Termination in Language-based Systems

Algis Rudys
and
Dan S. Wallach
Department of Computer Science, Rice University

Language runtime systems are increasingly being embedded in systems to support runtime extensibility via mobile code. Such systems raise a number of concerns when the code running in such systems is potentially buggy or untrusted. While sophisticated access controls have been designed for mobile code and are shipping as part of commercial systems such as Java, there is no support for terminating mobile code short of terminating the entire language runtime. This paper presents a concept called "soft termination" which can be applied to virtually any mobile code system. Soft termination allows mobile code threads to be safely terminated while preserving the stability of the language runtime. In addition, function bodies can be permanently disabled, thwarting attacks predicated on system threads eventually calling untrusted functions. Soft termination guarantees termination by breaking any potential infinite loops in mobile code. We present a formal design for soft termination and an implementation of it for Java, built using Java bytecode rewriting, which demonstrates reasonable performance (3-25% slowdowns on benchmarks).

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: object-oriented programming; D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages, Java, object-oriented languages*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls, Invasive software*; F.3.1 [**Logics And Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics And Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics, Program analysis*

General Terms: Languages, Security, Termination, Resource Control

Additional Key Words and Phrases: Java, Internet, applets, resource control, soft termination

1. INTRODUCTION

In recent years, many have turned to language runtime systems for enforcement of security in running mobile code. Language-based enforcement of security was popularized by Java [Gosling et al. 1996; Lindholm and Yellin 1996] and the Java Virtual Machine (JVM), which were adopted by Netscape for its Navigator 2.0 browser in 1995. Java promised an

An earlier version of this paper was published as Rudys, A., Clements, J, and Wallach, D. S., *Termination in Language-Based Systems*, Proceedings of the 2001 Network and Distributed System Security Symposium (February 2001, San Diego, California).
Address: Department of Computer Science, Rice University, 6100 Main St., MS 132, Houston, TX 77005-1892

environment where untrusted buggy or malicious code (hereafter called a "codelet"[1]) could 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 would run 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 would serve to prevent a malicious codelet from forging a reference to an arbitrary memory location and subverting the system. In addition to its applications within web browsers, codelets have also been touted for OS kernel extensions, active networking, extensible databases, agent-based negotiation systems, and other problem domains.

The *promise* of Java may be attractive, but a large number of security flaws have been discovered since its release [Dean et al. 1997; McGraw and Felten 1999]. Significant strides have been made at understanding the type system [Alves-Foss 1999; Stata and Abadi 1998; Drossopoulou and Eisenbach 1997; Drossopoulou et al. 1998; Dean 1997; Coglio and Goldberg 2000] and protecting the Java system classes from being manipulated into violating security [Wallach et al. 1997; Wallach et al. 2000; Gong 1999; Edjlali et al. 1998; Erlingsson and Schneider 1999], 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 [Malkhi et al. 1998; Sirer et al. 1999], surrendering any performance benefits from running the JVM together with its host application, or create a process-like abstraction inside the JVM [Back et al. 2000; Tullman and Lepreau 1998; Hawblitzel et al. 1998; Bernadat et al. 1998; Czajkowski and Daynès 2001]. These process abstractions either complicate memory sharing or make it completely impossible.

This paper describes a new language runtime-based mechanism called *soft termination*. While it is not specific to Java, soft termination can be deployed on Java, and we present a Java-based implementation. 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 may no longer be allowed to run. Soft termination provides semantics similar to the UNIX `ps` and `kill` commands, yet requires neither process-like structures nor limits on memory sharing to preserve system integrity after termination. Soft termination guarantees termination by breaking any potential infinite loops in codelets. Our design of soft termination is defined as a code-to-code transformation, and is thus more easily portable across languages and implementations of the same language.

Soft termination supports two kinds of program termination: safe thread termination and safe codelet disabling. Safe thread termination must deal with the possibility that the target thread is currently executing critical system code that may not necessarily be designed to respond to an asynchronous signal. Safe codelet disabling must deal with the possibility that future threads may invoke functions of the disabled codelet; the codelet should not be able to "hijack" the thread and continue execution.

In the following sections, we discuss the concept of soft termination, and present our implementation. Section 2 discusses how prior work has addressed our concerns. Section 3 formalizes and describes what we mean by soft termination. Section 4 describes our Java-based implementation of soft termination, and mentions a number of Java-specific

[1]The term "codelet" is also used in artificial intelligence, numerical processing, XML tag processing, and PDA software, all with slightly different meanings. When we say "codelet," we refer to a small program meant to be executed in conjunction with or as an internal component of a larger program.

issues that we encountered. We present performance measurements in section 5. Section 6 describes some experience with using soft termination in real-world situations. Finally, section 7 discusses future extensions to this project.

2. RELATED WORK

Systems such as Smalltalk [Goldberg and Robson 1989], Pilot [Redell et al. 1980], Cedar [Swinehart et al. 1986], Lisp Machines [Bromley 1986], and Oberon [Wirth and Gutknecht 1992] have taken advantage of language-based mechanisms to provide OS-like services. At least as early as the Burroughs B5000 [Burroughs Corporation 1969] series computers, language based mechanisms were being used for security purposes. 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 [Flatt et al. 1999] supports thread termination and management of resources like open files but has no way of disabling code from running in future threads. Some systems, such as PLAN [Hicks et al. 1998], 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)` [Sun Microsystems 1990] 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 [Nilsen et al. 1998], for instance, is an extension to Java which supports asynchronous exceptions while providing a mechanism for protecting system invariants in critical sections. A programmer may specify blocks with provable limits on their runtime and asynchronous exceptions are deferred while execution is in one of these blocks.

Chander et al. [2001] describe a system to target specific sorts of resource exhaustion attacks via bytecode instrumentation. The general technique presented here is to replace calls to sensitive methods (for instance, for setting thread priority or creating a new window) with calls to customized methods which first verify that the operation is not harmful. While 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 [Hawblitzel et al. 1998] 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 classfiles 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 [Czajkowski and von Eicken 1998] 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 [Binder 2001] 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 don't violate protection domain boundaries. The result is that sharing among codelets is restricted. Termination is guaranteed by a bytecode transformer which effectively prevents the codelet from catching a particular type of exception.

KaffeOS [Back and Hsieh 1999; Back et al. 2000] provides an explicit process-like abstraction for Java codelets. It is implemented as a heavily customized JVM with significant changes to the underlying language runtime 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) [Czajkowski and Daynès 2001] also uses JVM extensions to support separation of codelets, primarily to prevent the codelets from interfering in each other's execution. Bernadat et al. [1998] and van Doorn [2000] 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 micro-kernel. These systems provide a style of termination we call hard termination (see section 3.2).

3. SYSTEM DESIGN

A large space of possible designs exist for supporting termination in language runtimes. 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.

3.1 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[2]. Unfortunately, there are numerous reasons this cannot work in practice.

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 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

[2]see `http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitive-Deprecation.html`

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 runtime systems have support for making native OS system calls. A thread should not be descheduled while it is blocking, waiting for such a system call (*e.g.* an I/O call) to return.

Another naïve solution is to force an asynchronous exception, as done by Java's deprecated `Thread.stop()` operation. While 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 non-terminable thread. Finally, a workaround is needed to prevent user-level code from catching the exception.

3.2 Hard termination

Operating systems like 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 which 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.

3.3 Soft termination

Unlike a traditional operating system, the boundary between user and system code in a language runtime is harder to define. While 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 a thread could easily cross from user to system code and back many times, there may never be a correct time at which it becomes safe to terminate a thread.

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

3.3.1 *Key ideas.* Soft termination is based on the idea that a codelet may be 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. Furthermore, as with the Unix kernel, we would like the termination of a codelet to not disturb any system code it may be using at the time it is terminated, thus preserving 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 multi-

$$
\begin{array}{rcl}
P & = & \Gamma\, M \\
\Gamma & = & D \ldots D \\
D & = & (\textbf{define}\ \ (f\ x)\ M) \\
M & = & (f\ M) \mid (\textbf{if}_0\ M\ M\ M) \mid (\textbf{let}\ \ (x\ M)\ M) \mid (\textbf{try}\ \ M\ M) \mid (\textbf{throw}) \mid V \mid x \mid \\
 & & (\textbf{CheckTermination}) \\
V & = & c \in \mathbb{N} \\
f, x & = & \text{identifiers}
\end{array}
$$

Fig. 1. Simple language used for our analysis.

$$
\begin{array}{rcl}
E & = & [] \mid (f\ E) \mid (\textbf{if}_0\ E\ M\ M) \mid (\textbf{let}\ \ (x\ E)\ M) \mid (\textbf{try}\ \ E\ M) \mid \\
 & & (\textbf{CheckTermination}) \mid (\textbf{throw}) \mid x \\
\text{final states} & = & V \mid \textbf{error} \mid (\textbf{throw})
\end{array}
$$

Fig. 2. Evaluation context for reduction of the simple language.

$$
\begin{array}{rcll}
E[(f\ V)] & \mapsto & \begin{cases} E[V_0] & \text{if } \delta(f, V) = V_0 \\ E[[V\ /\ x]\ M]^{\dagger} & \text{if } (\textbf{define}\ \ (f\ x)\ M) \in \Gamma \\ \textbf{error} & \text{otherwise} \end{cases} & \\
E[(\textbf{CheckTermination})] & \mapsto & E[0] \text{ or } E[1] & \text{(depending on external conditions)} \\
E[(\textbf{if}_0\ 0\ M_1\ M_2)] & \mapsto & E[M_1] & \\
E[(\textbf{if}_0\ V\ M_1\ M_2)] & \mapsto & E[M_2] & \text{if } V \neq 0 \\
E[(\textbf{let}\ \ (x\ V)\ M)] & \mapsto & E[[V\ /\ x]\ M]^{\dagger} & \\
E[(\textbf{try}\ \ V\ M)] & \mapsto & E[V] & \\
E[(\textbf{try}\ \ (\textbf{throw})\ M)] & \mapsto & E[M] & \\
E[(f\ (\textbf{throw}))] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{if}_0\ (\textbf{throw})\ M_1\ M_2)] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{let}\ \ (x\ (\textbf{throw}))\ M)] & \mapsto & E[(\textbf{throw})] & \\
E[x] & \mapsto & \textbf{error} & \text{(unbound variables)}
\end{array}
$$

Fig. 3. An operational semantics for our 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. This is defined for our language in figure 4.

$$
\begin{array}{rcl}
[V\ /\ x]\ (f\ M) & \mapsto & (f\ [V\ /\ x]\ M) \\
[V\ /\ x]\ (\textbf{if}_0\ M_1\ M_2\ M_3) & \mapsto & (\textbf{if}_0\ [V\ /\ x]\ M_1\ [V\ /\ x]\ M_2\ [V\ /\ x]\ M_3) \\
[V\ /\ x]\ (\textbf{let}\ \ (x\ M_1)\ M_2) & \mapsto & (\textbf{let}\ \ (x\ [V\ /\ x]\ M_1)\ M_2) \\
 & & \quad \text{(the } \textbf{let} \text{ expression overrides the term being replaced).} \\
[V\ /\ x]\ (\textbf{let}\ \ (t\ M_1)\ M_2) & \mapsto & (\textbf{let}\ \ (t\ [V\ /\ x]\ M_1)\ [V\ /\ x]\ M_2) \\
 & & \quad \text{if } t \neq x \text{ (the } \textbf{let} \text{ expression introduces a new term).} \\
[V\ /\ x]\ (\textbf{try}\ \ M_1\ M_2) & \mapsto & (\textbf{try}\ \ [V\ /\ x]\ M_1\ [V\ /\ x]\ M_2) \\
[V\ /\ x]\ (\textbf{throw}) & \mapsto & (\textbf{throw}) \\
[V\ /\ x]\ (\textbf{CheckTermination}) & \mapsto & (\textbf{CheckTermination}) \\
[V\ /\ x]\ V' & \mapsto & V' \\
[V\ /\ x]\ x & \mapsto & V \\
[V\ /\ x]\ t & \mapsto & t \qquad \text{if } t \neq x
\end{array}
$$

Fig. 4. Lexical scoping rules for expanding $[V\ /\ x]\ M$ for any expression M.

(1)	$\mathcal{X}[\![\Gamma\ M]\!]$	$=$	$\mathcal{X}[\![\Gamma]\!]\ \mathcal{X}[\![M]\!]$
(2)	$\mathcal{X}[\![D_1\ \ldots\ D_n]\!]$	$=$	$\mathcal{X}[\![D_1]\!]\ \ldots \mathcal{X}[\![D_n]\!]$
(3)	$\mathcal{X}[\![(\textbf{define}\ \ (f\ x)\ M)]\!]$	$=$	$(\textbf{define}\ \ (f\ x)\ \mathcal{X}[\![M]\!])$
(4)	$\mathcal{X}[\![(f\ M)]\!]$	$=$	$(\textbf{let}\ \ (t\ \mathcal{X}[\![M]\!])\ ((\textbf{if}_0\ (\textbf{CheckTermination})\ (f\ t)\ (\textbf{throw}))))$ Where the identifier t occurs nowhere else in the program.
(5)	$\mathcal{X}[\![(\textbf{if}_0\ M_1\ M_2\ M_3)]\!]$	$=$	$(\textbf{if}_0\ \mathcal{X}[\![M_1]\!]\ \mathcal{X}[\![M_2]\!]\ \mathcal{X}[\![M_3]\!])$
(6)	$\mathcal{X}[\![(\textbf{try}\ \ M_1\ M_2)]\!]$	$=$	$(\textbf{try}\ \ \mathcal{X}[\![M_1]\!]\ \mathcal{X}[\![M_2]\!])$
(7)	$\mathcal{X}[\![(\textbf{let}\ \ (x\ M_1)\ M_2)]\!]$	$=$	$(\textbf{let}\ \ (x\ \mathcal{X}[\![M_1]\!])\ \mathcal{X}[\![M_2]\!])$
(8)	$\mathcal{X}[\![(\textbf{throw})]\!]$	$=$	$(\textbf{throw})$
(9)	$\mathcal{X}[\![(\textbf{CheckTermination})]\!]$	$=$	$(\textbf{CheckTermination})$
(10)	$\mathcal{X}[\![V]\!]$	$=$	V
(11)	$\mathcal{X}[\![x]\!]$	$=$	x

Fig. 5. The soft termination transformation.

tasking, inter-process and inter-task communication, barrier synchronization, garbage collection, and debugging functions. A good discussion on safe points is provided in Feeley [1993]. In Feeley's terminology, our implementation uses "minimal polling."

3.3.2 *Formal design.* For our analysis, we begin with a simple programming language having natural numbers, functions, conditional expressions, and simple exceptions (see figure 1). In our language, a program is a collection of function definitions (Γ) 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 4.2 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 [1992]. Figure 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 3 are applied to these contexts, defining the behavior of the language.

The semantics for our language define three possible ending states: V, **error**, and (**throw**). V represents a final state in which the program reduces to a value. The **error** state indicates that some error condition, such as a call to an undefined function, was reached in the evaluation of the program. The (**throw**) state occurs when the entire program reduces to (**throw**); it indicates that the program terminated as the result of an uncaught exception. While this is similar in nature to **error**, it is treated differently to indicate its different origin.

A (**CheckTermination**) expression reduces to a boolean value (0 or 1), indicating whether termination for the current codelet has been externally (and asynchronously) requested. Since (**CheckTermination**) behaves like a termination signal, we model it by assuming that when evaluation begins, (**CheckTermination**) evaluates to 0. Once (**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

Sample program:

$$\begin{aligned}
P &= (\mathbf{define}\ (f_1\ x)\ (\mathbf{if}_0\ x\ \ 1\ (\mathbf{throw})))\\
&\quad\ (\mathbf{try}\ \ (f_1\ (f_1\ (f_1\ 0)))\ (f_1\ 0))
\end{aligned}$$

$$\begin{aligned}
\Gamma &= (\mathbf{define}\ (f_1\ x)\ (\mathbf{if}_0\ x\ \ 1\ (\mathbf{throw})))\\
M &= (\mathbf{try}\ \ (f_1\ (f_1\ (f_1\ 0)))\ (f_1\ 0))
\end{aligned}$$

1	$(\mathbf{try}\ \ (f_1\ (f_1\ \boxed{(f_1\ 0)}))\ (f_1\ 0))$	$E[(f\ V)] \mapsto E[[V\ /\ x]\ M]$ if $(\mathbf{define}\ \ (f\ x)\ M) \in \Gamma$	Call to a user-defined function.
2	$(\mathbf{try}\ \ (f_1\ (f_1\ \boxed{(\mathbf{if}_0\ 0\ \ 1\ (\mathbf{throw}))}))\ (f_1\ 0))$	$E[(\mathbf{if}_0\ 0\ M_1\ M_2)] \mapsto E[M_1]$	Evaluation of $\mathbf{if}_0$ conditional, where the test is 0.
3	$(\mathbf{try}\ \ (f_1\ \boxed{(f_1\ 1)})\ (f_1\ 0))$	$E[(f\ V)] \mapsto E[[V\ /\ x]\ M]$	Call to a user-defined function.
4	$(\mathbf{try}\ \ (f_1\ \boxed{(\mathbf{if}_0\ 1\ \ 1\ (\mathbf{throw}))})\ (f_1\ 0))$	$E[(\mathbf{if}_0\ 1\ M_1\ M_2)] \mapsto E[M_2]$	Evaluation of $\mathbf{if}_0$ conditional, where the test is 1.
5	$(\mathbf{try}\ \ \boxed{(f_1\ (\mathbf{throw}))}\ (f_1\ 0))$	$E[(f\ (\mathbf{throw}))] \mapsto E[(\mathbf{throw})]$	Single-step reduction of (**throw**) occurring as a parameter to a function.
6	$\boxed{(\mathbf{try}\ \ (\mathbf{throw})\ (f_1\ 0))}$	$E[(\mathbf{try}\ \ (\mathbf{throw})\ M)] \mapsto E[M]$	Evaluation of **try** expression catching a (**throw**).
7	$\boxed{(f_1\ 0)}$	$E[(f\ V)] \mapsto E[[V\ /\ x]\ M]$	Call to a user-defined function.
8	$\boxed{(\mathbf{if}_0\ 0\ \ 1\ (\mathbf{throw}))}$	$E[(\mathbf{if}_0\ 0\ M_1\ M_2)] \mapsto E[M_1]$	Evaluation of $\mathbf{if}_0$ conditional, where the test is 0.
9	$\boxed{1}$		

Fig. 6. The steps in evaluating the program P. The left column shows the actual program in various stages of reduction. The subexpression which 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 3, that applies to the program at this stage. The right column describes the reduction.

the single-step reduction of (**throw**) expressions appearing as subexpressions in any other expression. When (**throw**) occurs as the body of a **let** or as the second or third parameter of an $\mathbf{if}_0$, 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 replace all bound variables with their values, according to the rules of lexical scope (see figure 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 to that program, and the result is another syntactically valid program. A formal proof of this property is beyond the scope of this paper.

Figure 6 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 $(\mathbf{throw})$ expression single-stepping out of a function call. Step 6 shows the same $(\mathbf{throw})$ expression being caught in a **try** expression.

In step 9, there are no more reduction rules to be applied, so the program returns the value 1.

3.3.3 *Proof of Result Preservation.* The soft termination transformation, $\mathcal{X}$, is described in figure 5. 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 $\mathcal{X}$ to catch any nested function applications. The other transformation rules describe how the transformation continues recursively on expressions.

We first prove that, so long as $(\mathbf{CheckTermination})$ is 0, evaluating any transformed expression $\mathcal{X}[\![M]\!]$ always results in the same final state as evaluating the corresponding expression M. That is, if termination has not been requested, the transformed program evaluates to the same final state as the original, untransformed program. We can show this by a structural induction on M, the grammar of expressions in our language.

In the base cases,

$$M_{base} = (\mathbf{throw}) \mid (\mathbf{CheckTermination}) \mid V \mid x$$

In each of these cases, $\mathcal{X}[\![M_{base}]\!] = M_{base}$, so the behavior of the expression is preserved.

In the inductive cases,

$$M_{inductive} = (\mathbf{if}_0\ M\ M\ M) \mid (\mathbf{let}\ \ (x\ M)\ M) \mid (\mathbf{try}\ \ M\ M) \mid (f\ M)$$

where

$$M = M_{base} \mid M_{inductive}$$

In the first case, $\mathbf{if}_0$ expressions are transformed by the rule:

$$\mathcal{X}[\![(\mathbf{if}_0\ M_1\ M_2\ M_3)]\!] = (\mathbf{if}_0\ \mathcal{X}[\![M_1]\!]\ \mathcal{X}[\![M_2]\!]\ \mathcal{X}[\![M_3]\!])$$

By the inductive hypothesis, evaluating each M_i has the same final state as evaluating the corresponding $\mathcal{X}[\![M_i]\!]$ as long as $(\mathbf{CheckTermination})$ is 0. Because the result of evaluating the $\mathbf{if}_0$ expression is solely dependent on these results, it is also unaffected by $\mathcal{X}$. 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 $\mathcal{X}$:

$$\mathcal{X}[\![(f\ M)]\!]\ =\ (\mathbf{let}\ \ (t\ \mathcal{X}[\![M]\!])\ ((\mathbf{if}_0\ (\mathbf{CheckTermination})\ (f\ t)\ (\mathbf{throw}))))$$

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

$$(\textbf{let}\ \ (t\ V)\ ((\textbf{if}_0\ (\textbf{CheckTermination})\ (f\ t)\ (\textbf{throw}))))$$

which further reduces to:

$$(\textbf{if}_0\ (\textbf{CheckTermination})\ (f\ V)\ (\textbf{throw}))$$

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

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

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

$$\begin{array}{rcll}
M_{lock} & = & (\textbf{if}_0\ (\textbf{CheckTermination})\ (f\ V)\ (\textbf{throw})\ )\ | & (a)\\
& & (\textbf{if}_0\ 1\ (f\ V)\ (\textbf{throw})\ )\ | & (b)\\
& & (\textbf{CheckTermination})\ | & (c)\\
& & (\textbf{if}_0\ M_{lock}\ M_{lock}\ M_{lock})\ | & (d)\\
& & (\textbf{let}\ \ (x\ M_{lock})\ M_{lock})\ | & (e)\\
& & (\textbf{try}\ \ M_{lock}\ M_{lock})\ | & (f)\\
& & (\textbf{throw})\ | & (g)\\
& & x\ | & (h)\\
& & V & (i)
\end{array}$$

Expressions *c*–*i* are derived directly from the definition of M in figure 1 above. 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 $(\textbf{CheckTermination})$ is 1.

To prove termination, we must show that, as long as $(\textbf{CheckTermination})$ is 1, then M_{lock} is closed under program stepping, and that the syntactic length of the program will be strictly decreasing.

Closure may be stated as follows: if $M \mapsto M'$ and $M \in M_{lock}$, then $M' \in M_{lock}$. By inspection, for all possible expressions in M_{lock}, we observe that our semantics preserve closure.

The syntactic length property may be stated as follows: If $(\textbf{CheckTermination})$ is always 1, then for $M, M' \in M_{lock}$ such that $M \mapsto M'$, we have that $|M'| < |M|$. 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 an expression in M_{lock}, all function applications are "fenced out" by a call to $(\textbf{CheckTermination})$.

When a program is being evaluated and $(\textbf{CheckTermination})$ is not guaranteed to be 1, a program may not be in M_{lock}. The case where this matters is when termination is requested just before an application; in this case, the next reduction step will be the application itself, which may increase the length of the program. However, this resulting state must necessarily be in M_{lock}, and at this point the program length will be strictly decreasing as reductions are applied. In other words, a transformed program which is not in M_{lock} is guaranteed to enter M_{lock} in one step.

Since program lengths are finite, a program transformed by $\mathcal{X}$ is guaranteed to finish in a finite number of steps, once the (**CheckTermination**) condition is raised. ■

3.4 Soft termination with codelet, system, and blocking code

While 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 7 and 9 introduce the syntax and semantics of the extended language. Section 4.3 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.

3.4.1 *Language extensions.* The distinction between system and user functions is especially relevant as a feature of mobile code; it allows for the 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 runtime system can be declared **system**; we assume that some pre-processor exists to remove any **system** declarations from user code. We use the term "system code" to refer to the collection of expressions declared 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 which behave like primitive operations, are declared explicitly, we have chosen to also declare primitive operations explicitly using the **primitive** keyword.

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 δ 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**.

3.4.1.1 *Blocking functions.* Blocking functions are those functions which 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 δ', similar to the δ function used to define primitives. δ' is a non-deterministic function indicating, based on external conditions, whether the application is defined and what value the corresponding blocking function application should reduce to. If the δ' 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

$$
\begin{array}{rcl}
P & = & \Gamma\, M \\
\Gamma & = & D \ldots D \\
L & = & \textbf{system} \mid \textbf{codelet} \\
D & = & (\textbf{define}\ \ L\ (f\ x)\ M) \mid (\textbf{define blocking}\ (f_{blocking}\ x)\ f_{nonblocking}) \mid \\
 & & (\textbf{define primitive}\ (f\ x)) \\
M & = & (f\ M) \mid (\textbf{if}_0\ M\ M\ M) \mid (\textbf{let}\ \ (x\ M)\ M) \mid (\textbf{try}\ \ M\ M) \mid (\textbf{throw}) \mid V \mid x \mid \\
 & & (\textbf{CheckTermination}) \\
V & = & c \in \mathbb{N} \mid \mathit{blocks} \\
f, x & = & \text{identifiers}
\end{array}
$$

Fig. 7. An extended language for analysis, distinguishing codelets from system code, and identifying blocking functions.

$$
\begin{array}{rcl}
E & = & [] \mid (f\ E) \mid (\textbf{if}_0\ E\ M\ M) \mid (\textbf{let}\ \ (x\ E)\ M) \mid (\textbf{try}\ \ E\ M) \mid \\
 & & (\textbf{CheckTermination}) \mid (\textbf{throw}) \mid x \\
\text{final states} & = & V \mid \textbf{error} \mid (\textbf{throw})
\end{array}
$$

Fig. 8. Evaluation context for reduction of this extended language.

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

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

$$
\begin{array}{rcll}
E[(\textbf{CheckTermination})] & \mapsto & E[1] \text{ or } E[0] & \text{(depending on external conditions)} \\
E[(\textbf{if}_0\ 0\ M_1\ M_2)] & \mapsto & E[M_1] & \\
E[(\textbf{if}_0\ \mathit{blocks}\ M_1\ M_2)] & \mapsto & \textbf{error} & \\
E[(\textbf{if}_0\ V\ M_1\ M_2)] & \mapsto & E[M_2] & \text{if } V \neq 0, \mathit{blocks} \\
E[(\textbf{let}\ \ (x\ V)\ M)] & \mapsto & E[[V\ /\ x]\ M]^{\dagger} & \\
E[(\textbf{try}\ \ V\ M)] & \mapsto & E[V] & \\
E[(\textbf{try}\ \ (\textbf{throw})\ M)] & \mapsto & E[M] & \\
E[(f\ (\textbf{throw}))] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{if}_0\ (\textbf{throw})\ M_1\ M_2)] & \mapsto & E[(\textbf{throw})] & \\
E[(\textbf{let}\ \ (x\ (\textbf{throw}))\ M)] & \mapsto & E[(\textbf{throw})] & \\
E[x] & \mapsto & \textbf{error} & \text{(unbound variables)} \\
\delta(\mathit{blocks}?, v) & = & \begin{cases} 1 & \\ 0 & \end{cases} & \begin{array}{l} \text{if } v = \mathit{blocks} \\ \text{otherwise} \end{array}
\end{array}
$$

Fig. 9. Operation semantics for the extended language described in figure 7. This is largely a superset of the semantics of the original language, described in figure 3. †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. This is defined in figure 4.

(1)	$\mathcal{X}_2[\![\Gamma\ M]\!]$	$=$	$\mathcal{X}_2[\![\Gamma]\!]\ \mathcal{X}_2[\![M]\!]$
(2)	$\mathcal{X}_2[\![D_1\ \ldots\ D_n]\!]$	$=$	$\mathcal{X}_2[\![D_1]\!]\ \ldots \mathcal{X}_2[\![D_n]\!]$
(3a)	$\mathcal{X}_2[\![(\textbf{define codelet}\ (f\ x)\ M)]\!]$	$=$	$(\textbf{define codelet}\ (f\ x)\ \mathcal{X}_2[\![M]\!])$
(3b)	$\mathcal{X}_2[\![(\textbf{define system}\ (f\ x)\ M)]\!]$	$=$	$(\textbf{define system}\ (f\ x)\ \mathcal{X}_2[\![M]\!])$
(3c)	$\mathcal{X}_2[\![(\textbf{define blocking}\ (f_{blocking}\ x)\ f_{nonblocking})]\!]$	$=$	$(\textbf{define system}\ (f_{wrapper}\ x)$ $(\textbf{let}\ (t\ (f_{nonblocking}\ x))\ (\textbf{if}_0\ (blocks?\ t)\ t$ $(\textbf{if}_0\ (\textbf{CheckTermination})\ (f_{wrapper}\ x)\ (\textbf{throw})))))$ $(\textbf{define blocking}\ (f_{blocking}\ x)\ f_{nonblocking})$ Where the identifier t occurs nowhere else in the program.
(3d)	$\mathcal{X}_2[\![(\textbf{define primitive}\ (f\ x))]\!]$	$=$	$(\textbf{define primitive}\ (f\ x))$
(4)	$\mathcal{X}_2[\![(f\ M)]\!]$	$=$	$(\textbf{let}\ (t\ \mathcal{X}_2[\![M]\!])\ (\textbf{if}_0\ (\textbf{CheckTermination})\ (\mathcal{X}_2[\![f]\!]\ t)\ (\textbf{throw})))$ if $(\textbf{define codelet}\ (f\ x)\ M) \in \Gamma$ Where the identifier t occurs nowhere else in the program. $(\mathcal{X}_2[\![f]\!]\ \mathcal{X}_2[\![M]\!])$ *Otherwise*
(5)	$\mathcal{X}_2[\![f]\!]$	$=$	$f_{wrapper}$ if $(\textbf{define blocking}\ (f\ x)\ f_{nonblocking}) \in \Gamma$ (causing the definition of $f_{wrapper}$ via rule $3c$ above) f *Otherwise*
(6)	$\mathcal{X}_2[\![(\textbf{if}_0\ M_1\ M_2\ M_3)]\!]$	$=$	$(\textbf{if}_0\ \mathcal{X}_2[\![M_1]\!]\ \mathcal{X}_2[\![M_2]\!]\ \mathcal{X}_2[\![M_3]\!])$
(7)	$\mathcal{X}_2[\![(\textbf{try}\ M_1\ M_2)]\!]$	$=$	$(\textbf{try}\ \mathcal{X}_2[\![M_1]\!]\ \mathcal{X}_2[\![M_2]\!])$
(8)	$\mathcal{X}_2[\![(\textbf{let}\ (x\ M_1)\ M_2)]\!]$	$=$	$(\textbf{let}\ (x\ \mathcal{X}_2[\![M_1]\!])\ \mathcal{X}_2[\![M_2]\!])$
(9)	$\mathcal{X}_2[\![(\textbf{throw})]\!]$	$=$	$(\textbf{throw})$
(10)	$\mathcal{X}_2[\![(\textbf{CheckTermination})]\!]$	$=$	$(\textbf{CheckTermination})$
(11)	$\mathcal{X}_2[\![V]\!]$	$=$	V
(12)	$\mathcal{X}_2[\![x]\!]$	$=$	x

Fig. 10. The soft termination transformation for this extended language.

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

3.4.2 *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 10, $\mathcal{X}_2$, to distinguish it from the $\mathcal{X}$ transformation in figure 5.

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 non-blocking function and checks to see if termina-

Suppose for all declarations

$$(\textbf{define system}\ (f_{system}\ x)\ M) \in \Gamma$$

for every application $(f_{codelet}\ V_0)$ and $(f_{blocking}\ V_1)$ in M, where

$$(\textbf{define codelet}\ (f_{codelet}\ x)\ M_2) \in \Gamma$$
$$(\textbf{define blocking}\ (f_{blocking}\ x)\ f_{nonblocking}) \in \Gamma$$

there exists an integer $c > 0$ such that

$$E[(f_{codelet}\ V_0)] \mapsto^{c} E[V_0'] \mid E[(\textbf{throw})]$$
$$E[(f_{blocking}\ V_1)] \mapsto^{c} E[V_1'] \mid E[(\textbf{throw})]$$

for values V_0', V_1'. Then there is an integer $c_2 > 0$ such that

$$E[(f_{system}\ V)] \mapsto^{c_2} E[V'] \mid E[(\textbf{throw})]$$

for every **system** function f_{system}, for some value V'.

Fig. 11. The safety property of system code.

tion has been indicated. When the non-blocking 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 $f_{wrapper}$ to refer to such a wrapper function generated for some blocking function $f_{blocking}$. Note that different programming languages choose different abstractions for managing blocking I/O primitives. In section 4.3, 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 3.1. Recall from section 3.2 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 **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 non-blocking wrapper function. This guarantees that no blocking function is ever applied in the transformed program, simplifying the termination proof. The remainder of the rules describe how the transformation continues recursively on expressions.

3.4.3 *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 (**CheckTermination**) is 1. This is complicated by the fact that **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 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 formally in figure 11.

The safety property states that, if every **codelet** and **blocking** function application in any **system** function reduces to a value or to (**throw**) in a finite number of steps, then every **system** function reduces to a value or to (**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 **system** functions guarantee it for their code. Recall from section 3.4.1 that untrusted functions are never declared **system**.

By this property, system code will always terminate unless it applies a **codelet** or **blocking** function which 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 **system** function fails to return.

3.4.4 *Proof of termination.* In order to prove termination, we start with the grammar M of all possible expressions from figure 7. After transforming M by $\mathcal{X}_2$, we call the resulting set of expressions M_{lock_2} to distinguish from M_{lock} defined in section 3.3.4; like M_{lock}, M_{lock_2} is a subset of M. Assuming (**CheckTermination**) is 1, we wish to show the program enters a "locked" state, where termination is guaranteed.

$$
\begin{array}{rcll}
M_{lock_2} & = & (\mathbf{if}_0\ (\mathbf{CheckTermination}) & \text{if } (\mathbf{define\ \ codelet}\ (f\ x)\ M) \in \Gamma \quad (a) \\
 & & \quad (f\ V)\ (\mathbf{throw})\ )\ | & \\
 & & (\mathbf{if}_0\ 1\ (f\ V)\ (\mathbf{throw})\ )\ | & \text{if } (\mathbf{define\ \ codelet}\ (f\ x)\ M) \in \Gamma \quad (b) \\
 & & (f_{nonblocking}\ V)\ | & \text{if } (\mathbf{define\ \ blocking}\ (f_{blocking}\ x) \quad (c) \\
 & & & \quad f_{nonblocking}) \in \Gamma \\
 & & (f\ V)\ | & \text{if } (\mathbf{define\ \ primitive}\ (f\ x)) \in \Gamma \quad (d) \\
 & & (f\ V)\ | & \text{if } (\mathbf{define\ \ system}\ (f\ x)\ M) \in \Gamma \quad (e) \\
 & & (\mathbf{CheckTermination})\ | & (f) \\
 & & (\mathbf{if}_0\ M_{lock_2}\ M_{lock_2}\ M_{lock_2})\ | & (g) \\
 & & (\mathbf{let}\ \ (x\ M_{lock_2})\ M_{lock_2})\ | & (h) \\
 & & (\mathbf{try}\ \ M_{lock_2}\ M_{lock_2})\ | & (i) \\
 & & (\mathbf{throw})\ | & (j) \\
 & & x\ | & (k) \\
 & & V & (l)
\end{array}
$$

To prove termination, we must show that, as long as (**CheckTermination**) is 1, M_{lock_2} is closed under program stepping and that the syntactic length of the program is decreasing. Closure is defined in detail in section 3.3.2. By inspection, for all possible expressions in M_{lock_2}, we observe that our semantics preserve closure.

The syntactic length property may be stated as follows: If (**CheckTermination**) is always 1, then for any $M \in M_{lock_2}$, there is some integer $c > 0$ and $M' \in M_{lock_2}$ such that $M \mapsto^c M'$ and $|M'| < |M|$. That is, for any program in M_{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–l all make a program strictly smaller. Likewise, applications of primitive operations and non-blocking 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 $\mathcal{X}_2$ 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**). Note that if a program is being evaluated and (**CheckTermination**) is not guaranteed to be 1, a program may not be in M_{lock_2}. The case where this matters is when termination is requested just before a codelet function application; in this case, the next step taken will be the application itself, which will increase the length of the program. However, this resulting state must necessarily be in M_{lock_2}. In other words, a transformed program which is not in M_{lock_2} is guaranteed to enter M_{lock_2} in one step.

The final case is expression e, 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, $f_{wrapper}$ (that is, functions created by the $\mathcal{X}_2$ transformation applied to a **blocking** function declaration). The resulting expression is in M_{lock_2}, and all applications of system functions from within $f_{wrapper}$ are fenced by a termination check. As a result, as long as (**CheckTermination**) is 1, applications of $f_{wrapper}$ 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 3.4.3. 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 $(f_{codelet}\ V)$, for some **codelet** function $f_{codelet}$, the transformed function reduces $(\mathbf{if}_0(\mathbf{CheckTermination})\ (f_{codelet}\ V)\ (\mathbf{throw}))$. Wherever the original **system** function reduces $(f_{blocking}\ V)$ for some **blocking** function $f_{blocking}$, the transformed function reduces $(f_{wrapper}\ V)$, where $f_{wrapper}$ is the wrapper function which corresponds to $f_{blocking}$.

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

$$E[(f_{wrapper}\ V)] \mapsto^c E[V_0] \mid E[(\mathbf{throw})]$$

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

$$E[(\mathbf{if}_0\ (\mathbf{CheckTermination})\ \ldots\ (\mathbf{throw}))] \mapsto^c E[(\mathbf{throw})]$$

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.

So as long as (**CheckTermination**) is 1, every expression in M_{lock_2} reduces to some smaller expression in a finite number of steps. Since program lengths are finite, a program transformed by $\mathcal{X}_2$ is guaranteed to terminate in a finite number of steps, once the (**CheckTermination**) condition is raised, given the safety constraint on system code stated in section 3.4.3. ■

4. 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 which 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 12, 13, and 14. While 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.

4.1 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 [Cohen et al. 1998] packages, which provide interfaces for parsing and manipulating Java class files.

The basic structure of our bytecode modification is exactly as described in section 3.3.2. A static boolean field, initially set to false, is added to every Java class. The CheckTermination operation, implemented in-line, 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 10, we allow threads and thread groups to be terminated as well as specific 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 runtime overhead is only the cost of loading and checking the value, and then branching forward to the remainder of the method body.

Figure 12 shows the how the soft termination transform would be applied to a Java method declaration.

4.2 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 which targets itself. The effect of transforming a method with loops is shown in figure 13.

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 without any method invocation or backward branching. While such code should most likely be rejected by the Java bytecode verifier, as it is not allowed in the JVM specification [Lindholm and Yellin

[3] `http://www.alphaworks.ibm.com/tech/cfparse`

```
void foo() {
    ...
}
```

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

Fig. 12. The soft termination transformation applied to any function definition. See section 4.1 for a description of this transformation. This example takes into account the optimizations discussed in section 4.5.

```
void foo() {
    ...
    while (...) {
        ...
    }
    ...
}
```

```
void foo() {
    if (termination_signal) {
        termination_handler();
    }
    ...
    while (...) {
        ...
        if (termination_signal) {
            termination_handler();
        }
    }
    ...
}
```

Fig. 13. The soft termination transformation applied to any loop. See section 4.2 for a description of this transformation.

```
void foo() {
    ...
    blocking_bar(...);
    ...
}
```

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

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

Fig. 14. 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 4.3 for a description of this transformation.

1996], the bytecode verifier currently 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 which may be consumed at most once. The verifier's intent is to ensure that these instructions 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.

4.3 Blocking calls

To address blocking calls, we wish to follow the $\mathcal{X}_2$ transformation outlined in section 3.4.2. 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.

While using a polling model for terminating blocking calls simplifies analysis, 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.InterruptedException` or `java.io.InterruptedIOException` exception.

As in the $\mathcal{X}_2$ transformation described in section 3.4.2, we still wrap blocking function calls; and like $\mathcal{X}_2$, the wrapper returns if either data is returned or termination is signalled. However, instead of polling a non-blocking 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 [Wallach et al. 2000; Gong 1999].

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 `doPrivileged()` to mark regions of privileged system code and `getContext()` to get dynamic traces for making access control checks. These regions are exactly the same regions where preserving system integrity upon termination is important; if system code is using its own security privileges, it wants the operation to succeed regardless of its caller's privileges. Thus, we overload the semantics of these existing security primitives to include whether blocking calls should be interrupted.

The effect of this transformation on Java source is shown in figure 14.

4.4 Invoking termination

Our system supports three kinds of 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 4.3). At this point, the thread requesting termination performs a `Thread.join()` on the target thread(s), 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 signalled 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 which 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 shown in section 3.4.4, 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.

4.5 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 that we performed a termination check before calling now instead begins with a termination check, so the resulting program will behave the same. The effect of transforming, with this optimization, any Java function is shown in figure 12.

Additionally, 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 which 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.

4.6 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 [Gosling et al. 1996; Lindholm and Yellin 1996]. 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.

Additionally, it is 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.

4.7 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. This would defeat soft termination.

4.8 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 3.4.3. 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.

4.9 Memory consistency models

The Java language defines a relaxed consistency model where updates need not be propagated without the use 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 `volatile` modifier. This modifier is provided to guarantee that changes must be propagated immediately [Gosling et al. 1996]. On the benchmark platform we used, the performance impact of using `volatile` versus not using it is negligible. However, on other platforms, especially multiprocessing systems, this may not be the case.

4.10 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 *false*, either directly or through Java's reflection interface). Such protection could be added as a verification step before the bytecode rewriting.

5. 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 runtime, 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 macrobenchmarks which represent a number of 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 with the leaf method optimization discussed in section 4.5. Generally, when we discuss results in this section, we refer to the optimized numbers because these reflect the performance of a production soft termination system.

5.1 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 University of California, San Diego, and modified at University of Arizona for the Sumatra Project[4]. The results are shown in

[4]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`

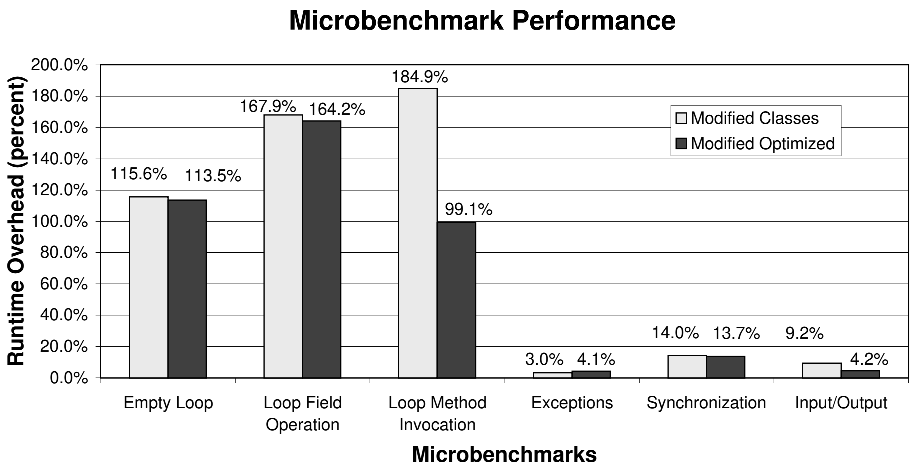

Fig. 15. Performance of rewritten microbenchmark class files relative to the performance of the corresponding original class files.

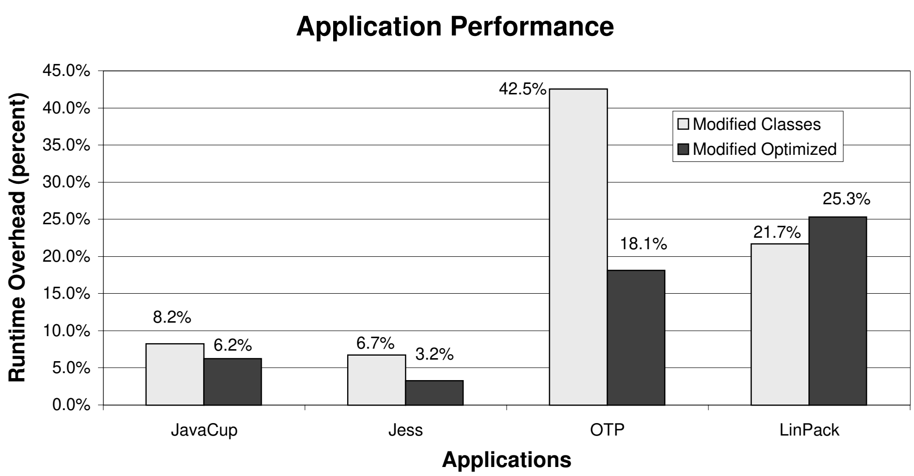

Fig. 16. Performance of rewritten application class files relative to the performance of the corresponding original class files.

figure 15.

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's 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. The additional overhead can be attributed to performing the termination check once for each iteration of the loop.

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 don't 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 macrobenchmark. The optimized programs are genuinely performing fewer termination checks, but still have longer runtimes. The culprit appears to be Sun's JIT compiler (sunwjit). When the benchmarks are run with the JIT disabled, the optimized programs are strictly faster than the non-optimized 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.

5.2 Application benchmarks

We benchmarked the real-world applications JavaCup[5], Linpack[6], Jess[7], and JOTP[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. JOTP is a one-time password generator which uses a cryptographic hash function. The results are shown in figure 16.

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 Jess through the system, and calculated the cumulative runtimes. This program, when rewritten, ran 3% slower. Both JavaCup and Jess represent applications which 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 runtime. For the Linpack benchmark, there was a 25% increase in runtime. Linpack is a loop-intensive program, while JOTP 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 JOTP got from the leaf method optimization.

---

[5]`http://www.cs.princeton.edu/~appel/modern/java/CUP/`
[6]`http://netlib2.cs.utk.edu/benchmark/linpackjava/`
[7]`http://herzberg1.ca.sandia.gov/jess/`
[8]`http://www.cs.umd.edu/~harry/jotp/`

Microbenchmarks			Application Benchmarks		
Microbenchmark	*Termination Checks*		*Application Benchmark*	*Termination Checks*	
	Unoptimized	*Optimized*		*Unoptimized*	*Optimized*
Empty Loop	$10,000,043$	$10,000,002$	JavaCup	$4,592,965$	$2,668,403$
Loop Field Operation	$55,000,257$	$55,000,012$	Jess	$992,765$	$720,499$
Loop Method Invocation	$60,000,117$	$30,000,006$	JOTP	$37,400,294$	$11,200,155$
Exceptions	$21,000,118$	$15,000,008$	Linpack	$1,266,960$	$1,214,477$
Synchronization	$60,000,115$	$40,000,008$			
Input/Output	$200,958$	$200,006$			

Fig. 17. Average total number of termination checks performed for each benchmark.

5.3 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 figure 17.

All of the microbenchmarks performed one termination check per iteration, with few additional overhead checks. This is exactly as expected. These results translate to roughly $40,000,000$ checks performed for every second of runtime overhead for all but the input/output microbenchmark. This evaluates to around 10 CPU cycles 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 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. While 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 which have tight loops may experience at worst a factor of two slowdown and more commonly $15-25\%$. Applications without tight loops can expect more modest slowdowns, 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.

6. SOFT TERMINATION IN PRACTICE

A number of attacks against Java focusing on resource exhaustion have been proposed [McGraw and Felten 1999]. Several of these focus on flaws in Java's access control framework. For example, the standard recipe for designing such attack includes setting a thread's priority to `MAX_PRIORITY` to help ensure the program does its job. Java specifies an access control privilege for changing thread priority. The more serious *Business Assassin* applet relies on Java allowing unprivileged threads to stop one another. Our soft termination system does not try to stop these attacks.

A number of other resource exhaustion attacks do not take advantage of flaws in Java's security system. These include creating threads which loop infinitely, overriding the `Applet.stop()` method, or catching a `ThreadDeath` exception and recreating the thread. As mentioned in section 4.4, soft termination successfully stops such applets.

7. FUTURE WORK

Now that there is a mechanism for guaranteed termination of user-level code, the obvious extension is to establish policy for terminating code. We plan to extend this project to the realm of resource control. Termination could be conditioned on the exhaustion of such system resources as memory, CPU, or network bandwidth.

Another area for future work is in extending this system to allow for the restart of user code. Currently, any state maintained by a user codelet can be rendered inconsistent by an untimely termination request. Some mechanism for rolling the codelet's state back to a consistent state is necessary for the safe restart of user codelets. To achieve this, we plan to build a transaction-style manager around memory [Printezis et al. 1997; Daynès and Czajkowski 2001].

8. CONCLUSION

While 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.

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 runtime system. While 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 slowdowns of $3-25\%$. This could possibly be further reduced if we could leverage a safe point mechanism already implemented within the JVM.

A larger research area remains: building language runtimes that support the general process-management semantics of operating systems. Because language runtimes allow and take advantage of threads and memory references that easily cross protection boundaries, traditional operating system processes may not be appropriate in this new setting. Opportunities exist to design new mechanisms to add these semantics to programming language runtimes.

ACKNOWLEDGMENTS

John Clements made significant contribution to earlier versions of this paper. Jiangchun "Frank" Luo and Liwei Peng helped implement an early prototype of this system. Matthias

Felleisen and Shriram Krishnamurthy provided many helpful comments, as did Natarajan Shankar and Drew Dean. Comments by the anonymous NDSS and TISSEC reviewers were also very insightful.

This work is supported by NSF Grant CCR-9985332.

REFERENCES

ALVES-FOSS, J. Ed. 1999. *Formal Syntax and Semantics of Java*. Number 1523 in Lecture Notes in Computer Science. Springer-Verlag.

BACK, G. AND HSIEH, W. 1999. Drawing the Red Line in Java. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems* (Rio Rico, Arizona, March 1999).

BACK, G., HSIEH, W. C., AND LEPREAU, J. 2000. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, California, Oct. 2000).

BACK, G., TULLMANN, P., STOLLER, L., HSIEH, W. C., AND LEPREAU, J. 2000. Techniques for the design of Java Operating System. In *Proceedings of the 2000 Usenix Annual Technical Conference* (San Diego, California, June 2000).

BERNADAT, P., LAMBRIGHT, D., AND TRAVOSTINO, F. 1998. Towards a resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, Dec. 1998).

BINDER, W. 2001. Design and implementation of the J-SEAL2 mobile agent kernel. In *2001 Symposium on Applications and the Internet* (San Diego, CA, USA, Jan. 2001).

BROMLEY, H. 1986. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers.

Burroughs Corporation. 1969. *Burroughs B6500 Information Processing Systems Reference Manual*. Detroit, Michigan: Burroughs Corporation.

CHANDER, A., MITCHELL, J. C., AND SHIN, I. 2001. Mobile code security by Java bytecode instrumentation. In *2001 DARPA Information Survivability Conference & Exposition (DISCEX II)* (Anaheim, CA, USA, June 2001).

COGLIO, A. AND GOLDBERG, A. 2000. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. In *2nd ECOOP Workshop on Formal Techniques for Java Programs* (Sophia Antipolis and Cannes, France, June 2000).

COHEN, G., CHASE, J., AND KAMINSKY, D. 1998. Automatic program transformation with JOIE. In *Proceedings of the 1998 Usenix Annual Technical Symposium* (New Orleans, Louisiana, June 1998), pp. 167–178.

CZAJKOWSKI, G. AND DAYNÈS, L. 2001. Multi-tasking without compromise: a virtual machine approach. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications* (Tampa Bay, Florida, Oct. 2001).

CZAJKOWSKI, G. AND VON EICKEN, T. 1998. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 1998), pp. 21–35.

DAYNÈS, L. AND CZAJKOWSKI, G. 2001. High-performance, space-efficient, automated object locking. In *Seventeenth International Conference on Data Engineering* (Heidelberg, Germany, April 2001).

DEAN, D. 1997. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security* (Zurich, Switzerland, April 1997).

DEAN, D., FELTEN, E. W., WALLACH, D. S., AND BALFANZ, D. 1997. Java security: Web browsers and beyond. In D. E. DENNING AND P. J. DENNING Eds., *Internet Besieged: Countering Cyberspace Scofflaws*, pp. 241–269. New York, New York: ACM Press.

DROSSOPOULOU, S. AND EISENBACH, S. 1997. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)* (Jyväskylä, Finland, June 1997).

DROSSOPOULOU, S., WRAGG, D., AND EISENBACH, S. 1998. What *is* Java binary compatibility? In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Oct. 1998), pp. 341–358.

EDJLALI, G., ACHARYA, A., AND CHAUDHARY, V. 1998. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS '98)* (San Francisco, California, Nov. 1998), pp. 38–48. ACM Press.

ERLINGSSON, U. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop* (Caledon Hills, Ontario, Canada, Sept. 1999). ACM Press.

FEELEY, M. 1993. Polling efficiently on stock hardware. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993).

FELLEISEN, M. AND HIEB, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 102*, 235–271.

FLATT, M., FINDLER, R. B., KRISHNAMURTHY, S., AND FELLEISEN, M. 1999. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP '99)* (Paris, France, Sept. 1999).

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts.

GONG, L. 1999. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts.

HAWBLITZEL, C., CHANG, C.-C., CZAJKOWSKI, G., HU, D., AND VON EICKEN, T. 1998. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, June 1998). USENIX.

HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. 1998. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages* (1998), pp. 86–93. ACM.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts.

MALKHI, D., REITER, M., AND RUBIN, A. 1998. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, California, May 1998), pp. 40–51.

MCGRAW, G. AND FELTEN, E. W. 1999. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, New York, New York.

NILSEN, K., MITRA, S., SANKARANARAYANAN, S., AND THANUVAN, V. 1998. Asynchronous Java exception handling in a real-time context. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain, Dec. 1998). NewMonics, Inc.

PRINTEZIS, T., ATKINSON, M. P., DAYNÈS, L., SPENCE, S., AND BAILEY, P. 1997. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)* (Half Moon Bay, CA, USA, 1997).

REDELL, D., DALAL, Y., HORSLEY, T., LAUER, H., LYNCH, W., MCJONES, P., MURRAY, H., AND PURCELL, S. 1980. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2 (Feb.), 81–92.

SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. 1999. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles* (Kiawah Island Resort, South Carolina, Dec. 1999), pp. 202–216. ACM.

STATA, R. AND ABADI, M. 1998. A type system for Java bytecode subroutines. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages* (Jan. 1998), pp. 149–160. ACM.

Sun Microsystems. 1990. *ptrace(2) Manual Page*. Palo Alto, CA: Sun Microsystems.

SWINEHART, D. C., ZELLWEGER, P. T., BEACH, R. J., AND HAGMANN, R. B. 1986. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (Oct.), 419–490.

TULLMAN, P. AND LEPREAU, J. 1998. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop* (Sept. 1998).

VAN DOORN, L. 2000. A secure Java virtual machine. In *Ninth USENIX Security Symposium Proceedings* (Denver, Colorado, Aug. 2000).

WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. 1997. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles* (Saint-Malo, France, Oct. 1997), pp. 116–128.

WALLACH, D. S., FELTEN, E. W., AND APPEL, A. W. 2000. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology 9*, 4 (Oct.), 341–378.

WIRTH, N. AND GUTKNECHT, J. 1992. *Project Oberon*. ACM Press.