluated in system code; to prevent this in our Java implementation of soft termination, blocking method calls from system code are not wrapped, as described in Section 2.2.3. Since the termination result we prove in Section 2.1.4 includes the assumption that blocking function applications return in finite time, this result is still valid.} The remainder of the rules describe how the transformation continues recursively on expressions.

\textbf{Safety Property of System Code}

For this extended language, we wish to prove that a program transformed by $\mathcal{X}_2$ terminates in a finite number of steps, given that \texttt{(CheckTermination)} is 1. This is complicated by the fact that \texttt{system} functions are not guaranteed to terminate, even after the program has been transformed by $\mathcal{X}_2$. As a result, we must assume some safety property of the system code in order to prove termination.

The intuition behind this safety property is that if the program diverges or reaches a fixed point, it is not the fault of system code. A program diverges when, as reduction rules are applied, the length of the program grows without bound. A program reaches a fixed point when, as reduction rules are applied, the result is the same program. The property is stated as follows:

\textbf{Definition 2.1 (Safety Property of System Code).}

Suppose for all declarations

\[
(\texttt{define system } (f_{system} \ x) \ M) \in \Gamma,
\]
for every application \((f_{\text{codelet}}~V_0)\) and \((f_{\text{blocking}}~V_1)\) in \(M\), where

\[
(\text{define codelet } f_{\text{codelet}} x) M_2 \in \Gamma \text{ and } (\text{define blocking } f_{\text{blocking}} x) f_{\text{nonblocking}} \in \Gamma,
\]

there exists an integer \(c > 0\) such that

\[
E[(f_{\text{codelet}} V_0)] \rightarrow^c E[V'_0] \quad \text{and} \quad E[(f_{\text{blocking}} V_1)] \rightarrow^c E[V'_1] \quad \text{at} \quad E[(\text{throw})]
\]

for values \(V'_0, V'_1\). Then there is an integer \(c_2 > 0\) such that

\[
E[(f_{\text{system}} V)] \rightarrow^{c_2} E[V'] \quad \text{at} \quad E[(\text{throw})]
\]

for every \textit{system} function \(f_{\text{system}}\), for some value \(V'\).

The safety property states that, if every \textit{codelet} and \textit{blocking} function application in any \textit{system} function reduces to a value or to \textit{(throw)} in a finite number of steps, then every \textit{system} function reduces to a value or to \textit{(throw)} in a finite number of steps. This property applies to system functions in the original program, before transformation by \(\mathcal{X}_2\). It does not come automatically, but we assume the authors of \textit{system} functions guarantee it for their code. Recall from Section 2.1.4 that untrusted functions are never declared \textit{system}.

By this property, system code will always terminate unless it applies a \textit{codelet} or \textit{blocking} function that either diverges or reaches a fixed point. As a result, system code acting alone will always return in finite time, and is never at fault if a \textit{system} function fails to return.

**Proof of Termination**

In order to prove termination, we start with the grammar \(M\) of all possible expressions from Figure 2.7. After transforming \(M\) by \(\mathcal{X}_2\), we call the resulting set of expressions \(M_{\text{lock2}}\) to distinguish from \(M_{\text{lock}}\) defined in Section 2.1.3; like \(M_{\text{lock}}\), \(M_{\text{lock2}}\) is a subset of
Assuming (CheckTermination) is 1, we wish to show the program enters a “locked” state, where termination is guaranteed.

\[ M_{\text{lock}_2} = \begin{cases} \text{(if}_0 \ (\text{CheckTermination}) \\ (f \ V) \ (\text{throw}) ) \mid \\ (\text{if}_0 \ 1 \ (f \ V) \ (\text{throw}) ) \mid \\ (f_{\text{nonblocking}} \ V) \mid \end{cases} \quad \begin{cases} \text{if (define codelet } (f \ x) M \in \Gamma) \quad (a) \\ \text{if (define codelet } (f \ x) M \in \Gamma) \quad (b) \\ \text{if (define blocking } (f_{\text{blocking}} x) \quad (c) \\ f_{\text{nonblocking}}) \in \Gamma \quad \end{cases} \\
(\text{define primitive } (f \ x)) \in \Gamma) \quad \begin{cases} (d) \\ \text{(define system } (f \ x) M \in \Gamma) \quad (e) \\ \text{if (CheckTermination) } \begin{cases} \text{if (if}_0 \ M_{\text{lock}_2} M_{\text{lock}_2} M_{\text{lock}_2}) \quad (g) \\ \text{(let } (x M_{\text{lock}_2} M_{\text{lock}_2}) \quad (h) \\ \text{(try } M_{\text{lock}_2} M_{\text{lock}_2}) \quad (i) \\ (\text{throw}) \mid \quad (j) \\
 x \mid \quad (k) \\ V \quad (l) \end{cases} 
\end{cases}

Note in particular, that \( X_2[M] \subseteq M_{\text{lock}_2} \). To prove termination, we must show that, as long as (CheckTermination) is 1, \( M_{\text{lock}_2} \) is closed under program stepping and that the syntactic length of the program is decreasing. Closure is defined in detail in Section 2.1.3.

**Lemma 2.4 (The Closure Property).** Suppose (CheckTermination) is 1. For arbitrary expressions \( M \) and \( M' \), if \( M \rightarrow M' \) and \( M \in M_{\text{lock}_2} \), then \( M' \in M_{\text{lock}_2} \).

**Proof.** By inspection, for all possible expressions in \( M_{\text{lock}_2} \), we observe that our semantics preserve closure. \( \square \)

**Lemma 2.5 (The Syntactic Length Property).** If (CheckTermination) is always 1, then for any \( M \in M_{\text{lock}_2} \), there is some integer \( c > 0 \) and \( M' \in M_{\text{lock}_2} \) such that \( M \rightarrow^c M' \) and \( |M'| < |M| \).

**Proof.** That is, for any program in \( M_{\text{lock}_2} \), after \( c \) reduction steps, it will have gotten smaller. Since programs are finite length, a program with these properties will terminate in a finite number of steps.

Based on the operational semantics, the reductions of expressions \( f \) through \( l \) all make a program strictly smaller. Likewise, applications of primitive operations and nonblocking functions (which is to say, expressions \( c \) and \( d \)) reduce to values in one step. Note that applications of blocking functions may never reduce to a value; however, the \( X_2 \) transformation removes all such applications from the program.
Expressions $a$ and $b$ are applications of codelet functions fenced by (CheckTermination) checks. Since (CheckTermination) is 1, these expressions reduce to (throw). The final case is expression $c$, applications of system functions. This expression must be considered in two cases. The first case is if the function being applied is a blocking wrapper function, $f_{\text{wrapper}}$ (i.e., functions created by the $\mathcal{X}_2$ transformation applied to a blocking function declaration). The resulting expression is in $M_{\text{block}}$, and all applications of system functions from within $f_{\text{wrapper}}$ are fenced by a termination check. As a result, as long as (CheckTermination) is 1, applications of $f_{\text{wrapper}}$ reduce to a value in finitely many steps.

To address the second case, of any other functions declared system being applied, we recall the safety property in Section 2.1.4. This states that if, at every point where any system function applies a codelet or blocking function, that application reduces in finitely many steps, then the application of every system function reduces in finitely many steps.

By the $\mathcal{X}_2$ transformation, wherever an application of the original system function reduces ($f_{\text{codelet}} V$), for some codelet function $f_{\text{codelet}}$, the transformed function reduces (if$_0$(CheckTermination) ($f_{\text{codelet}} V$) (throw)). Wherever the original system function reduces ($f_{\text{blocking}} V$) for some blocking function $f_{\text{blocking}}$, the transformed function reduces ($f_{\text{wrapper}} V$), where $f_{\text{wrapper}}$ is the wrapper function that corresponds to $f_{\text{blocking}}$.

We have already shown that, as long as (CheckTermination) is 1, for every wrapper function $f_{\text{wrapper}}$, for some $c > 0$,

$$E[(f_{\text{wrapper}} V)] \rightarrow^c E[V_0] | E[(\text{throw})]$$

for some value $V_0$. Likewise we can see that, as long as (CheckTermination) is 1,

$$E[(\text{if}_0 \ (\text{CheckTermination}) \ldots \ (\text{throw}))] \rightarrow^c E[(\text{throw})].$$

As a result, wherever the original system function applied either a codelet or a blocking function, the expression to which $\mathcal{X}_2$ transforms this application reduces in a finite number of steps. By applying the safety property, we can conclude that an application of any system function reduces to a value in a finite number of steps, and all expressions in $e$ have the syntactic length property.

$\square$
Lemma 2.6. When \((\text{CheckTermination})\) becomes set, a program being evaluated enters \(M_{\text{lock}_2}\) within at most two reduction steps.

Proof. When a program is in \(M_{\text{lock}_2}\) and \((\text{CheckTermination})\) is set, by Lemma 2.4, the program is guaranteed to stay in \(M_{\text{lock}_2}\). If the program is not in \(M_{\text{lock}_2}\) when \((\text{CheckTermination})\) is set, then it is either evaluating an application of a \(\text{codelet}\) function or a conditional which will result in such an application. Within one or two steps, then, it will be evaluating the body of the application, \(X_2[M]\). As noted above, this is defined to be in \(M_{\text{lock}_2}\). In other words, a transformed program that is not in \(M_{\text{lock}_2}\) is guaranteed to enter \(M_{\text{lock}_2}\) within two steps. \(\square\)

Theorem 2.3. Any program in \(M\), after being transformed by \(X_2\), is guaranteed to terminate in a finite number of steps, once the \((\text{CheckTermination})\) condition is raised.

Proof. By Lemma 2.6, within two steps of the \((\text{CheckTermination})\) condition being raised, the program is guaranteed to be in \(M_{\text{lock}_2}\). By Lemma 2.4, the program is guaranteed to remain in \(M_{\text{lock}_2}\). By Lemma 2.5, then, the program reduces to some smaller program in a finite number of steps. Since program lengths are finite, the program is guaranteed to reduce to some non-reducible expression (a value or \((\text{throw})\)) in a finite number of steps, once the \((\text{CheckTermination})\) condition is raised, given the safety constraint on system code stated in section 2.1.4. \(\square\)

2.2 Java Implementation

In an effort to understand the practical issues involved with soft termination, we implemented it for Java as a transformation on Java bytecodes. Our implementation relies on a number of Java-specific features. We also address a number of Java-specific quirks that we would not expect to exist in other language systems. Examples showing how the transformations as implemented alter Java source language are shown in Figures 2.11 through 2.13. Although the transformations actually operate on Java bytecode, there is a direct mapping from the targeted bytecodes to the corresponding Java source representation, and using Java source language is clearer.
void foo() {
    if (termination_signal) {
        termination_handler();
    }
    ...
}

Figure 2.11: The soft termination transformation applied to any function definition. See Section 2.2.1 for a description of this transformation. This example takes into account the optimizations discussed in Section 2.2.5.

2.2.1 Termination Check Insertion

Java compilers normally output Java bytecode. Every Java source file is translated to one or more class files, later loaded dynamically by the JVM as the classes are referenced by a running program. JVMs know how to load class files directly from the disk or indirectly through “class loaders,” invoked as part of Java’s dynamic linking mechanism. A class loader, among other things, embodies Java’s notion of a name space. Every class is tagged with the class loader that installed it, such that a class with unresolved references is linked against other classes from the same source. A class loader provides an ideal location to rewrite Java bytecode, implementing the soft termination transformation. A codelet appears in Java as a set of classes loaded by the same class loader. System code is naturally loaded by a different class loader than codelets, allowing us to simplify the implementation by applying different transformations to codelets and system code. Our implementation uses the CFParse\(^3\) and JOIE [18] packages, which provide interfaces for parsing and manipulating Java class files.

The basic structure of our bytecode modification is exactly as described in Section 2.1.3. A static Boolean field, initially set to false, is added to every Java class. The CheckTermination operation, implemented inline, tests if this field is true, and if so, calls a handler method that decides whether to throw an exception. As an extension to the semantics of Figure 2.10, we allow threads and thread groups to be terminated as well as specific

\(^3\)http://www.alphaworks.ibm.com/tech/cfparse.
void foo() {
    if (termination_signal) {
        termination_handler();
    }
    ...
    while (...) {
        ...
        if (termination_signal) {
            termination_handler();
        }
    }
    ...
}

Figure 2.12: The soft termination transformation applied to any loop. See Section 2.2.2 for a description of this transformation.

codelets, regardless of the running thread. The termination handler, when invoked, looks its caller and current thread up in a list of known termination targets. Note that, if the Boolean field is set to false, the run-time overhead is only the cost of loading and checking the value, and then branching forward to the remainder of the method body. Figure 2.11 shows how the soft termination transform would be applied to a Java method declaration.

2.2.2 Control Flow

Java has a much richer control flow than the little language introduced earlier. First and foremost, Java bytecode has a general-purpose branch instruction. We do nothing special for forward branches, but we treat backward branches as if they were method invocations and perform the appropriate code transformation. An additional special case we must handle is a branch instruction that targets itself. The effect of transforming a method with loops is shown in Figure 2.12.

Java bytecode also supports many constructions that have no equivalent Java source code representation. In particular, it is possible to arrange for the catch portion of an exception handler to be equal to the try portion. That means an exception handler can be defined to handle its own exceptions. Such a construction allows for infinite loops
void foo() {
    ... 
    blocking_bar(...);
    ... 
}

void foo() {
    if (termination_signal) {
        termination_handler();
    }
    ... 
    blocking_wrapper_bar(...);
    ...
}

Bar blocking_wrapper_bar(...) {
    register_blocking(
        Thread.currentThread());
    // Uses stack inspection
    Bar tmp = blocking_bar();
    unregister_blocking(
        Thread.currentThread());
    return tmp;
}

Figure 2.13: The soft termination transformation applied to a blocking call.
This figure includes an outline for the definition of the blocking call wrapper function.
The signature of function blocking_wrapper_bar() is the same as that for function
blocking_bar(). See Section 2.2.3 for a description of this transformation.

without any method invocation or explicit backward branching. Although such code is not
allowed according to the JVM specification [53], the bytecode verifier we used treats such
constructions as valid. We specifically check for and reject programs with overlapping try
and catch blocks.

Lastly, Java bytecode supports a notion of subroutines within a Java method using the
jsr and ret instructions. jsr pushes a return address on the stack, and ret consumes
this address before returning. The Java bytecode verifier imposes a number of restrictions
on how these instructions may be used. In particular, a return address is an opaque type
that may be consumed at most once. The verifier’s intent is to ensure that these instruc-
tions may be used only to create subroutines, not general-purpose branching. As such, we
instrument jsr instructions the same way we would instrument a method invocation and
we do nothing for ret instructions.
2.2.3 Blocking Calls

To address blocking calls, we wish to follow the $\mathcal{X}_2$ transformation outlined in Section 2.1.4. Luckily, all blocking method calls in the Java system libraries are native methods (implemented in C) and can be easily enumerated and studied by examining the source code of the Java class libraries.

Using a polling model for terminating blocking calls simplifies analysis, but it is not a very practical implementation. This is because of the processing time required for polling. However, Java provides a mechanism for interrupting blocking calls, Thread.interrupt(). If a thread has called a blocking function and is blocking, this method, when called on the thread, causes the blocking method to throw a java.lang.InterruptedIOException or java.io.InterruptedIOException exception.

As in the $\mathcal{X}_2$ transformation described in Section 2.1.4, we still wrap blocking function calls; and like $\mathcal{X}_2$, the wrapper returns if either a value is returned or termination is signaled. However, instead of polling a nonblocking function, the wrapper uses the interruption support already inside the JVM. When termination is requested for a codelet, if a corresponding thread is in a blocking call, that thread is interrupted with Thread.interrupt().

To accomplish this, we must track which threads are currently blocking and the codelets on behalf of which they are blocking. The wrapper functions now get the current thread and save it in a global table for later reference. In order to learn the codelet on whose behalf we are about to block, we take advantage of the stack inspection primitives built into modern Java systems [80, 41].

Stack inspection provides two primitives that we use: java.security.AccessController.doPrivileged() and getContext().getContext() returns an array of ProtectionDomains that map one-to-one with codelets. The ProtectionDomain identities are then saved alongside the current thread before the blocking call is performed. When we wish to terminate a codelet, we look up whether it is currently in a blocking function call, and if so, we interrupt the corresponding thread.

Taking advantage of another property of Java stack inspection, we can distinguish between blocking calls being performed on behalf of system code and those being performed
indirectly by a codelet. We do not want to interrupt a blocking call if system code is depending on its result and system state could become corrupted if the call were interrupted. On the other hand, we have no problem interrupting a blocking call if only a codelet is depending on its result. Java system code already uses **getApplicationContext()** to get dynamic traces for making access control checks. The **doPrivileged()** mechanism is likewise already used in the Java API library to identify code to run with system privileges. This is used for performing security-sensitive operations that codelets themselves should not be authorized to perform. Thus we overload the semantics of these existing security primitives to include whether blocking calls should be interrupted.

In principle, any code where preserving system integrity upon termination is important should already be wrapped with **doPrivileged()**, and thereby executed with system privileges. However, verifying this would require an audit of the Java API library. The effect of this transformation on Java source is shown in Figure 2.13.

### 2.2.4 Invoking Termination

Our system supports three levels of granularity for termination: termination of individual threads, termination of thread groups, and termination of codelets. To terminate a thread or thread group, we must map the threads we wish to terminate to the set of codelets potentially running those threads and set the termination signal on all classes belonging to the target codelet. Furthermore, we must check if any of these threads are currently blocking and interrupt them (see Section 2.2.3). At this point, the thread requesting termination performs a **Thread.join()** on the target threads, waiting until they complete execution. Once all target threads have completed, the termination signals are cleared and execution returns to normal.

If multiple threads are executing concurrently over the same set of classes and only one is terminated, the termination handler will be invoked for threads not targeted, only to return shortly thereafter. These threads will experience degraded performance while the target thread is still running.

In the case where we wish to terminate a specific codelet, disabling all its classes forever, we simply set the termination signal on all classes in the codelet and immediately
return. Any code that invokes a method on a disabled class will receive an exception indicating the class has been terminated.

Once a codelet has been signaled to terminate, if a codelet’s thread is executing in a system class at the time, execution continues until the thread returns to a user class. If the codelet is currently making a blocking call, the call is interrupted and the thread resumes execution. Once the thread has resumed executing in the user’s class, it becomes subject to the soft termination system.

For all codelet threads that are executing within the codelet, if they try to call a method within the codelet, the method fails with an exception. If they try to perform a backward branch, the soft termination code will throw an exception. In all cases, each thread of control unwinds, preventing the codelet from performing any meaningful work. Finally, if any other codelet or the system makes a call into this codelet, it will fail immediately, preventing the codelet from hijacking the caller thread for the codelet’s own use. As proven in Section 2.1.4, the codelet is guaranteed to terminate.

Note that termination requests can be handled concurrently. A potential for deadlock occurs when a thread requests its own termination, or when a cycle of threads request each others’ termination. When a user is manually terminating threads or codelets, this would not be an issue. However, care should be taken to prevent untrusted codelets from invoking the termination operations. For this reason, these operations are protected using the same security mechanisms as other Java privileged calls.

### 2.2.5 Optimizations

If a Java method contains a large number of method invocations, the transformed method may be significantly larger than the original, potentially causing performance problems. To address this concern, we observe that we get similar semantics by moving the soft termination check from the call sites to the entry points of methods. Every function for which we performed a termination check before calling now instead begins with a termination check, so the resulting program will behave the same. Figure 2.11 shows the effect of using this optimization in transforming a Java function.
In addition, we implemented an optimization to statically determine if a method has no outgoing method calls (i.e., is a leaf method). For leaf methods, a termination check at the beginning of the method is unnecessary. If the method has loops, they will have their own termination checks. If not, the method is guaranteed to complete in a finite time. Regardless, removing the initial termination check from leaf methods preserves the semantics of soft termination and should offer a significant performance improvement, particularly for short methods such as "getter" and "setter" methods.

A more aggressive optimization, which we have not yet performed, would be an interprocedural analysis of statically terminating methods. A method that only calls other terminating methods and has no backward branches will always terminate. Likewise, we have not attempted to distinguish loops that can be statically determined to terminate in a finite time (i.e., loops that can be completely unrolled). Such analyses could offer significant performance benefits to a production implementation of soft termination.

2.2.6 Synchronization

A particularly tricky aspect of supporting soft termination in a Java system is supporting Java’s synchronization primitives. The Java language and virtual machine specifications are not clear on how the system behaves when a deadlock is encountered [42, 53]. With Sun’s JDK 1.2, the only way to recover from a deadlock is to terminate the JVM. Obviously, this is an unsatisfactory solution. Ideally, we would like to see a modification to the JVM where locking primitives such as the monitorenter bytecode are interruptible, like other blocking calls in Java. We could then apply standard deadlock detection techniques and choose the threads to interrupt.

It is also possible to construct Java classes where the the monitorenter and monitorexit bytecodes, which acquire and release system locks, respectively, are not properly balanced. Despite the fact that there exist no equivalent Java source programs, current JVM bytecode verifiers accept such programs. This makes it possible for a malicious program to acquire a series of locks and terminate without those locks being released until the JVM terminates. Our current system makes no attempt to address these issues.
2.2.7 Thread Scheduling

Our work fundamentally assumes the Java thread system is preemptive. This was not the case in many early Java implementations. Without a preemptive scheduler, a malicious codelet could enter an infinite loop and no other thread would have the opportunity to run and request the termination of the malicious thread. We can work around such an attack by inserting calls to \texttt{Thread.yield()} at the cost of some additional slowdown.

2.2.8 System Code Safety

Our work also assumes that all system methods that may be invoked by a codelet will either return in a finite time or will reach a blocking native method call which can be interrupted. This property of system code is stated and justified in Section 2.1.4. It may be possible to construct an input to system code that will cause the system code itself to have an infinite loop. Addressing this concern would require a lengthy audit of the system code to guarantee there exist no possible inputs to system functions that may cause infinite loops.

2.2.9 Memory Consistency Models

The Java language defines a relaxed consistency model where updates need not be propagated in the absence of locking primitives. In our current prototype, we use no synchronization primitives when accessing the termination flag. Since external updates to the termination signal could potentially be ignored by the running method, this could defeat the soft termination system.

Instead, we take advantage of Java's \texttt{volatile} modifier. This modifier is provided to guarantee that changes must be propagated immediately [42]. On the benchmark platform we used, the performance impact of using \texttt{volatile} versus not using it is negligible. However, on other platforms, especially multiprocessing systems, this may not be the case.

2.2.10 Defensive Security

Our prototype implementation makes no attempt to protect itself from bytecode designed specifically to attack the termination system (e.g., setting the termination flag to \texttt{false}, ei-
ther directly or through Java’s reflection interface). Such protection could be added as a verification step before the bytecode rewriting.

2.3 Performance

We measured the performance of our soft termination system using Sun Microsystems Ultra 10 workstations (440 MHz UltraSPARC II CPUs with 128 MB memory, running Solaris 2.6), and Sun’s Java 2, version 1.2.1 build 4, which includes a just-in-time compiler (JIT). A JIT compiles the Java bytecodes to native machine code at run-time, eliminating the overhead of interpreting Java code. Our benchmarks were compiled with the corresponding version of javac with optimization turned on.

We used two classes of benchmark programs: microbenchmarks that test the impact of soft termination on various Java language constructs (and also measuring worst-case performance), and some real-world applications. We measured the performance of these systems in three configurations: their original unmodified state, their state after being rewritten, and their state after being rewritten using the leaf method optimization discussed in Section 2.2.5. Generally, when we discuss results in this section, we refer to the optimized numbers because these better reflect the performance of a production soft termination system.

2.3.1 Microbenchmarks

We first measured a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, method and field accesses, exception handling, synchronization, and I/O. We used a microbenchmark package developed at the University of California, San
Figure 2.14: Soft termination overhead for microbenchmarks. This graph illustrates the performance of rewritten microbenchmark class files relative to the performance of the corresponding original class files. The graph measures how much longer the benchmarked class took to run than the original class. The left bars show the overhead for the original transformed classes. The right bars show the overhead for the classes transformed using the leaf method optimization discussed in Section 2.2.5.

Diego, and modified at the University of Arizona for the Sumatra Project. The results are shown in Figure 2.14.

As one would expect, loops suffered the worst overheads, of between 100 and 170%. In the transformed version of the loop, the cost of the termination check is roughly the same as the cost of the looping construct itself, so it is sensible to see such a performance degradation. In addition, in some cases, the added termination checks might inhibit loop optimizations.

For other microbenchmarks, we saw much smaller overheads. The overhead of handling exceptions, performing synchronization, or doing I/O operations dominates the cost of checking for termination. The largest overhead of these was 13.7% for the synchronization microbenchmark. This overhead can be attributed to performing the termination

---

4The original Web site is http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html. The source we used was distributed from http://www.cs.arizona.edu/sumatra/ftp/benchmarks/Benchmark.java.
check once for each iteration of the benchmark. For the I/O and exception-handling microbenchmarks, the performance figures are much better. Since I/O and exception handling are relatively costly operations, modifications do not have as significant an impact on performance.

We observe that the leaf method optimization generally has some performance benefit. The loop method invocation microbenchmark shows the most dramatic improvement; the optimized benchmark has nearly half the overhead of the unoptimized benchmark. In one case, the exception handling benchmark, the optimized program ran roughly 1% slower than the unoptimized program. Similar behavior occurred in the Linpack application benchmark. The optimized programs are genuinely performing fewer termination checks, but still have longer run-times. The culprit appears to be Sun’s JIT compiler (sun-wjit). When the benchmarks are run with the JIT disabled, the optimized programs are strictly faster than the unoptimized programs. We have observed similar deviant behavior with Sun’s HotSpot JIT running on SPARC/Solaris and x86/Linux. We have sent an appropriate bug report to Sun.

2.3.2 Application Benchmarks

We benchmarked the real-world applications JavaCup,\(^5\) Linpack,\(^6\) Jess,\(^7\) and OTP.\(^8\) These programs were chosen to provide sufficiently broad insight into our system’s performance. JavaCup is a LALR parser-generator for Java, and Jess is an expert system shell. These programs were chosen to demonstrate how soft termination performed in tasks that are more dependent on I/O and computation than on iteration. Linpack is a loop-intensive floating-point benchmark. OTP is a one-time password generator which uses a cryptographic hash function. The results are shown in Figure 2.15.

For the JavaCup test, we generated a parser for the Java 1.1 grammar. When rewritten, it ran 6% slower. For the Jess test, we ran several of the sample problems included with

\(^6\)http://netlib2.cs.utk.edu/benchmark/linpackjava/.
\(^7\)http://herzberg.ca.sandia.gov/jess/.
\(^8\)http://www.cs.umd.edu/~harry/jotp/.
Figure 2.15: Soft termination overhead for application benchmarks. This graph illustrates the performance of rewritten application class files relative to the performance of the corresponding original class files. The graph measures how much longer the benchmarked class took to run than the original class. The left bars show the overhead for the original transformed classes. The right bars show the overhead for the classes transformed using the leaf method optimization discussed in Section 2.2.5.

Jess through the system, and calculated the cumulative run-times. This program, when rewritten, ran 3% slower. Both JavaCup and Jess represent applications that do not make extensive use of tight loops. Instead, these applications spend more of their time performing I/O and mathematical or symbolic computations. The performance results reflect this.

For the Java OTP generator, we generated a one-time password from a randomly chosen seed and password, using 200,000 iterations. There was a 18% increase in run-time. For the Linpack benchmark, there was a 25% increase in run-time. Linpack is a loop-intensive program, whereas OTP makes extensive use of method calls as well as loops. As a result, we would expect the overhead from soft termination checks to be somewhat higher than for the other two application benchmarks. Note in particular the benefit OTP got from the leaf method optimization.
<table>
<thead>
<tr>
<th>Microbenchmarks</th>
<th>Application Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Benchmark</strong></td>
<td><strong>Termination Checks</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Unoptimized</strong></td>
</tr>
<tr>
<td>Empty Loop</td>
<td>10,000,043</td>
</tr>
<tr>
<td>Loop Field</td>
<td>55,000,257</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
</tr>
<tr>
<td>Loop Method</td>
<td>60,000,117</td>
</tr>
<tr>
<td>Invocation</td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td>21,000,118</td>
</tr>
<tr>
<td>Synchronization</td>
<td>60,000,115</td>
</tr>
<tr>
<td>Input/Output</td>
<td>200,958</td>
</tr>
</tbody>
</table>

Table 2.1: Average total number of termination checks performed for each benchmark.

2.3.3 Termination Check Overhead

To gauge the actual impact of our class file modifications, we counted the number of times we checked the termination flag for each benchmark. This gave us an idea of how much extra work each rewritten program was actually doing. The results for all benchmarks are listed in Table 2.1.

The microbenchmarks essentially perform one simple operation repeatedly in a tight loop; as expected, all microbenchmarks performed one termination check per loop iteration, with few additional overhead checks. These results translate to roughly 40,000,000 checks performed for every second of run-time overhead for all but the input/output microbenchmark. This evaluates to around 10 CPU cycles of normal computation for each check performed.

For the input/output microbenchmark, however, only around 970,000 termination checks are performed per second of overhead. This can be almost entirely attributed to the additional overhead of the blocking call management code. It is also important to keep in mind that the cost of performing I/O far outweighs the cost of termination checks.

The application benchmarks reflect the results of the microbenchmarks. OTP, which suffered a much greater performance impact from the modifications than either Jess or JavaCup, performed over four times as many checks as JavaCup. Although Linpack performed fewer checks than either, it is a much shorter-running benchmark. For all of these
benchmarks, the results translate to between 15,000,000 and 29,000,000 checks per second of overhead.

These performance figures seem to indicate that for real-world applications, the slowdown will be roughly proportional to how much the application’s performance is dependent on tight loops. Applications that have tight loops may experience a slowdown of at worst a factor of two and more commonly 15 to 25%. Applications without tight loops can expect a more modest slowdown, most likely below 7%. The number of termination checks the system can perform per second seems not to be a limiting factor in system performance.

2.4 Summary

We have introduced a concept we call soft termination, along with a formal design and an implementation for Java, that allows for asynchronous and safe termination of misbehaving or malicious codelets. Soft termination can be implemented without making any changes to the underlying language or run-time system. Although our implementation is for Java, the basic design of soft termination does not depend on Java, and can be used with a variety of different languages.

Our Java implementation relies solely on class-file bytecode rewriting, making it portable across Java systems and easier to consider applying to non-Java systems. In real-world benchmarks, our system shows a slowdown of 3 to 25%. This could possibly be further reduced if we could leverage a safe point mechanism already implemented within the JVM.
Chapter 3

Transactional Rollback

In the last chapter, we developed a language-based mechanism for terminating codelets. The main impetus behind this work was to allow for safe and effective termination without impeding a codelet's ability to share state. However, this shared state can still become corrupted when a codelet is asynchronously terminated. Restarting the codelet can have unpredictable results.

In this chapter, we introduce transactional rollback, our solution to the problem of restarting terminated codelets. Section 3.1 discusses our design goals and presents and justifies our design. In Section 3.2, we discuss a Java-based implementation of our transactional rollback design, and address a number of implementation-specific issues. Section 3.3 presents performance results. We summarize transactional rollback in Section 3.4.

3.1 System Design

The ability to terminate codelets, assumed to be executing transiently in an otherwise long-running system, is a necessary property of this long-running system. We have addressed this problem with soft termination, discussed in the previous chapter. However, as a result of a codelet's untimely termination, shared state might become inconsistent. To address this, we need a mechanism to return the system to a known-consistent state.

A number of possible designs exist for rolling back state in a language run-time system. We begin by explaining how transactions are an appropriate mechanism for rollback. We then explore the range of possible designs, and explain how we chose to approach the problem. We also address issues in the design of rollback.
3.1.1 Rollback and Transactions

The most straightforward solution for designing language-based rollback is to simply keep a record of changes made by a codelet, and roll the changes back if the codelet is terminated. However, due to the concurrent nature of multi-threaded language systems, where multiple codelets may read and write to shared data in parallel, termination could still result in inconsistent state. Two general types of data conflicts complicate our design, read-after-write conflicts and write-after-write conflicts.

Write-after-write conflicts occur when two codelets write to the same variable. If one of the codelets is terminated, it becomes unclear how to roll back that transaction’s writes to the variable. Read-after-write conflicts occur when one codelet reads a variable another codelet has previously written to. If the writing codelet is terminated, the value read by the reading codelet becomes invalid, and the reading codelet must now be terminated, or at least restarted, as well.

Write-after-write and read-after-write conflicts are well known in the domain of databases. They are generally addressed by ensuring that all state is operated on strictly by ACID (atomic, consistent, isolated, durable) transactions. In particular, we can prevent these conflicts by constraining the order of operations within transactions to prevent the offending cases. The resulting order is said to have the properties of \textit{serializability} and \textit{strictness}. Conveniently, a well known locking protocol, \textit{strict two-phase locking}, guarantees these properties. Any textbook on databases, such as Silbershatz et al. [72], provides a more in-depth coverage of this material.

For our system, then, each codelet runs within a transaction, utilizing the system memory as a database. In the rest of this section, we show how this can be integrated into a language system. We also discuss some issues that arise in adding database functionality to a language run-time system for supporting transactional rollback.

3.1.2 Architecture of Transactional Rollback

Transactional rollback has a number of similarities in design to language persistence. Persistence is the notion that the state of a program is maintained even in cases as extreme as
the computer rebooting. To accomplish this, the system must keep track of which data a program accesses. At prescribed points in the code, the system must save the state to disk, making it persistent. It must also deal with system failures before the data has been written to disk. Similarly, transactional rollback must keep track of which state a codelet has modified. If the codelet is terminated, the changes the codelet has made must be rolled back to stored, stable values. The system might track this meta-state at many different levels.

Likewise, there are numerous parallels between transactional rollback and language security. In language security, the basic goal is to protect system-level invariants, such as codelet separation and resource limitations. Transactional rollback is concerned with protecting unspecified invariants at the user level. Like language persistence and language security, we can design transactional rollback to run below the language run-time system, inside the run-time system, or above the run-time system.

Below the Run-Time System

At the lowest level, transactional rollback can be designed to operate below the language run-time system, generally as a service of the operating system. Since operating system mechanisms are simple and well-understood, greater assurance can be provided by adding transactional behavior at this level. However, the operating system only sees pages and words in memory, and does not understand the semantics of the data it is operating on. As a result it cannot take advantage of these semantics to prevent unnecessary contention for system resources.

Numerous persistent object systems run at the operating system level. The operating system can use the page access patterns of running codelets to determine which pages of memory need to be made persistent. This is the approach taken by such persistent systems as Grasshopper [28], KeyKOS [39], and others. Similar approaches are taken by software distributed shared memory systems [65] to propagate changes. Finally, Howell [47] describes an implementation of Java persistence which operates above the operating system but below the language run-time system. A more complete discussion of operating system support for persistence and transactional systems is provided by Dearle and Hulse [27].
The operating system can also be used to enforce language security invariants. For language security, the operating system already has built-in protections for cross-domain access of state. However, the operating system uses processes to separate protection domains, so individual codelets have to be run in separate JVM processes. At a more extreme level, one can take advantage of the separation afforded by running the codelets on different machines entirely. Several systems use these mechanisms to provide language security [55, 73].

Inside the Run-Time System

We can also implement transactional rollback as a customization to the language run-time system. With the language run-time system’s semantic understanding of the language’s data structures, we can provide transactional rollback at the granularity of these data structures. This more precise granularity can result in fewer cases of false sharing contention (that is, cases where two codelets are accessing memory that is within the same memory page, but is actually used for separate and unrelated objects). This granularity has also been shown to reduce contention in distributed shared memory systems [48].

The language run-time system implementation itself does not necessarily suffer from the performance or design constraints imposed on codelets running above the language run-time system. However, this approach suffers from a lack of portability; to provide transactional rollback in different implementations of the same programming language, the design must be re-implemented for each language run-time system.

Language persistence is commonly integrated into the language run-time system. This approach is the earliest in persistent systems. Persistent systems grew up around such early persistent programming languages as PS-algol [4] and Elle [1], as well as the later Napier88 [26], in which persistence support existed as a part of the language run-time system. The earliest persistent object system was POMS [16]; more recent examples include Thor [54] and Mneme [58]. Java has been a specific target of modifications to support persistence, with such systems as PJama [64].

Finally, the language run-time system can provide enforcement of language security. The language run-time system understands the data structures a program is using, and can
use this information to provide more precise protection. In fact, many systems enforcing
language security exist as part of the language run-time systems because these aspects of
language security are actually integrated into the design of the language. A number of Java
resource management systems also rely on customizations to the JVM [5, 6, 8, 79]. Some,
such as PERC [60], even go so far as to modify the language, in the case of PERC adding
primitives to ensure atomic execution of blocks of code and imposing run-time restrictions
on these blocks. Vino [71] directly tracks the resources used by its codelets, allowing for a
limited form of transactional rollback.

Above the Run-Time System

Designing transactional rollback to run on top of the language run-time system solves the
issue of portability. Since the transactional rollback system is designed in terms of the
language itself, any implementation of the language run-time system can use the transac-
tional rollback system unmodified. What’s more, code-to-code transformations are well-
understood in language theory, and a high-level design based on code-to-code transforma-
tions could be more easily adapted to work with many different programming languages.
However, these code-to-code transformation systems suffer performance and flexibility
penalties because they are unable to modify all aspects of the underlying language run-time
system. In designing soft termination, discussed in Chapter 2, facing these same tradeoffs,
we chose to implement our system as a code-to-code transformation, and the performance
penalties were still quite reasonable (worst case benchmarks, doing numerical processing,
experienced an 18 to 25% overhead, where other benchmarks experienced only a 3 to 6%
overhead).

This approach has been used in designing language persistence. In such systems, the
codelet is transformed at run-time to provide the persistent system, also running in the
language run-time system, with access information for that codelet. Marquez et al. [56]
describe a persistent system implemented in Java entirely using bytecode transformations
at class load time.

Some aspects of language security can also be enforced on top of the language run-time
system. In these systems, the codelet is transformed as it is loaded by the run-time system
to make run-time checks enforcing language invariants. These run-time checks likewise run entirely on top of the language run-time system. In addition to soft termination, such checks have also been used for access control [35, 62, 80] and resource management [14, 21, 43, 51].

3.1.3 Design Discussion

Because we want a portable system, we have chosen to implement transactional rollback above the language run-time system using code-to-code transformations, as described in section 3.1.2 above. We begin by describing the transactional rollback transformation. We also describe some complications inherent in designing transactional rollback.

The Transactional Rollback Transformation

The code-to-code transformation for transactional rollback is relatively straightforward. First, each subroutine of the codelet is duplicated, and a parameter is added to the duplicate subroutines. This parameter represents the current transaction, and is used to identify the current codelet. In addition, all subroutine applications from duplicate subroutines are rewritten to instead call the duplicate-equivalent subroutines, passing the current transaction parameter along. This effectively duplicates the call graph of the entire system, forming separate transactional and non-transactional contexts for each codelet.

Each object instance is mapped one-to-one to a lock. The duplicate subroutines are rewritten such that before a codelet can access a data structure, the transaction which corresponds to the codelet must first acquire the lock corresponding to the data structure. As mentioned in Section 3.1.1 above, lock acquisition follows the strict two-phase locking protocol to ensure the system’s consistency. If a deadlock is detected, the codelet is terminated, and its modifications rolled back. The original subroutines remain unmodified, allowing for minimal overhead in the event that a non-transactional context is desired; the cost of this flexibility is a $2 \times$ overhead in the code size of the program.

When a write lock is granted, the corresponding data structure must additionally be backed up. A reference is thus maintained from every data structure to its shadow backup.
Figure 3.1: Object structure before a transaction has begun operation on the object. Before a transaction has begun operating on the object, its backup is empty and it points to an empty lock. The original object consisted only of the fields $c1$, $c2$, and $c3$. The backup and lock fields were added in transforming the object for transactional rollback.

Figure 3.2: Object structure after a transaction has read from the object. Transaction 1 has read from the object. It has a read lock on the object, but the object has not been written to, so no backup has been created. Grayed-out fields are those which are not modified in the operation.

Figure 3.3: Object structure after a transaction has written to the object. Transaction 1 has written to the object. It has a write lock on the object, and the backup has been allocated and filled with the old values of each field. Grayed-out fields are those which are not modified in the operation.
If the current transaction is aborted, any modifications will be rolled back, which depending on implementation may additionally require the maintenance of references from the backups to the original data structure. When the write lock is granted, a shallow copy of the data structure is made into the backup. Deep copies are unnecessary, as any data structures referred to by the current data structure are themselves backed up as needed. Note that a data structure is only backed up once per transaction, the first time the write lock is acquired. Figures 3.1 through 3.3 illustrate a transaction reading from and writing to an object.

If a transaction commits, any backups being maintained for that transaction’s write locks are thrown away. If the transaction aborts, its backups are restored. Aborts occur whenever the corresponding codelet abnormally terminates, either because of a bug, in response to a termination request by the system, or to break a deadlock. Locks are released when the transaction finishes, either by committing or aborting.

Deadlocks

Introducing transactions to a language, as in transactional rollback, can cause a previously deadlock-free program to deadlock. These deadlocks must be dealt with, either by preventing them beforehand or by detecting them after the fact. Deadlock prevention algorithms involve acquiring locks before they are needed and in a well-defined order. Unfortunately, codelets are not written to use any particular lock access discipline (and are not, in general, designed around the thought that they might be running in a transactional environment), and attempting to statically add such a discipline to a codelet would be infeasible without grossly overestimating the lock usage of a codelet. This makes it unreasonable to attempt an implementation of deadlock prevention.

Deadlock detection simply involves maintaining a directed graph of lock dependencies. If this graph ever develops a cycle, there is a deadlock. This can be determined by performing a simple graph traversal when a lock is requested. If a cycle is detected, the newest transaction in the cycle is terminated. This guarantees that at least one transaction in the system (the oldest) will never be terminated due to deadlock, and therefore that
deadlock will never prevent the system from making forward progress. These concepts are well-known in the field of databases [72].

Note that locks available to a language are not directly used in transactional rollback, although such language-based locks, if available, may be used short-term within critical sections of the lock manager. As a result, our deadlock-detection scheme does not address deadlocks which are preexisting in the codelet. This also introduces the possibility that codelets may deadlock on a mix of normal language-based locks and transactional rollback locks.

If the lock-state of language-based locks is accessible to our deadlock-detection algorithm, we can incorporate this information to detect such “mixed” deadlocks. Otherwise, we could implement a timeout for transactional rollback locks. When some application-specific heuristic function determines that no forward progress is occurring (for instance, by detecting that no lock activity has occurred for one minute), we can begin aborting transactions until the system resumes making forward progress.

3.2 Java Implementation

We chose to implement transactional rollback for the Java programming language, giving us experience that directly applies to a popular language and which can be easily applied to other object-oriented languages. We also favored Java for the presence of preexisting tools to help implement and debug the code-to-code transformations.

In this section, we discuss how to transform Java code to support transactional rollback. We then discuss a number of implementation issues and how we addressed them. Some are Java-specific, and would not pose a problem for other language systems, while others are universal.

Our implementation of transactional rollback utilizes Java bytecode rewriting; we use IBM’s JikesBT\(^1\) bytecode toolkit. Examples of how Java source code is modified to support transactional rollback can be found in Figures 3.4 through 3.7. Note that while the transformer actually operates on Java bytecode, there is a straightforward mapping from

\(^1\)http://www.alphaworks.ibm.com/tech/jikesbt
void foo(...) {
  ...
  bar(...);
  ...
}

void foo(...) {
  ...
  bar(...);
  ...
}

void foo.transact(..., Transaction t) {
  ...
  bar.transact(..., t);
  ...
}

Figure 3.4: Transactional rollback transformation applied to method definitions and method calls in Java.

\(t\) represents a Transaction object which is passed from method to method along the call stack.

the targeted bytecodes to the corresponding Java source. We present Java source code for the purpose of clarity.

3.2.1 Locking Code Insertion

The systems we have in mind in our design of transactional rollback include those which run potentially untrusted code. As a result, we make no assumptions about the code as it is input into the system, and the system itself must ensure that the code has been transformed to support transactional rollback. We can ensure this by performing the transactional rollback transformations in the class loader, as the class is being loaded into the JVM.

A compiled Java program is represented as a set of .class files, each of which contains the bytecode for a single Java class. These classes are loaded, typically on-demand, by a class loader. Class loaders are built into most JVMs and can be extended to support other functionality, such as rewriting code as it is loaded. By performing code transformations in the class loader, the point through which all code is loaded into the system, we can guarantee that all code in the system will be consistently transformed.

The transformer implements the design described in Section 3.1.3 for transactional rollback. All methods are duplicated, and a Transaction parameter is added to the end of the parameter list of the duplicate methods. All method calls within these duplicate
void bar(...)
{
    ...
    o1.i1 = o2.i2;
    ...
}

void bar(...)
{
    ...
    o1.i1 = o2.i2;
    ...
}

void bar.transact(..., Transaction t)
{
    ...
    o2.readLock(t);
    o1.writeLock(t);
    o1.i1 = o2.i2;
    ...
}

Figure 3.5: Transactional rollback transformation applied to object field accesses (read and write).

o1 and o2 represent arbitrary Java objects with fields i1 and i2, respectively.

methods are then rewritten to call the duplicates of the original targets, passing along the Transaction argument from the caller to the callee. Finally, all state accesses in these duplicate methods are preceded by calls to the lock manager. These transformations are illustrated in Figures 3.4 and 3.5.

A number of instance fields need to be added to every class. First, every instance must have a reference to a lock object. Since we employ lock state sharing (see Section 3.2.2), any state specific to the object (such as other transactions waiting for the object to become free) need also be stored in the object. Finally, we need a way to store backup data for when a codelet must be rolled back. To address this, two new Java classes are created. The instance backup class stores backups for instances variables; it is instantiated on demand by the backup routine. The static backup class stores backups for static class variables; it is instantiated when the class is initialized. This transformation is illustrated in Figure 3.6.

The backup operation, when called on an object, saves the fields of that object into the respective backup object, instantiating the backup object if necessary. The fields are copied using Java’s assignment operator. If the field is a primitive, then rollback will restore the appropriate value. If the field is an object, then any modifications of that object require write locks on that object. If the backup operation is working on an array, as opposed to an
public class Foo {
    Object o;
    static float f;
}

public class Foo implements TransObject {
    Object o;
    static float f;
    Foo$Backup foo.backup;
    static Foo$StaticBackup foo.static;
    Lock lock;

    public void readlock(Transaction t) {
        ...
    }

    public void writelock(Transaction t) {
        ...
        foo.backup = new Backup(o);
        ...
    }

    public void commit(Transaction t) {
        ...
        foo.backup = null;
        ...
    }

    public void abort(Transaction t) {
        ...
        o = foo.backup.o;
        ...
    }

    public class Backup {
        Object o;

        Backup(Object _o) {
            o = _o;
        }
    }

    public class StaticBackup implements TransObject {
        float f;
        Lock lock;

        Backup(float _f) {
            f = _f;
        }
    }
}

Figure 3.6: Transactional rollback transformation applied to an arbitrary class.
The transformation causes the creation of static and dynamic backup classes. We have also shown the lock management functions created for managing the instance data of the class. The equivalent functions for the static data are put in the StaticBackup class; these are not shown.
object, similar backups are made, but some special handling is necessary. See Section 3.2.3 for details.

The static backup class services an additional purpose. When a class has been rewritten, it is also declared to implement the TransObject interface. This interface allows the transactional system to operate on transaction-enabled objects in a uniform manner. As static classes cannot be passed as parameters and cannot be operated on through an interface, we instead use the static backup class. It is created as an implementation of TransObject, wrapping the static class for the transaction system.

3.2.2 Lock State Sharing

We implement two-phase locking at the granularity of object instances to guarantee consistency. We used object instance granularity as a matter of convenience. In particular, it allows us to use Java interfaces as the primary interface between the lock manager and the units of locking. We can also store some lock state in objects, eliminating the overhead of separately allocating additional objects.

Note that the lock manager must maintain lock state for each object in the system. There will tend to be many more objects in a system than active transactions operating on these objects, and multiple objects will tend to have the same lock state. Traditionally, a distinct lock object would need to be allocated, instantiated, and maintained for each object that was created. To take advantage of this pattern, we use NLSS [22], a form of lock state sharing. In such a system, objects with the same lock state (that is, objects locked by the same transactions and in the same modes) will have references to the same lock object.

NLSS also speeds the process of unlocking objects when a transaction completes. Because multiple objects may share the same lock state, removing the completed transaction from the lock state of each locked object in turn may not be necessary. Instead, we iterate over the table of unique lock states, removing the finished transaction from each. This eliminates the need to maintain mappings from each transaction to the objects that the transaction has locked for reading. We still must maintain a mapping from each transaction to the objects it has write-locked, for the purpose of rollback. We expect this set to be much smaller, however.
3.2.3 Arrays

Arrays are treated specially in Java. On the one hand, they are instances of `java.lang.Object`. They cannot be treated as primitives, because they are composite data structures. In addition, two separate objects could maintain references to the same array. As a result, it is necessary to maintain lock state for each array independently. On the other hand, it is not possible to add or change methods or fields of arrays, so they cannot be treated like other objects.

Since we must maintain the lock state for each individual array and we cannot modify the array implementation, we must maintain an external mapping from arrays to their lock state. A hash table is used to store this mapping. To lower the cost of hash table accesses, we memoize hash table queries. In the common case, when the lock state we are looking for is in the memo table, the additional work performed when locking an array versus a regular object is a method call to get the array’s `hashCode()` and a method call to retrieve the lock from the memo table. In practice, codelets which rely heavily on array manipulation experience significant slowdown in our system. Fixing this would require access to the Java implementation for arrays to directly add a reference to the array’s lock in the array object’s header.

One workaround is to replace each array with a wrapper (called a `box`) that contains the array as well as the lock state. This requires maintaining the class inheritance relationships between boxes to mimic the relationships between arrays. In addition, all array accesses, even those in non-transactional methods, need to be through boxes to maintain the consistency of state.

3.2.4 Constructors

Constructor methods also need to be dealt with specially in Java. Arbitrary work can be done in constructors, so they must be rewritten just like any other Java method. Java constructors are also responsible for initializing variables to appropriate default values. This includes the new fields which are introduced as part of the transactional rollback (e.g., the lock field, the backup fields).
We observe an opportunity for optimization here, as noted in Daynès and Cza-
jkowski [22]. When a codelet running inside a transaction instantiates a new object, it
is only visible to that codelet until either the transaction aborts, and the object becomes
unreachable; or the transaction commits. Furthermore, the very act of instantiation writes
to the object.

Therefore, when an object is instantiated, we initialize the lock pointer of the object to
the single write owner for the transaction. The single write owner is a lock structure which
identifies the transaction as having an exclusive write lock. Since lock states are shared,
we only need to create the single write owner once for each transaction. This optimization
eliminates the need for newly instantiated objects to be explicitly (and inevitably) locked,
reducing overhead, particularly if we implement a fast-path, as discussed in Section 3.2.7
below.

3.2.5 Native Methods

Java programs can invoke native methods (methods not written in Java), which cannot
be transformed. In particular, this means that we cannot pass Transaction objects to
the native methods. This is problematic because the native code might then call back to
Java code, which then needs some way of getting the current transaction. Otherwise, any
field accesses made in the Java method are not protected by the transactional system’s
concurrency control.

To solve this problem, we rewrite Java classes to store the current transaction with
the current thread before calling a native method. If the native method then calls back to
the codelet, we can restore the current transaction. Our system only adds the transaction
restoration logic at upcall points that we know occur in our benchmark tests, such as when
threads are started.

However, native code can make upcalls to any Java method; all Java methods would
need to be converted to include the transaction restoration logic, similar to the analysis
performed in SAFKASI [80]. SAFKASI showed performance overheads of 15 to 30% for
passing its security context argument to all methods in the system. We would expect a
similar overhead here.
3.2.6 Open Files

As noted by Howell [47] and others, open files and network connections pose a problem because they can be used to store state where the transactional system cannot roll it back. They also themselves represent state (such as file offsets) that is managed natively by the language run-time system or operating system. Should the codelet terminate prematurely, there is no way for the transactional system to roll back any writes to files or onto the network, or otherwise restore the state of the object.

We observe, however, that multiple instances of a codelet will not tend to share the same open file descriptor or network socket. As a result, if a codelet terminates prematurely, all of its open file descriptors and network sockets, which were created while the codelet was running, become inaccessible. If multiple instances of a codelet share an open file in an environment where codelets can fail asynchronously, the system is equally at risk of inconsistent data in files with or without transactional rollback; we do not exacerbate the problem.

3.2.7 Optimizations

We implemented a number of optimizations on the code, and can foresee a number of others that may speed the code up considerably. The optimizations we did implement were inspired by the results of profiling runs of the code. This is discussed in Section 3.3.3. The simplest optimization we use is a fast path. As part of the thread locking transformation, we insert code inline to check whether the locking operation is necessary. Only if we determine that a lock needs to be acquired do we actually attempt to acquire it. This will generally save at least one method call per field access, and could potentially save many more.

As a more ambitious optimization, we observed that the most commonly accessed object in a method is this, the object upon which instance methods generally operate. In almost all cases, a method will access its this object. As a result, if we can statically determine that a method accesses the this object at least once, we can acquire a single lock for that object at the beginning of the method. If any accesses are writes, we acquire a write lock. Otherwise, we acquire a read lock. All subsequent locks of this are omitted.
We would like to extend the above to apply to all objects in a method. If we could statically determine that multiple locks are acquired for the same object, all but the first could be removed. Similarly, if a read lock request and a write lock request were made on the same object from the same method, the read lock request would be redundant.

To address this problem, we could use a variation of lazy code motion [49, 30], a well known compiler optimization. Using this technique, we could eliminate more of the redundant lock acquisitions within a method. Since this method has been shown to be optimal, any redundant lock requests that can be detected will be detected with this method. Lazy code motion also guarantees that no code paths are extended, meaning a lock request is only made if it is necessary. Hosking et al. [45] discuss similar methods for optimizing such a system. Note that with any of these methods, there still needs to be one lock acquisition per object per method. Interprocedural analysis may help further eliminate these lock operations.

3.2.8 Integration into a Real-World System

As mentioned in Section 3.1, we assume that this system is already running in a pseudo-transactional environment. That is, there is a long-running system with a number of transient codelets running concurrently in separate threads and providing some services to the system. In this case, starting a transaction coincides with starting a codelet thread: the method Transaction.doTransaction() is called with a java.lang.Runnable object as a parameter. The run() method of this Runnable object, as rewritten by the transactional rollback transformer, is the entry point to the transaction. This mechanism is shown in Figure 3.7. When the method returns, either normally or abnormally, the transaction is completed, either having committed or aborted, respectively. If the transaction was aborted, an exception is thrown. The system hosting the codelets might choose to implement a loop which automatically restarts codelets if they are aborted.
public class Foo {
    void foo() {
        Transaction.doTransaction(
            new Runnable {
                void run() {
                    bar();
                }
            });
    }

    void bar() {
        ...
    }
}

public class Foo implements TransObject {
    void foo() {
        Transaction.doTransaction(
            new Runnable {
                void run() {
                    bar();
                }
                void run.transact(Transaction t) {
                    bar.transact(t);
                }
            });
    }

    void bar() {
        ...
    }

    void bar.transact(Transaction t) {
        ...
    }
}

Figure 3.7 : Transactional rollback transformation applied to a code block which starts a transaction.
The run() method of the Runnable object is duplicated and the transaction parameter is added; this method is the entry point for the transaction. Note the duplicate Foo.foo.transact() is not shown.
3.3 Performance

We measured the performance of our system using an AMD Athlon workstation (1.2 GHz CPU with 512 MB of RAM, running Linux version 2.4.16) and version 1.3 of Sun's Java 2 JVM, which includes the HotSpot JIT. We used two sorts of benchmarks, microbenchmarks that measure the performance of specific Java language constructs, and some real-world programs. We performed three different measurements for each benchmark. The first was the unmodified benchmark. In the second, the class files were rewritten, but were not executed in transactional contexts. This case was designed to strictly show the performance overhead of the larger class files and additional per-object data storage (with nothing stored there, but taking up more space). Finally, we ran each benchmark in a transaction to measure the total overhead of the transaction system. For the application benchmarks, we further measured the performance with an "array cheat" which we will describe below.

3.3.1 Microbenchmarks

We first ran a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, method and field accesses, exception handling, synchronization, and I/O. We used a microbenchmark package developed at University of California, San Diego, and modified at University of Arizona for the Sumatra Project\(^2\). The results are shown in Figure 3.8.

The microbenchmark results were entirely unsurprising. In the benchmark which focuses on field and array accesses, the overhead of the transactional system was significant (over 10× overhead). All other test benchmarks had significantly lower overhead. The only other microbenchmark with an overhead of more than 10% was the loop test, at 1.5×. However, note that this reflects an actual margin of 0.008 seconds, and the overhead shows up for the rewritten classes in both transactional and non-transactional contexts.

\(^2\)The original web site is http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html. The source we used was distributed from http://www.cs.arizona.edu/sumatra/ftp/benchmarks/Benchmark.java
Normalized Microbenchmark Performance

Figure 3.8: Transactional rollback overhead for microbenchmarks. This graph illustrates the performance of the rewritten classes run both inside and outside of a transactional context, relative to the performance of the corresponding original class files. The graph measures how many times longer the benchmarked class took to run than the original class. The left bars show the overhead when running the rewritten classes using only the non-transactional code. The right bars show the overhead when running the rewritten classes using the transactional code and in a transactional context.

3.3.2 Application Benchmarks

We benchmarked the real-world applications JavaCup\(^3\), Linpack\(^4\), Jess\(^5\), and OTP\(^6\), to get a feel for the performance impact of our transformations on real-world code. JavaCup is a LALR parser-generator for Java. Jess is an expert system shell. Linpack is a loop-intensive floating-point benchmark. OTP is a one-time password generator which uses a cryptographic hash function. The results are shown in Figure 3.9.

The first trend we noticed was in the overhead of the transformed code not running in a transactional context. Jess and JavaCup demonstrated relatively large overheads, compared

\(^{3}\)http://www.cs.princeton.edu/~appel/modern/java/CUP/
\(^{4}\)http://netlib2.cs.utk.edu/benchmark/linpackjava/
\(^{5}\)http://herzberg.ca.sandia.gov/jess/
\(^{6}\)http://www.cs.umd.edu/~harry/jotp/
Normalized Application Performance

Figure 3.9: Transactional rollback overhead for application benchmarks.
This graph illustrates the performance of the rewritten classes run both inside and outside of a transactional context, as well as inside with the array cheat enabled, relative to the performance of the corresponding original class files. The graph measures how many times longer the benchmarked class took to run than the original class. The left bars show the overhead when running the rewritten classes using only the non-transactional code. The middle bars show the overhead when running the rewritten classes using the transactional code and in a transactional context. The right bars show the overhead when running the rewritten classes in a transactional context with a cheat enabled to eliminate the hash table lookup on array accesses.

to OTP and Linpack. These overheads are roughly proportional to the sizes of the respective packages. The transformation added four instance fields and one static field to each class. In addition to two extra classes that need to be loaded per original class, the original class file is itself grown to over three times its original size. This code and memory bloat can be blamed for some of the overhead.

When transactions were enabled, the overheads were impressively large — performance overhead ranged from a factor of 6× to 23×. Clearly, these numbers indicate our system is not yet suitable for deployment in practice. We suspected that a major component of this overhead was related to our handling of arrays (see Section 3.2.3). To study this further, we implemented an “array cheat.” While no longer semantically sound, this
cheat eliminates the hash table lookups to find the per-array locks (one global lock is used), and it no longer performs backups of arrays. All other lock operations were performed unchanged. This cheat represents an upper bound on the performance benefit that might be achieved with a hypothetical extensible array implementation, giving us a slot per array to store the locks. Our measurements show significant gains relative to the original transactional system (reducing the overhead to between a factor of $2 \times$ and $15 \times$). Excluding Linpack, which is a fundamentally array-driven benchmark, the overhead experienced on the other application benchmarks with the array cheat was at most $5 \times$. The relative speedup on OTP was quite impressive (from $23 \times$ to $4 \times$). OTP allocates a large number of small arrays which it uses for temporary storage and which then quickly become garbage. Clearly, such a program introduces a large overhead when external references must be kept to each temporary array.

An interesting question is how these performance numbers compare to other systems that attempt to solve similar problems. When reading the literature on persistent object systems, we have not found many papers willing to compare their performance to the original, non-persistent system. Marquez et al. [56] present a code-to-code transformation that implements orthogonal persistence and compare its performance to JDK 1.2 with no support for persistence. They indicate their system, with a warm disk cache, has a roughly $9 \times$ performance cost relative to JDK 1.2 when running their test benchmark. This confirms that other persistent systems experience overhead similar to what we observed. However, further optimization is still necessary before this system is viable in production.

### 3.3.3 Optimization Profiling

In order to gain insight into the best methods for optimizing the system, we instrumented the transformed classes to provide an accounting of when locks were acquired. This includes attempts to acquire locks when the fast path determined the acquisition is not necessary. Our goal was to develop a strategy to allow the transformer to statically determine that a lock acquisition is not necessary and omit it.

In analyzing these results, an interesting pattern emerged: a large portion, and for some benchmarks, the vast majority, of redundant lock requests were on the `this` object. This
result is because a method will access fields of its own object far more frequently than fields of other objects. Note that it is never safe to remove all lock acquisitions for an object from a single function without performing inter-procedural analysis. However, there were still many redundant lock requests of this even without counting a required first lock acquisition.

Our solution, and the only optimization we performed for our system, was, whenever we could statically detect that a method would be accessing the this object, we acquire the appropriate lock to this at the beginning of the method and omit any lock requests for this anywhere else in the method. This reduces many redundant lock requests while adding limited deadlock pressure. The result is best illustrated in the OTP benchmark.

OTP uses the MD5 hash function to generate one-time passwords. The MD5 implementation keeps mathematical state in object fields, and performs long sequences of mathematical operations on them. Before this optimization, each mathematical operation needed to be prefixed by a lock acquisition. Afterwards, a lock acquisition was only needed at the beginning of each function. The result was a substantial drop in overhead for this benchmark. Smaller drops were observed in other benchmarks.

Even with this optimization, however, there is still considerable room for improvement. There are still a large number of subsequent lock acquisitions to the same object, read lock acquisitions followed closely by write lock acquisitions, and other redundant lock acquisitions. As discussed in section 3.2.7, data flow and control flow analysis could be used to identify redundant lock requests and reduce this overhead.

3.4 Summary

Termination is a crucial capability for providing resource control in language-based systems. In the face of data sharing, the ability to roll the system back to a safe state and to safely restart programs can be equally important. Transactional rollback provides a language-based portable solution to the problem of restarting codelets. Our design is independent of any particular language run-time system, allowing implementations which do
not depend on a single language. Our Java implementation shows a worst-case overhead of $23 \times$, with overheads of 6 to $7 \times$ in the absence of extensive array usage.

While these overheads are quite large relative to the original system’s performance, they represent a starting point for semantics that are otherwise unavailable to the designer of systems that must reliably execute untrusted or buggy codelets. A number of opportunities exist to further optimize our system, particularly with regard to making small modifications to the run-time system to allow us to annotate arrays with direct references to their backup information. With small run-time modifications, we believe we can offer transactional rollback semantics for codelets with reasonable performance.
Chapter 4

Bytecode Rewriting

In the previous chapters, we discussed two language-based mechanisms, soft termination and transactional rollback. These mechanisms were implemented in Java using the technique of bytecode rewriting. Bytecode rewriting is a common technique for adding semantics to Java. Bytecode rewriting is typically implemented either by statically rewriting Java classes to disk, or through dynamically rewriting classes as they are requested by a class loader. This process is illustrated in Figure 4.1.

In this chapter, we discuss our experiences using bytecode rewriting to implement our systems. We also discuss the roadblocks encountered in implementing a previous system called SAFKASI [80], which served as a precursor to and inspiration for our work. We describe our experiences in implementing these systems in Section 4.1. Section 4.2 discusses JVM design issues encountered when building these systems. Section 4.3 discusses optimizations we used to reduce the performance impact of our systems. We summarize this chapter in Section 4.4.

4.1 Bytecode Rewriting Implementations

This section discusses the implementations of three security systems built with Java bytecode rewriting. The first, SAFKASI [80], uses bytecode rewriting to transform a program into a style where security context information is passed as an argument to every method invocation. Soft termination, discussed in Chapter 2, is a system for safely terminating Java codelets by trapping backward branches and other conditions that might cause a codelet to loop indefinitely. Transactional rollback, discussed in Chapter 3, allows the system to undo any of the side-effects made to the system’s state by the codelet after soft termination has terminated the codelet, returning the system to a known stable state.
Figure 4.1: How a Java bytecode transformation changes the process of loading a class. If an already loaded class, `Foo`, uses an as-yet undefined class `Bar` (either accesses a static member or creates an instance) (1), the JVM traps the undefined reference to `Bar` (2), and sends a request for the class loader to load the class (3). The class loader fetches the class file (`Bar.class`) from the filesystem (4). In standard Java, the input class is then loaded into the JVM (5). In a bytecode rewriting system, the bytecode transformer is first invoked to transform the class (4a). In either case, the class is now loaded in the JVM (6).

4.1.1 SAFKASI

SAFKASI [80] (the security architecture formerly known as stack inspection), is an implementation of Java’s stack inspection architecture [41] using Java bytecode rewriting. SAFKASI is based on security-passing style, a redesign of stack-inspection which provably maintains the security properties of stack-inspection, improves optimizability and asymptotic complexity, and can be reasoned about using standard belief logics.

Stack inspection is a mechanism for performing access control on security-sensitive operations. Each stack frame is annotated with the security context of that stack frame. Stack inspection checks for privilege to perform a sensitive operations by inspecting the call stack of the caller. Starting at the caller and walking down, if it first reaches a frame that does not have permission to perform the operation, it indicates failure. If it first reaches
a frame that has permission to perform the operation and has explicitly granted permission, the operation is allowed. Otherwise, the default action is taken as defined by the JVM.

With security-passing style, instead of storing the security context in the stack, the security context is passed as an additional parameter to all methods. This optimizes security checks by avoiding the linear cost of iterating over the call stack.

SAFKASI is implemented by passing an additional parameter to every method in the system. This parameter is the security context. It is modified by the says operator, which is used by a class to explicitly grant its permission for some future operation. The stack-inspection check simply checks this context for the appropriate permission.

**Performance.** SAFKASI was tested using the NaturalBridge BulletTrain Java Compiler, which compiles Java source to native binary code [59]. With CPU-bound benchmarks, SAFKASI-transformed programs executed 15 to 30% slower than equivalent stack inspecting programs.

### 4.1.2 Soft Termination

Soft termination is a technique for safely terminating codelets. The basis for soft termination is that the codelet doesn’t need to terminate immediately, as long as it is guaranteed to eventually terminate. We implemented soft termination by first adding a termination flag field to each class. We then instrument each method, preceding all backward branches with termination checks to prevent infinite loops and beginning all methods with termination checks to prevent infinite recursion. Detailed discussion of soft termination, including performance results, can be found in Chapter 2.

### 4.1.3 Transactional Rollback

Transactional rollback was designed to complement soft termination in a resource management system. Where soft termination seeks to prevent system state from being left inconsistent by a codelet’s untimely termination, transactional rollback seeks to preserve the consistency of the codelet’s persistent state. We do this by keeping track of all changes made by a codelet, and if the codelet is terminated, the changes are undone. Each codelet
is run in the context of a transaction to avoid data conflicts. We implement transactional rollback by first duplicating each method in all classes (including system classes). The duplicate methods take an additional argument, the current transaction, and in the duplicate methods, all field and array accesses are preceded by respective lock requests. Detailed discussion of transactional rollback, including performance results, can be found in Chapter 3.

4.2 JVM Design Issues

Bytecode rewriting is inherently different from language run-time system customization. Bytecode rewriting results in classes that are run inside a standard Java virtual machine. These classes treat the JVM as a black box; functionality internal to the JVM cannot be altered. Rewritten classes must by valid bytecode, according to the virtual machine specification [53]. However, because codelets might be malicious, we cannot assume that the code we receive as input was generated by valid Java source, and so cannot necessarily use the Java Language Specification [42] for guidance. Table 4.1 summarizes our bytecode rewriting implementations and associated wish list items.

4.2.1 JVM Internals

The basic distinction of bytecode rewriting as opposed to language run-time system customization is that with bytecode rewriting, operating on JVM internals is not possible.

Class Reloading

Once classes are loaded into the JVM, they can neither be unloaded nor reloaded. Likewise, one cannot control when a class’s static initializer will be called or when dead objects’ finalizers will be invoked. Java version 1.4 includes a feature in its debugger architecture called HotSwap, which allows classes to be reloaded\(^1\). Existing activation records continue to run with code from the original version of the class. Also, the new class’s static initializer

\(^1\)See [http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html](http://java.sun.com/j2se/1.4/docs/guide/jpda/enhancements.html) for more information.
<table>
<thead>
<tr>
<th>SAFKASI</th>
<th>Soft Termination</th>
<th>Transactional Rollback</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Added Fields</strong></td>
<td>• Security principal</td>
<td>• Termination flag</td>
</tr>
<tr>
<td></td>
<td>• Added security context parameter to each method</td>
<td>• Added safe point-like termination checks on backward branches and method calls</td>
</tr>
<tr>
<td></td>
<td>• Added security context parameter to all method invocations</td>
<td>• Wrapped blocking calls to register with soft termination system to allow interrupting</td>
</tr>
<tr>
<td></td>
<td>• Add Security-passing style security checks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Added stub methods for upcalls</td>
<td></td>
</tr>
<tr>
<td><strong>Modified Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Added Classes</strong></td>
<td>• Wrapping subclass for class with blocking native methods</td>
<td>• Added classes for static and instance backups of each existing class</td>
</tr>
<tr>
<td><strong>JVM Issues</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Closed world</td>
<td>• System classes</td>
</tr>
<tr>
<td></td>
<td>• System classes</td>
<td>• Native methods</td>
</tr>
<tr>
<td></td>
<td>• Native methods</td>
<td>• Bootstrapping</td>
</tr>
<tr>
<td></td>
<td>• Bootstrapping</td>
<td></td>
</tr>
<tr>
<td><strong>Wish List Items</strong></td>
<td>• Synchronization interface</td>
<td>• Synchronization interface</td>
</tr>
<tr>
<td></td>
<td>• Garbage collector interface</td>
<td>• Access to array implementation</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of the three systems based on bytecode rewriting. This table describes the transformations to Java bytecode, JVM-related issues, and items on the JVM wish list inspired by that particular project.
is not run again, and new fields added to existing classes will be null or zero. This new feature was not available when we were building our systems, and would have been a welcome addition, despite its limitations.

Memory Management

One of the features on our “wishlist” is the addition of hooks into the JVM’s memory system. Had we been able to exploit the garbage collector’s safe point traps, our soft termination system could have performed periodic checks with more flexibility and lower overhead than trapping backward branches. Also, by modifying the garbage collector, we might have been able to enforce memory usage policies that are more robust than those used in JRes and J-SEAL2, which assume that whoever allocated a block of memory should be responsible for “paying” for that memory. We implemented exactly such a mechanism to account for memory usage in IBM’s RVM [2] by modifying the RVM’s garbage collector [63]. New pluggable garbage collection systems for Java, such as GCTk [10], may allow us to implement such features without requiring changes to the underlying JVM. In addition, the Real-time Specification for Java\(^2\) allows programs to create individual memory regions treated differently by the garbage collector and select from which region memory is allocated.

Thread Scheduling

Thread scheduling is another black box subsystem of the Java virtual machine. No mechanism is provided for either replacing the thread scheduler or fine-tuning how threads are scheduled; the only interface provided is the \texttt{Thread.setPriority()} method. The Real-time Specification for Java does allow for replacing the thread scheduler, but has only recently been finalized.

\footnote{See \url{http://www.rtj.org/} for more information.}
Native Methods

Native methods, while not strictly part of the JVM, are also treated as black boxes. We cannot control where a native method might go, and how that native method might behave. Native methods might perform arbitrary computations and are not necessarily guaranteed to return in a timely fashion (e.g., I/O routines might block indefinitely while they wait for data to arrive).

Furthermore, we have no control over up-calls from native methods back to the Java classes which we do control. In particular, we have no access to the Java Native Interface (JNI) calls used by native methods to interface with Java. If we could intercept these calls, then we could transform native methods to see a view of the Java classes consistent with what the Java classes themselves see after being transformed. Since that is not an option with current JVMs, we have adopted a number of strategies to cope with native methods, described in Section 4.2.3.

Note that certain JVMs implemented in Java, such as BulletTrain and IBM’s RVM [2], use fewer native methods than JVMs implemented in machine code. The tradeoff is that they have more methods which are not native but are treated specially by the JVM. In our bytecode transformations, these methods need to be treated the same as standard native methods. See Section 4.2.3 for details on such special methods and how we handled them.

4.2.2 "Local" Java Semantics

The flexibility of Java bytecode is both its benefit and its curse. A bytecode rewriting system must deal with every feature in the bytecode. We first deal with “local” features, which is to say, aspects of the Java bytecode where an individual class or method can be rewritten without needing any knowledge of how other classes in the JVM might interoperate with it.

Constructors

Constructors have two properties which set them apart from normal methods. First, no operations, whether initialization or otherwise, can be performed until either the super-class
constructor or a same-class constructor has been called, although arguments to a super-class constructor may be legally computed. This presented a problem in the soft termination system.

Soft termination is designed to check the value of the termination flag at the beginning of every method, including constructors, but inserting this check into a constructor results in a verifier error. Instead, the best we can do is to check for termination immediately following the call to the super-class constructor. As a result, we may not be able to terminate a loop constructed with constructors that recursively invoke other constructors before calling their super-class constructor.

Transactional rollback requires every object to be locked before it can be modified. Normally, this is accomplished with the addition of a specialized lock instance to every object instance. However, while an object is being constructed, this lock instance will be null. To address this, we needed to add checks everywhere an object is accessed to check whether its lock instance is null, slowing down all locking code to handle this special case.

Exceptions

Exceptions (and errors) can introduce implicit or asynchronous control flows into a Java method. Asynchronous exceptions, induced by calls to Thread.stop(), can occur literally anywhere, and at any time. For this reason, among other reasons, Sun has deprecated the method. Implicit exceptions typically occur as a side-effect of a program error, such as a NullPointerException or the various arithmetic errors. Here, the control flow edge from the potentially faulty operation to the construction and throwing of an exception does not appear explicitly in the Java bytecode.

This caused problems for SAFKASI, which needs to pass its security context as a parameter to the exception constructors. The only available choices were to either save the security context in thread-local storage or to give up and pass no context. Using thread-local storage is expensive, requiring a hash table operation on every store, and the exception constructors for Java’s implicit exceptions appear to never perform security critical operations. As a result, SAFKASI gives up, allowing these constructors to be invoked with no security context, and letting them start over as if there was no call stack before them.
We also observe that, in Java bytecode, an exception handling block (that is, a try-catch block) is represented as offsets delimiting the try block and the offset for the start of the catch block. The JVM specification requires that the catch block start strictly after the try block ends. That is, an exception-handler cannot handle its own exceptions. However, Sun’s JVM does not enforce this restriction.

This can be used by a malicious codelet to evade a soft termination signal. The exception handler acts as a backward-branch; when an exception is thrown in the exception handler, control passes to an earlier point in the text of the program. If this exception always occurs, like the termination signal of soft termination, the result will be an infinite loop. We solved the problem by having the soft termination system detect and reject such exception-handling blocks.

**Arrays**

Transactional rollback needs to be able to save backup copies of all objects before they are written. For most classes, we can create “backup” fields for every original field in the class. Assigning the backup to refer to the original object is sufficient to preserve the original value of the backup. However, for arrays, this no longer works; there is no place in the array to store a reference to the array’s backup. Our solution is to maintain a global hash table that maps arrays to their backups. For each array, the backup array must be the same size as the original. Creating this backup requires copying the whole array. For codelets that make extensive use of arrays, whether large or small, this creates a significant overhead. Our preferred solution would be for Java to have a mechanism to let us add our own field to all arrays. Java’s arrays already track their length; we want one more reference that we can use transparently to the Java application.

**Threads**

Threads can be thought of as a special case of native methods which make up-calls to Java code (in this case, during thread initialization, the up-call happens to be on a new thread while control returns to the original thread). In both SAFKASI and transactional
rollback, we needed state computed in the parent thread to be sent to the child thread. This was performed by modifying java.lang.Thread to have an additional field where the parent thread can store context information in the child thread. This context is consulted when the child’s run() method is invoked by the thread run-time system.

Another important issue with threads is controlling them when they block. Blocking can be caused by synchronization primitives (see below) or by native I/O methods, which might not return right away. Luckily, all of the JVM’s methods which might block will respond to calls to Thread.interrupt(), causing the formerly blocked thread to immediately throw an exception and canceling the operation that was previously under-way. We used this mechanism with soft termination to signal blocking threads that we wished to kill.

However, in implementing soft termination, we found that some mechanism was still needed to determine which thread is blocking, and whether it was blocking on behalf of system code (which should not be interrupted) or on behalf of a codelet (which we want to interrupt). We chose to wrap blocking methods with code to register the current thread with the soft termination system as blocking before the call, and unregister it afterward. We use Java’s stack inspection mechanism to determine the context of the blocking call (system versus codelet). The soft termination system could now interrupt blocking threads as necessary.

**Synchronization**

The semantics of Java’s monitorenter and monitorexit bytecode instructions and synchronized method modifier cannot be changed through bytecode rewriting. When a deadlock occurs in Sun’s JDK, the only way to recover is by restarting the JVM. The JDK 1.4 debugging architecture provides a mechanism to address this (a debugger is allowed to forcibly pop activation records from a thread’s stack), which might be useful to clean up after deadlock. This can similarly be used to terminate threads that are in a deadlock situation.

Another issue which soft termination had to deal with was the exact semantics of the monitorenter and monitorexit bytecodes, which acquire and release system locks,
respectively. If these calls are not properly balanced, it becomes possible to lock a monitor in such a way that terminating the codelet will not cause the monitor to be released. Despite the fact that neither the JVM nor the Java language specifications allow such construction, current JVM bytecode verifiers accept such programs. Our soft termination system did not attempt to deal with this problem.

Verification

We have seen several cases where our own code had to effectively extend the Java bytecode verifier in order to guarantee correctness of our system. We saw these issues with Java’s synchronization and exception features. We also saw cases where Java’s verifier got in the way of perfectly sound program transformations, particularly with regard to the restrictions on how Java’s constructors invoke their super-class constructors.

Ideally, the Java bytecode verifier should be disentangled from the current class loading system to stand on its own. This would simplify the addition of new checks to the verifier, such as checks for undesirable exception and synchronization behavior, and it would make it easier to remove checks that, at least in our context, are unnecessary, such as the super-class constructor checks. Furthermore, a modular Java bytecode verifier would be quite useful to our systems as a mechanism for checking our input before we rewrite it, allowing us to make stronger assumptions about the quality of our input and reducing the opportunity for carefully crafted codelets to trick a code rewriting system into mistakenly outputting a rewritten codelet with more privilege than it should have been given.

4.2.3 “Global” Java Semantics

In many cases, particularly when we consider changes that effect the method signatures advertised by a class, we must consider how our changes interact with other Java classes, with native code, and even with classes that may not have been loaded into the system yet.
"Special" Methods and Classes

Every JVM has certain methods and classes which are special in some way. Sun's JVM, for example, doesn't allow the rewriter to add fields to java.lang.Class or java.lang.Object. The NaturalBridge Java system is even more restrictive, as it is implemented, itself, in Java, and has many "special" Java classes that provide the necessary machinery to implement language internals. Calls to these (privileged) classes are quietly replaced with the (dangerous) primitive operations that they represent. Changes to these classes are simply ignored.

If a global transformation could be applied to all Java classes in a consistent way, such as the security-passing style transformation of SAJKASI, then the resulting system would be perfectly self-consistent. However, once special methods and classes are added, everything becomes more difficult. Now, the system must keep a list of special classes and methods and treat calls to them as special cases.

For SAJKASI, this meant that the security context could not be passed as an additional argument to all methods. Instead, thread-local storage was used to temporarily hold the security context, while calling the original, special method. If that method just returned, then everything would continue as normal. If that method were to make a call to another Java method, it would call the method using the original Java method signature. The callee method does not exist, because the additional security context parameter is missing.

To support this, a wrapper method was added for every method in the system. These wrappers exist solely to receive calls from special methods, fetch the security context from the thread-local storage, and then invoke the proper target method. Luckily, this wrapper technique proved to be quite general. It also supports native method up-calls and Java reflection (although no attempt was made to hack Java reflection such that it would show all the classes as if they had never been rewritten). By pairing wrappers, one to convert from security-passing style methods to special methods, and one to convert from regular methods back to security-passing style methods on up-calls from special code, it was possible to maintain the illusion that the whole system supported security-passing style calling
conventions. We used a similar strategy in our transactional rollback system to manage the transactional state, normally passed as an additional argument to all methods.

**Inheritance**

Java’s class inheritance is also a complicating factor in global transformations. When a subclass overrides a special method, as described above, it inherits the “specialness” of the method. For example, `java.lang.Object.hashCode()` is a native method. Any other class can override this, providing a Java (or native) implementation. At an arbitrary call site, where `java.lang.Object.hashCode()` is invoked, there may not be enough information to determine a more specific type for the callee. Thus, the caller must assume the worst case: the callee is special. This requires saving the security context, and then making a call to the special method. Of course, if the concrete type was something other than `java.lang.Object`, and it had, in fact, overridden `hashCode()`, then control would enter a wrapper method which would recover the security context and call back into the world of rewritten code.

**Open World Java**

Java is an open world system. That is, classes can be loaded and the class hierarchy modified at run-time. The only restrictions that Java makes are that, for a class to be loaded, all of its super-classes must already be loaded. Even with this restriction, the open world assumption complicates code analysis. It is no surprise, then, that many systems that change Java’s semantics, including SAFKASI, assume a closed world (that once the system starts running, no more classes will be loaded). SAFKASI performs a class hierarchy analysis [25], reading every class in the system into a large data structure that allows a number of queries. SAFKASI uses this to determine if all possible callees of a given call site are “normal” methods. For the `java.lang.Object.hashCode()` example above, SAFKASI would conservatively conclude that the callee might be special, and therefore the security context should be saved. SAFKASI’s analysis is fairly simplistic, as it doesn’t take
advantage of data flow and control flow information to narrow its idea of the type of a given callee. Still, even this simple flow-invariant analysis provided for significant optimizations.

Because Java is actually an open world, every Java class to be inserted into the system can potentially invalidate a judgment made by the optimizer. As a result, the optimizer would need to back out the optimizations that now reside in code previously loaded into the system. As we discussed in Section 4.2.1, such functionality is only now becoming available in the JDK 1.4 debugging architecture, and might make such optimizations possible, even in an open world.

Bootstrapping

Bootstrapping presents a unique problem to bytecode rewriting systems. When the JVM is launched, it normally proceeds through the initialization of its core classes before loading any applications. These core classes are, by necessity, carefully designed to avoid circular dependencies in their static initializers. Circular dependencies can be particularly hazardous in JVMs implemented themselves in Java, where the very first classes to be initialized are all “special” to the system in some way, and are very fragile with respect to changes.

In the implementation of soft termination, we largely did not need to worry about bootstrapping because we only had to transform codelets, not the entire system. For transactional rollback and SAFKASI, it was necessary to transform everything (more precisely, everything that was not special, in some fashion or another). In practice, when implementing SAFKASI on the NaturalBridge Java system, there was a large list of “special” classes that could not be touched. Likewise, it was necessary to make sure the bootstrapping process could complete before any of the SAFKASI-specific classes were initialized. The solution employed was to pass null as the security context argument and to only allocate a security context, and thus cause the SAFKASI system to be initialized, right before starting a codelet. For transactional rollback, we have the benefit that we support two modes of operation: with and without transactions. As a result, the JVM can initialize itself normally, and we only transition to the transactional world when we are about to start a codelet. Transactional rollback, in practice, proved much simpler to debug.
4.3 Optimizing Bytecode Rewriting

In the development of our systems, as with SAFKASI, we used extensive profiling to determine where optimization would be effective. For example, if we suspect that, for a large number of call sites, the callee has a desirable property for optimizing the call site, then we would instrument the call sites to count exactly how many callees happen to have the property in question, and at which call sites. Using this style of analysis, we were able to rapidly focus our attention on the optimizations that might matter for our systems. Of course, running our programs with such profiling slowed them down, but these profiling checks are only actually included during code rewriting when we wish to gather profiling data.

**Soft Termination.** The unoptimized design of soft termination required a termination check to be inserted before every call site. For methods with multiple call sites, there would be one termination check added per call site. We measured, through profiling, that on average there was more than one termination check per method invocation. This led us to “push” the termination checks from the call site to the head of the callee, thus reducing the number of termination checks.

Next, we could easily determine which methods, having no outward method invocations, are guaranteed to return in a finite time. For these methods, the termination check at the beginning of the method is unnecessary and can be omitted. In some cases, this more than halved the overhead of soft termination relative to the overhead of the unoptimized soft termination system. These optimizations are also described in Section 2.2.5.

**Transactional Rollback.** In our transactional rollback system, we learned a number of seemingly obvious facts through profiling. We observed that, for most lock acquire operations, the transaction acquiring the lock was the same transaction already holding the lock. We also observed that, by far, the most common object to be locked in a given method is the this object.

These measurements led us to some simple optimizations with profound effects on system performance. By checking if the lock to be acquired is already held by the current
transaction, we generally saved at least one method call, sometimes more. Likewise, by checking if a method contains multiple locking operations on this and consolidating them to a single operation at the head of the method, we were able to remove a significant number of lock operations. These two optimizations alone bought us a 25% speed improvement. These optimizations are also described in Section 3.2.7.

SAFKASI. SAFKASI performed a number of different optimizations aimed at lowering the cost of computing transitions in the security context information for each method invocation. 10 to 37% of method invocations were measured to be to leaf methods (with no outgoing method calls), leading to optimizations similar to those described above for soft termination. Likewise, after implementing a global class hierarchy analysis, the security context information for 10 to 76% of the remaining call sites could be statically predicted. By combining these with a simple one-reference cache of the last security context transition computed at a given call site, virtually all security context transitions could be optimized away.

The operations for storing and retrieving context information from thread-local storage were then considered. By noting the current method every time one of these operations was performed, and keeping one counter for every method in the system, it was quickly revealed that most of the context stores were focused on calls to a relatively small number of methods: generally these were methods that were called from all over the system, such as Object.toString() on the result of Hashtable.get(). Further optimization would require implementing control and data flow analysis in order to infer, for example, what the concrete type emerging from the Hashtable might be, rather than just java.lang.Object. Having access to the analysis that is, no doubt, performed within the JVM's internal optimizer may have enabled such optimization.

4.4 Summary

Bytecode rewriting is an extremely powerful mechanism for modifying the behavior of Java by modifying the Java classes directly. It allows for essentially arbitrary changes to Java's semantics. It is portable in that it allows a system based on bytecode rewriting to run on any
JVM. There are a number of limitations to bytecode rewriting, as well. These stem from the fact that we cannot directly affect how the JVM processes a given bytecode instruction or set of bytecode instructions, except using a relatively small number of restricted interfaces.

Considering these limitations, bytecode rewriting systems will never be able to match the capabilities and performance of customized JVMs, which have access “under the hood” of the JVM. However, customized run-time systems suffer from a lack of portability. In addition, bytecode rewriting systems benefit from optimization courtesy of the JVM. With appropriate specialized optimizations of the sort discussed above, a system based on bytecode rewriting could be competitive performance- and capability-wise with a system based on language run-time system customization.
Chapter 5

Conclusions

Language-based systems offer several advantages over operating systems running traditionally compiled programs. Like operating systems, these systems routinely host potentially buggy or malicious code from external sources that are not necessarily trusted. Operating systems have mechanisms that let us safely run such programs. Although Java and other general-purpose language-based systems have good support for memory protection, authorization, and access controls, there is little support for termination. Without termination, a system can be vulnerable to denial of service caused by malicious or buggy codelets.

In this thesis, we have developed two mechanisms to help solve the problem of codelet termination. The first, soft termination, allows us to safely, asynchronously terminate codelets. Unlike many similar systems, soft termination places no limits on state sharing among codelets. We present a formal proof of the effectiveness and safety of soft termination. We also discuss a Java implementation based on bytecode rewriting. In real-world benchmarks, our system shows a slowdown of 3 to 25%. We discussed soft termination in Chapter 2.

In the presence of state sharing, the ability to roll the system back to a safe state and to restart programs safely can be as important as the ability to terminate a codelet in the first place. The second mechanism we developed, transactional rollback, enables us safely to restart terminated codelets. We discuss the design of transactional rollback as well as a Java implementation also based on bytecode rewriting. Our implementation shows a worst-case overhead of 23×, with overheads of 6 to 7× in the absence of extensive array usage. We discussed transactional rollback in Chapter 3.

Both of these mechanisms are designed as code-to-code transformations, and implemented in Java using bytecode rewriting. Bytecode rewriting is an extremely powerful mechanism for modifying the behavior of Java by modifying the Java classes directly. It
allows for essentially arbitrary changes to Java’s semantics. It is portable in that it allows a system based on bytecode rewriting to run on any JVM. There are a number of limitations to bytecode rewriting, however, stemming from the fact that we cannot arbitrarily affect how the JVM processes Java bytecodes. While these limitations can generally be worked around, the cost can be prohibitive. We discussed bytecode rewriting in Chapter 4.

With soft termination and transactional rollback, we have the beginnings of operating system-style process semantics implemented in a language-based system using language-based mechanisms. We can effectively terminate codelets that are misbehaving and safely restart them. While other systems provide similar capabilities, these systems by and large sacrifice the benefits of running programs in a language-based system. The next step is a complete resource management system for language-based systems. Soft termination and transactional rollback can serve as enforcement mechanisms for this resource management system.

What remains to be designed are effective monitoring and policy engines. Because language run-time systems allow and take advantage of threads and memory references that easily cross protection boundaries, traditional operating system notions of resource monitoring and usage policies don’t apply. It is necessary to come up with new ideas to address these problems in the world of language-based systems.
Bibliography


[40] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.


