Technical Report

UCAM-CL-TR-525
ISSN 1476-2986

Number 525

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Extensible virtual machines

Timothy L. Harris

December 2001

JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted by the author for the degree of Doctor of Philosophy to the University of Cambridge.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

Virtual machines (VMs) have enjoyed a resurgence as a way of allowing the same application program to be used across a range of computer systems. This flexibility comes from the abstraction that the VM provides over the native interface of a particular computer. However, this also means that the application is prevented from taking the features of particular physical machines into account in its implementation.

This dissertation addresses the question of why, where and how it is useful, possible and practicable to provide an application with access to lower-level interfaces. It argues that many aspects of VM implementation can be devolved safely to untrusted applications and demonstrates this through a prototype which allows control over run-time compilation, object placement within the heap and thread scheduling. The proposed architecture separates these application-specific policy implementations from the application itself. This allows one application to be used with different policies on different systems and also allows naïve or premature optimizations to be removed.

Acknowledgements

I would like to thank my supervisor, Simon Crosby, for his enthusiasm and advice when in the department and his introduction to self-directed research when away; he was like a mentor to me.

I have been very fortunate to be in the company of many excellent people who have contributed to the completion of this work, either directly through their expertise or indirectly though their tolerance of my rantings. I am indebted to Paul Barham, Austin Donnelly, Keir Fraser, Steven Hand, Adrian Joyce, Paul Menage, Richard Mortier, Dickon Reed and Richard Stamp for proof-reading this dissertation: any residual mistakes are down to me. For distracting me from this work with many more interesting things I would also like to thank Steve Heller and the members of the Java Technology Research Group at Sun Labs.

Finally, I would particularly like to thank Andrew Herbert for his support throughout my PhD: in securing funding, arranging numerous conference visits and providing much helpful discussion and feedback on this work he has shown me that managers need not have pointy hair.

Contents

Chapter 1

Introduction

There is a trend in computer science towards implementing software over a *virtual machine*. Such a machine provides the familiar functions and services expected by computer programs but does so using software rather than hardware. The machine *virtualizes* the instruction set, memory and other resources of the underlying physical machine, thereby presenting application programmers with a standard interface.

The primary motivation for using a virtual machine is that it decouples the design of computer software from the evolution and diversity of computer hardware and operating systems. This is because the same application code can be used on any system that supports the appropriate virtual machine. Advocates of virtual machines say that it is far more convenient to implement a single virtual machine for each computer than it would be to re-implement or even just recompile each application for every different system. Furthermore, a virtual machine that allows one program to be used on different computers also aids mobility: allowing programs to move seamlessly between computers, perhaps to follow the user to a different physical location, or to relocate from a busy computer to an idle one.

This level of portability becomes particularly important in web-based applications in which components of the system may be deployed on remote clients' machines as downloadable code executed within a web browser.

Unfortunately, the facilities provided by a portable virtual machine are not without their drawbacks. In fact, these problems stem from the very flexibility that gives virtual machines their advantages. A virtual machine that provides a uniform programming environment over a range of processor types cannot provide access to special facilities that are only available on some of them. Similarly, some form of software-based dispatching and execution is required because the instruction format of the virtual machine is different from the native format of the microprocessor. This introduces complexity into the virtual machine and the time taken for translation can harm performance.

This dissertation addresses the question of how to reconcile these advantages and disadvantages. In particular, it concentrates on how the resources of the underlying computer can be presented more directly to applications without sacrificing the portability of programs between systems.

By way of introduction, consider the problem of translating a program from the instruction set of the virtual machine (VM) to the instruction set of a physical machine. In a modern VM it is common for both a compiler and an interpreter to be available and for some portions of the program to be compiled to native code during the operation of the VM while other portions are only ever executed by the interpreter. A typical policy is to interpret most of the program and to compile any section that is observed to execute frequently. This balances the poor performance experienced when interpreting against the delays introduced as the compiler operates. Since many programs contain small hot-spots that account for most of their execution time it makes sense to try to identify these and to spend time compiling only them.

However, this general-purpose policy does not work well for all kinds of application. Consider one that decodes and plays digital audio data. In this case the hot-spot may be the decoding algorithm. However, if the VM starts compiling this part of the program while it is running then the delay introduced may cause an audible glitch in the sound. Furthermore the initial performance of the application – before the compiler is invoked – may be unacceptably poor.

In contrast, the VM architecture presented in this dissertation allows application-specific policies to be used. These are defined in a general-purpose programming language and can consequently perform arbitrary computation – for example to evaluate and to adapt to the rate of progress of the system. However, the logical separation between the two permits the same application to be used with different policies (perhaps when it is used on different systems) and also allows one policy to be reused with multiple applications (perhaps a default policy suitable for general-purpose programs).

The work presented in this dissertation defines a common framework within which policy definitions may be used to control the various resources that an application uses through a VM. Aside from the previous example of a run-time compilation service, the placement of objects within memory and the scheduling of threads within the VM are taken as examples. In each case safe interfaces, accessible by an untrusted application programmer, have been designed and implemented.

My thesis is that applications can benefit through such control of resources and services in three ways. Firstly, the speed with which some applications execute is improved – the facilities provided, particularly those concerned with the placement of objects within memory, are similar to those exploited by programmers in lower-level languages to tune the performance of applications. Secondly, even if a program does not complete more quickly, it may be possible to adjust its performance so that it is preferable to users – for example by scheduling run-time compilation or garbage

collection work to occur less intrusively. Finally, many of the proposed interfaces can also be exploited to trace system behaviour, aiding debugging and profiling.

The dissertation is organized as follows:

Chapter 2 describes background work relating the historical development of VMs and the more recent development of extensible operating systems. The former clarifies the kind of environment within which the work described here is intended to be employed. The relevance of the latter comes from the analogy between moving processing from the kernel of an operating system into untrusted user-space code and moving VM-internal facilities into application code.

Chapter 3 surveys related work. It describes existing projects that exploit the flexibility in conventional VMs and identifies areas in which the limited extent of that flexibility presents a barrier to the use of a VM as a ubiquitous execution environment.

Chapter 4 presents the overall design for an eXtensible Virtual Machine (XVM) supporting application-specific policies for resource management. A common framework is developed within which run-time compilation, memory allocation and thread scheduling policies are taken as examples. In outline, policies are defined in a general-purpose programming language and are implemented by making invocations on *protected mechanism implementations* that are provided by the VM. A *policy registry* records the association between policies and sections of the application.

Chapter 5 describes the implementation of this infrastructure over the Java Virtual Machine (JVM) as a mechanism for defining application-specific policies to control run-time compilation. As with the initial example given in this introduction, the primary purpose of such a policy is to define those parts of the application which should be compiled, when that compilation occurs and what kinds of optimization are attempted. An early version of the work on controllable run-time compilation was presented at the IEEE Workshop on Programming Languages for Real-Time

Industrial Applications [Harris98].

Chapter 6 describes the corresponding implementation for supporting policies for application-specific memory allocation. In this case the policies may control where objects are placed within the heap – for example to cluster objects that are expected to be used together.

Chapter 7 shows the development of an application-level thread scheduling environment with which an untrusted program may define the way in which its threads are multiplexed over the CPUs available to the VM.

Finally, Chapter 8 concludes and presents ideas for future work.

1.1 Contributions

The primary contributions of this dissertation are (*i*) the development of a framework supporting safe application-accessible interfaces to control policy decisions in a safe execution environment and (*ii*) the realization of this framework for controlling run-time compilation, the placements of objects within the heap and the scheduling of threads.

In addition the infrastructure developed enables novel policy definitions to be defined. Particular examples include the *background compilation* policy (Section 5.6) and the thread scheduler using *allocation inheritance* (Section 7.6).

1.2 Terminology

Some of the terms in this dissertation have come to be used elsewhere with a variety of meanings. This section clarifies how they will be used here.

Virtual machines

The term *virtual machine* is used in this dissertation to refer to systems such as the Java Virtual Machine [Lindholm97] and Smalltalk VM [Goldberg83], typified by the use of a non-native instruction set, a substantial set of standard library functions and an implementation that exposes only high-level interfaces to the resources of the underlying system. Such systems effectively provide the complete set of services that an application programmer may expect from an operating system, including device abstractions and isolation between parts of the system. This is in contrast to systems based around *virtual machine monitors* [Creasy81] which export multiple virtualized instances of the same underlying native system, or simple interpreters which provide only an instruction set without library support.

Processes and applications

Much of the work described in this dissertation has been implemented in user-space over the Nemesis operating system, the salient features of which will be described in Section 2.2. Nemesis distinguishes between different concepts that are coincident in traditional environments such as UNIX. For example it separates the ideas of the *protection domain* (the principal for which access control is performed), *scheduling domain* (the entity to which resource allocations are made) and *activation domain* (the code that implements the application). However this distinction is not used in the work presented here and therefore the more familiar term *process* is used throughout.

The term *application* is reserved for the bodies of code that operate over the VM – for example a word processor or media player.

Classes and types

In an object-oriented language, *classes* are the things from which objects are instantiated. Each object is therefore an instance of exactly one class. In contrast, *types* are predicates over values, indicating those that are compatible with that type under the particular rules of the language.

For example, in the Java programming language, the `class` definition files that comprise an application can be identified with both classes and reference-types. An object instantiated from the root class `java.lang.Object` is an instance of only the type `java.lang.Object`. An object instantiated from the class `java.lang.Dictionary` is an instance of two types: `java.lang.Object` and `java.lang.Dictionary`. In particular, note that such an object is *not* an instance of the *class* `java.lang.Object`.

In general it should be clear from context where a name is used to refer to a class and where it is used to refer to a type.

Chapter 2

Background work

This chapter presents background information on the design of *virtual machines* (in Section 2.1) and *extensible operating systems* (in Section 2.2).

The emphasis in the first part of this chapter is on the evolution of VMs and on the features (and usage patterns) that typify current systems. Subsequent chapters will draw on that work when describing the rationale, design and implementation of an extensible virtual machine.

The second part of this chapter concerns *extensible operating systems* which, in a different context, share the goals of abstracting certain aspects of a computer system's native execution environment and providing isolation between different tasks that are running. This further area of background work is important to the current dissertation in two ways: the manner in which interfaces are designed to be safe for access by untrusted code, and more concretely in that the Nemesis operating system (Section 2.2.2) forms the implementation platform for much of the work in this dissertation.

2.1 Virtual machines

The individual systems selected for discussion in Sections 2.1.1-2.1.8 are ones used primarily for imperative and object-oriented languages. These hold most relevance to the ideas developed in subsequent chapters. In addition to these systems VMs have also been designed for functional languages and for logic programming languages. Examples include the AMOZ machine for Oz [Mehl95], the Brisk machine for Haskell [Holyer98], and the STG and Three Instruction machines [Chitnis95]. However, the facilities provided by those machines are typically at a higher level and more specialized to the language, or class of languages, in question.

In this chapter the machines are presented in approximately chronological order of development. The reality, of course, is not so precise with substantial overlap between the times at which these systems have been in use, if not between the times over which they were designed.

2.1.1 INTCODE

The INTCODE system is a simple intermediate code which lends itself to straightforward execution by an interpreter [Richards79]. Richards reports that it was used as part of the bootstrapping process for the BCPL programming language: a simple assembler and interpreter could be implemented in less than one week. This interpreter could be used to deploy an INTCODE version of the BCPL compiler rapidly without constructing a separate code generator for each target platform.

The INTCODE machine has six special-purpose registers: an accumulator, auxiliary accumulator, program counter, address register, stack pointer and global-vector pointer. Instructions are generated from eight basic operations: load, store, add, branch, branch-if-true, branch-if-false, procedure call and execute. They are encoded

in an orthogonal bitwise representation in which each instruction specifies some mechanism of computing an address and some operation to perform between the contents of that address and one of the machine registers. The design of the instruction set reflects then-contemporary accumulator architectures [Hennessy90].

2.1.2 Pascal-P

The Pascal programming language [Wirth71] was implemented on over sixty different kinds of computing system during the 1970s [Barron81]. The language enjoyed particular popularity among educators for its handling of data types and more generally as a 'language for structured programming' [Webster81, Barron81].

Among these implementations, the Pascal-P system is notable in that it was designed to operate over a simple machine-independent VM termed the Stack Computer (SC) [Nori81]. The Pascal-P distribution kit included an implementation of the Pascal-P compiler for use over the SC. As with the INTCODE bootstrap implementation of BCPL this meant that Pascal could be made available on a new system by merely implementing the SC. Nori *et al* suggest that such an implementation would suffice 'if the expected use of Pascal is for teaching purposes and only short programs are to be compiled and executed' [Nori81].

The SC provides facilities superficially similar to those of a conventional microprocessor operating in user mode. However, the instruction set includes separate operations for each of the basic Pascal data types such as characters, pointers, integers and booleans. This enables a SC implementation to select appropriate representations for each data type.

At run-time the memory of the SC is divided into separate regions containing the code, stack and heap. As suggested by the name of the machine, a stack is used

for temporary values during computation whereas the machine registers are used for holding only the program counter and managing the areas of memory. The instructions available on the SC are reasonably high-level – for example the format of the static and dynamic links between stack frames is prescribed by the implementation of the operations used to perform procedure calls. A special 'call standard procedure' operation is provided to invoke library facilities such as file access or trigonometric computation.

2.1.3 Smalltalk-80

Smalltalk was conceived towards the end of the 1970s as a new way that 'people might effectively and joyfully use computing power' [Goldberg83]. In Smalltalk-80 this is realized in a pure class-based object-oriented programming language [Krasner84] [1]. Unlike impure object-oriented languages (such as C++ [Stroustrup97], Modula-3[Nelson, editor91] or Java [Gosling97a]) *all* computation is notionally performed by dynamic method invocation or equivalently, in the terminology of Smalltalk, by *message send* operations. This includes simple arithmetic operations – such as integer addition – and also control flow operations on objects representing blocks of code and class definitions themselves. This approach allows common control flow structures within methods to be implemented as operations on blocks rather than as first-class language features.

Smalltalk has traditionally been implemented over a stack-based virtual machine coupled with a suite of standard libraries. These libraries include implementations of data structures such as *sets*, *sequences* and *dictionaries* as well as common interfaces to

[1] Smalltalk-80 succeeds five similarly-named systems developed at Xerox PARC. As the most recent, well-known and widely studied version, discussion is confined to that revision. Ingalls describes the prior evolution of the Smalltalk virtual machine and the corresponding development of the Smalltalk programming language [Ingalls84].

peripheral devices. The majority of these libraries would themselves be implemented in Smalltalk and compiled to the Smalltalk bytecode format. The exception is that around one hundred *primitive routines* are implemented directly within the virtual machine and handled as special cases during message send operations. These primitives implement simple arithmetic operations, object allocation, process control and input/output functions. The bytecode operations themselves are generally concerned with manipulating the run-time stack and performing message sends. The initial state of the virtual machine is obtained from a *virtual image* which describes the contents of the heap and the initial programs and processes available within the system. These may include compilers, loaders and development tools.

The programming language, the virtual image and the virtual machine allow portability at two different levels. Firstly, any Smalltalk program may be executed unmodified on any hardware platform supporting a conforming implementation of the VM. Secondly, existing Smalltalk virtual images may be loaded by different implementations of the virtual machine and primitive routines. For example Ingalls reports that substantial parts of the virtual image released with Smalltalk-80 correspond to parts of an original image cloned from Smalltalk-76 [Ingalls84].

Deutsch and Schiffman describe an implementation of the Smalltalk-80 system that is designed to be efficient on conventional hardware – that is, it should not rely on the availability of user-microprogrammable instructions [Deutsch84]. They describe the approach as using *dynamic change of representation* in which there may be multiple versions of the same data held concurrently in different formats. In particular the implementation of a program may exist in both bytecode and native forms. The Smalltalk bytecode language enforces encapsulation by only providing operations for accessing data fields within the current object, thereby allowing a native code generator to use different representations for different classes.

In the Deutsch-Schiffman implementation, method invocations are implemented using *inline caches* to reduce the frequency of dynamic look-ups and computed

branches. An inline cache system operates by assuming that the receiver object is of a particular class and branching directly to the appropriate implementation of the method. That method checks the actual class and, if it differs, backs off to perform a full method look-up. The chosen class to inline may be selected by using run-time feedback to identify common receiver classes for each call site – a simple feedback scheme for call sites with few dynamic targets is to update the inline cache after each full look-up.

The Smalltalk on a RISC (SOAR) implementation made further changes to the Smalltalk-80 system with the aim of improving performance on emerging reduced-instruction-set architectures. Unlike earlier implementations in which references identified slots in an *object table*, SOAR represented object references by the direct address of the per-instance field data. Such a representation was a more natural fit for the limited set of addressing modes present on RISC systems. The new structure avoids one level of indirection on most object-based operations but makes the implementation of the `becomes` bytecode operation [2] more complex: where previous Smalltalk systems could simply update the pointer in the object table, SOAR had to update each reference to the object. Consequently the SOAR system used re-written system classes using explicit indirection in place of such operations. The implementation recognized that most method invocations occurred with conventional last-in first-out (LIFO) ordering and so placed activation records directly on the processor stack, lazily moving to the heap any that are required by closures.

[2] The `becomes` operation is a curious one that is not available in mainstream object-oriented languages. Its effect is to replace one object with another, with the change concurrently updating all references to the first object. It can be used to provide the illusion of extensible arrays (by replacing a shorter array with a new longer one) and to model changes in the life cycle of modeled objects. The `becomes` operation is troublesome to implement without the indirection provided by an object table. Furthermore, its utility is diminished by the use of static typing and its applicability in multi-threaded environments is limited because of potential race-conditions between invocations by concurrent threads.

Meta-classes

In addition to its class-based object model, Smalltalk-80 also defines a relatively straightforward *meta-class system*. The relationship between meta-classes and ordinary classes is similar to the relationship between ordinary classes and their instances: a meta-class is simply a class whose instances are themselves classes. Within the Smalltalk VM a hierarchy of meta-classes is implicitly defined to mirror the ordinary class hierarchy. For example, if class `X` extends class `Y` then the meta-class `XClass` will extend the meta-class `YClass`. The class `Object` is the root of the hierarchy of ordinary classes. The meta-class `Class` is the root of the hierarchy of meta-classes. Meta-classes themselves are instances of `Meta-class` which is itself an instance of `Class`.

The meta-class system in Smalltalk-80 provides fairly limited functionality. The intent was that meta-classes would generally be concerned with instance initialization and with the representation of state that is associated with a class as a whole rather than with each instance [Goldberg83]. In the Java programming language these functions are provided by constructors, and `static` members of ordinary classes [Gosling97a].

Other languages, such as Common Lisp Object System (CLOS) [Bobrow88, Lawless91], allow explicit meta-classes to be defined, in contrast to the implicit definitions generated in Smalltalk-80. These languages dissociate the inheritance relationships between classes from those between meta-classes. This often permits meta-classes to be used to control more general aspects of class behaviour – such as the manner in which methods are invoked – because the same meta-class can be associated with a group of classes for which common behaviour is desired. A usual idiom is that the meta-class introduces pre-call and post-call operations around each method invocation, for example to produce traces of application behaviour, to introduce locking around invocations or to perform access control checks.

2.1.4 SELF

Unlike Smalltalk, SELF is a pure *prototype-based* object-oriented programming language [Chambers89]. This means that it uses an existing object, rather than a class definition, as the 'blueprint' when making an instantiation. For example the new object may obtain methods, fields and field values from the prototype object. A class-based style of programming is possible by creating separate prototype-objects for each intended class [3]. Being a *pure* object-oriented language, all data values are treated as objects and all operations on them are method invocations. As with Smalltalk, SELF takes an extreme approach to this by expressing intra-method control flow operations in terms of method invocations on anonymous code blocks.

The combination of dynamic typing and prototype-based inheritance apparently conspires to prevent inlining across abstraction boundaries: the target address of each method invocation must be computed based on the ancestry of the receiving object. Furthermore, the high frequency of method invocations in pure object-oriented programming makes it particularly important to provide an efficient implementation. This area has therefore been the primary focus of the published work on the SELF VM.

In a series of papers, Agesen, Chambers, Hölzle, and Ungar describe a number of mutually-beneficial techniques for the efficient implementation of SELF [Chambers91, Hölzle92, Hölzle94b, Hölzle94a, Agesen95]. *Type feedback* gathers statistics about the frequency of different targets at each method call site. *Adaptive optimization* identifies frequently executed methods and uses an optimizing compiler to translate their bytecode implementation to native code. A faster non-optimizing compiler is used for other methods to make the system responsive for interactive use. *Dynamic de-optimization* allows source-level debugging, even once code has been optimized, by recreating non-optimized code and using on-stack replacement of activation records

[3] Agesen uses such an idiom when implementing the operations of the JVM over the SELF VM. This work will be discussed in Section 3.1.3

to switch between the two forms. Finally, polymorphic in-line caches (PICs) of frequently executed methods at their call sites enable efficient method dispatch (for example, by avoiding a computed branch) and collect concrete type information for the optimizing compiler. Unlike an *inline cache*, a PIC records multiple target addresses.

It is important to note that these techniques were performed dynamically in the production VM. They therefore had not only to make judicious optimization decisions but the infrastructure had to be implemented efficiently so that the cost of the analyses and code transformations did not detract from the benefits achieved. However, this dynamic setting does enable the VM implementation to exploit differences between individual runs of an application in a way that is not possible with static or trace-driven analyses.

The techniques used in the SELF VM exploit various forms of locality seen to be exhibited. In particular, the distribution of concrete types encountered at many call sites remained fixed over the course of a program execution. As Hölzle reports, typical benchmark applications contained substantial numbers of *monomorphic* call sites at which only a single concrete type was even seen, fewer *polymorphic* call sites at which a few concrete types were seen and a small number of *megamorphic* call sites from which methods were invoked on many different kinds of object [Hölzle94a].

Hölzle and Ungar argue that the implementation of SELF invalidates earlier claims that object-oriented programming languages required special hardware or microcode support [Hölzle95]. They claim that the optimizations performed at run-time in the SELF VM enable the generation of code which is comparable with that generated from the C programming language. For example, concrete type analysis is effective at identifying where native arithmetic operations or formats may be used in lieu of an object-oriented implementation.

2.1.5 The Java Virtual Machine

The Java Virtual Machine [Lindholm97] is designed to host programs implemented in Java bytecode [Gosling95], an instruction set used for easy generation from the Java programming language [Gosling97a]. Java bytecode is an object-oriented stack-based encoding which was intended for both direct execution by an interpreter and for run-time compilation to native code. The JVM loads bytecode from `class` files – each of which contains the bytecode instructions and auxiliary information needed to define a single class. Loading may be triggered explicitly by the programmer through the *reflection* API [Gosling97b], or implicitly according to detailed rules set out in the VM specification [Lindholm97].

These rules also prescribe when per-class initialization code is executed and when errors are reported. This has important consequences for the JVM implementor:

1. Demand-loading classes means that it is not generally possible to analyze the complete code of a program in advance. This means that many run-time optimizations are only safe if dependency information is recorded and the optimization can be reversed if the underlying assumptions cease to be valid. For example a method may only be inlined at a call site while it remains the only possible target, or when a run-time check is performed.

2. Per-class initialization code must be executed at well-defined times, meaning that it is not possible to speculatively load classes through the normal class loading mechanism. This also creates problems for an optimizing compiler performing method inlining. Liang and Bracha describe the JVM class-loading mechanism in more detail [Liang98]. McDowell and Baldwin present the associated problem of unloading classes from a running instance of the VM [McDowell98].

There is little information lost when generating bytecode from a source file written in the Java programming language. The object model of the VM differs solely in that it identifies methods by return type in addition to name and argument types.

Aside from the object model, the other notable differences are that the JVM does not provide direct arithmetic operations on sub-word integer types, that Java bytecode provides an intra-method `goto` operation and that exception handlers are specified out-of-line in the `.class` file and need not be nested.

This low semantic gap allows rapid translation from Java source code to Java bytecode. It also permits reverse compilation in which, in most cases, a close analogue of a source file can be reconstructed from its `.class` file representation [Proebsting97]. Code obfuscation tools can attempt to reduce the quality of the reconstructed source file by introducing the features that are only expressible directly in Java bytecode [Collberg98].

The JVM allows safe execution of untrusted programs by *sandboxing*: limiting an application's access to resources that are not deemed *safe*. This is achieved through a number of techniques. Firstly, the Java bytecode is *verified* before execution. The verification process is defined informally as part of the JVM specification, although there have been many attempts to retrospectively fit formal definitions to both the behaviour of bytecode operations and to the operation of the bytecode verifier [Bertelsen98, Coglio98, Qian99]. Secondly, run-time checking occurs as part of many methods in the standard Java APIs. These checks are implemented as method invocations on an instance of `SecurityManager`. The security manager rejects access by throwing an exception.

Many flaws have been found in the specification and implementation of the bytecode verifier and trusted classes. Dean *et al* report implementation errors, unintended feature interactions and unexpected differences between the expected and implemented behaviour of Java bytecodes in early implementations of the JVM [Dean97].

The standard Java API allows multiple threads of execution to be created. These can interact, at a primitive level, by accessing shared fields using `putfield`, `getfield`, `putstatic` and `getstatic` bytecodes, by acquiring and releasing

mutual-exclusion locks using `monitorenter` and `monitorexit` bytecodes, by invoking `synchronized` methods, and by calling native `wait`, `notify` and `notifyAll` methods on an object locked by the calling thread.

A lock is conceptually associated with every object instantiated. This 'ubiquitous' synchronization enables simple locking strategies for enforcing mutual exclusion. The approach is typified by the reference implementation of the library classes `java.util.Hashtable` and `java.util.Vector` in which most methods must acquire a lock on the current instance.

Including synchronization in this form in standard libraries also means that locking operations are frequent. This is true even in applications that are apparently single-threaded if the same libraries are employed. Initial implementations of the JVM use a software-based cache to map from objects to their associated lock structures. This avoids imposing any per-object space overhead but places the cache look-up on the critical path of many operations. Recent implementators of the JVM have developed techniques for making these operations efficient in the common (uncontended) case.

Bacon *et al* proposed *thin locks* [Bacon98] in which atomic updates on a single field are used to acquire and release the lock in the absence of contention. A contended lock becomes *inflated* to form a *fat lock* including additional structures, such as wait queues and an operating-system supplied mutual-exclusion lock. Agesen *et al* report several disadvantages with this scheme: locks never become thin again, the implementation admits a possibility for unbounded busy waiting, and the thin lock field introduces a 24-bit per-object overhead. They designed an alternative representation in which atomic operations are used to acquire a *meta-lock* on a multi-use field in the object header [Agesen99]. The two-bit meta-lock field can represent uncontended states directly, otherwise it is used to arbitrate access to externally-held locking information associated with the object. When a lock is contended then the contents of the multi-use field are displaced to the full lock structure.

As with many other object-oriented run-time environments, the existing JVM does not provide flexible meta-classes in the manner of CLOS. Although it is true that, in some situations, the classes within the VM may be considered to be instances of `java.lang.Class`, it is more accurate to consider instances of `java.lang.Class` merely to *represent* those classes for the purpose of providing run-time access to their definitions. It is neither possible to derive subclasses from `java.lang.Class` nor to control the run-time behaviour of classes by making method invocations on it.

2.1.6 The Mite Virtual Machine

Mite is a VM designed to provide a common execution platform which is independent of both the underlying machine and the programming languages with which it is used [Thomas99]. It provides a layer of abstraction above the native facilities of the processor and, in doing so, allows almost all of the code in a system to be executed within the VM. Unlike the heavyweight systems of the Smalltalk and Java VMs, Mite does not attempt to provide a full programming environment in itself: the intent is that functionality normally present in the operating system (or code libraries) will be implemented over the VM, rather than being an integral part of it. The goal of this approach is that these other components, in addition to normal applications, will benefit from the portability and potential code-reuse enabled by a common low-level programming interface.

The portable binary format defined for Mite is designed for quick translation to native code without sacrificing language-neutrality, processor-neutrality or the quality of native code that could be generated. The instruction set uses a RISC-inspired three-operand code. Programs are concretely represented in a simple bytecode format comprising a module header and instruction stream. Constant data and static data areas are defined within *data blocks* specified inline in the instruction stream.

Heterogeneity in machine word sizes is tackled by using *three dimensional* numbers to represent the sizes of data items and offsets within data structures. An expression $b@w@r$ has the value $b + Aw/8 + 4\lfloor A/64 \rfloor r$, in which b is a number of bytes, w is a number of words, r is a number of 'roundings up' and A is the size (in bits) of the representation of a memory address.

In addition to code and data, the binary format can contain *hints* to the native code generator. These are intended for use in decisions that must be guided both by high-level information known to the language-to-bytecode compiler and by low-level information known to the VM implementation. An example is in the association between program variables or compiler-produced temporary values and the physical registers of the machine.

2.1.7 The Microsoft Common Language Runtime

The Microsoft Common Language Runtime (CLR) forms part of the `.net` framework. As a whole `.net` is intended to provide facilities for constructing and deploying applications in a flexible manner – for example partitioning code between client-side and server-side execution. The system supports multiple programming languages, including mainstream languages (C, C++, Visual Basic), current research languages (Mercury, Haskell) and at least one new language (C#, pronounced 'c-sharp').

As with the Java and Smalltalk virtual machines, code presented to the language runtime is expressed using a bytecode format. The particular operations available are in some ways similar to those of the Java Virtual Machine in that the basic execution model is a stack based object-oriented language with single inheritance of method implementations. However, there are notable differences.

Firstly, the basic integer formats available include *unsigned* as well as *signed* variants of each type. This may simplify the use of the VM as a target for languages that require unsigned types. In contrast the designers of the JVM claimed that it was sufficient to provide only signed types along with a guarantee that negative numbers are held in a twos' complement representation.

More substantial differences lie in the approach that the two virtual machines take to safety. As described in Section 2.1.5, the JVM requires that classes are subject to load-time bytecode verification – effectively ensuring the correct construction and type safety of the contents of the Java `.class` files. If an application needs to perform operations that cannot be verified as correct then those sections of it must be distributed in a separate system-specific binary format. In contrast, the structure of the Microsoft CLR permits a mixture of verified and un-verified methods to be provided in a portable bytecode format: some bytecode operations (such as direct access to memory locations) may only be used in un-checked code because their use cannot generally be verified. There are mechanisms to control how un-verified code may come to be executed on a particular instance of the CLR. Previous systems, such as Modula-3 [Nelson, editor91], have provided similar facilities to allow a single language to be used to write both safe and unsafe portions of an application.

A further innovation of the Microsoft Intermediate Language (IL), compared with Java bytecode, is the introduction of 'hints' from the IL-generator to subsequent native code generators. These hints are encoded directly in the instruction stream as operations. In the current specification these can be used for a number of reasons. Firstly, hints can identify particular programming idioms, such as sequences of IL instructions passing method parameters, so that these may be translated more easily into efficient code. Secondly, they can be used to describe how the local variables manipulated by the IL map back to a static single-assignment (SSA) form [Cytron89] – the aim being that run-time code generators would not need to derive such a representation.

2.1.8 Miscellaneous virtual machines

Cint is a run-time system for the C programming language driven by the principles used to design RISC processors [Davidson87]. As with the JVM, the Cint VM (CVM) uses stack-based operations in the expectation that this simplifies both compilers targeting the Cint system and the production of simple interpreter-based implementations. The CVM uses separate stacks for evaluation (the E-stack) and for function invocation (the C-stack). Although the authors claim that the CVM is inspired by RISC principles, it includes higher-level operations that are usually implemented in the C standard libraries – for example the string manipulation function `strncpy` is provided as a primitive. Function call and return, with fixed on-stack layouts, are also provided as basic operations.

Clarity MCode is a high-level machine-independent intermediate representation used as part of the tool chain for compiling *Clarity*, a C++ dialect influenced by OMG IDL, Modula-3 and ADL. As with the previously-described systems, the basic operations are stack based. However, the conventional control flow operations of C, such as `switch` statements, are represented explicitly in the bytecode so that a compiler back-end can select an appropriate implementation. The MCode distribution format carries some optimization details from the front-end, such as variable aliasing information and flags to identify leaf procedures.

Liedtke proposes an 'unconventional' code distribution format: the user-space operations and instruction formats of the x86 microprocessor along with a standardized interface to the operating system and services [Liedtke98]. He motivates this choice by observing that there are many compilers and tools for that architecture and that the execution environment is already widely deployed – either as actual workstations using the x86 processor, or as existing binary-translation tools.

2.1.9 Discussion

The development of the virtual machines described in this chapter has spanned more than thirty years. A number of trends are apparent, both in terms of the situations in which VMs are used and the structure of the systems themselves.

Performance of simple implementations

Foremost among these trends is the commercial acceptance of VM-based environments for deploying general-purpose applications – either directly interpreting bytecode instructions (as in early implementations of the JVM) or by using a simple run-time code generator [Agesen00]. The widespread availability of more powerful computer systems enables such simple VM implementations to operate adequately for many tasks.

However, there has been much more development in how high-performance VM implementations are created. In particular, the use of feedback-directed optimization is now well-established as the focus of research, rather than the implementation of efficient interpreters. The SELF, Smalltalk and JVM systems described here all use some form of dynamic feedback to improve the performance of method invocations – either in place of a whole-program class-hierarchy analysis that might be used in a static compiler, or as a mechanism for specializing common library implementations according to the applications within which they are used.

Meta-information to aid run-time compilation

The second trend, apparent in the MCode, Mite, JVM and Microsoft CLR systems is again related to program optimization. Those systems all provide some way for

a bytecode-generating compiler to signal 'hints' about the code to a run-time code generator. Of course, such hints could also be used by a traditional off-line code generator producing an ordinary native binary format, but the motivation appears primarily to be reducing the work done by the run-time compiler, rather than simply to avoid loss of information. The co-design of an appropriate instruction format and meta-information remains a subject for future research. Other authors have investigated alternatives to bytecode representations, most notably Kistler and Franz' *Slim Binaries* [Kistler96] in which the distribution format is a compressed version of the Abstract Syntax Tree (AST).

Object-oriented type system

The third trend is the de-facto standardization on an object-oriented programming model with single inheritance of method implementations – for example as seen in the Smalltalk, JVM and Microsoft CLR platforms. As we shall illustrate more thoroughly in Section 3.1, such an object model (with some extensions) is sufficiently flexible to form the basis of a target language for many compilers. Furthermore, it is directly suited to contemporary languages such as C# or the Java Programming Language and, more pragmatically, can be implemented with reasonable performance by associating a single virtual method table structure with each class.

Convergence with OS functions

Finally, more recent VM designs, including Smalltalk, the JVM and the Microsoft `.net` system, provide many of the facilities that have traditionally been associated with computer operating systems. In each of those cases the environment exposed to the programmer includes abstractions for input/output devices, network access and (to varying extents) some analogue of process management. This is a substantial

departure from earlier designs such as INTCODE which were intended to provide only an abstraction of the processor itself rather than of a complete execution environment.

The degree to which different VMs provide these facilities does still vary a great deal. For example, from within the Microsoft CLR, the programmer can make invocations on general Component Object Model (COM) interfaces – when allowed by the security policies in place. The Mite VM is a notable exception to this trend; however it was intended to provide a very lightweight abstraction layer over which traditional Operating System (OS) functions could be provided if desired.

2.2 Extensible operating systems

Section 2.1 described a number of contemporary VMs along with earlier designs from which they were developed. One of the observations made in that section was that current systems were taking on many of the tasks that had traditionally been performed by the operating system. Of course, there are close analogues between the purpose of a VM and the purpose of an operating system: both are concerned with providing *isolation* between various parts of the systems and with building an *abstraction* of the underlying system.

This section concentrates on work in the field of extensible operating systems – meaning ones in which policy decisions (and typically implementations) are devolved either from code executing in kernel mode to code executing in user mode, or from code executing with administrator privileges to code executing on behalf of an ordinary user. This has been done, for example, to allow applications to control network protocol implementations, thread scheduling, file layouts and on-disk meta-data formats. Chapter 4 will then draw on this work in the design of an extensible virtual machine with the aim of similarly devolving control from trusted to untrusted code.

Seltzer *et al* discuss the problem from five points of view [Seltzer97]: the *technology* that is used to extend the system, the *trust and failure* of extensions, the *lifetime* over which a particular extension remains in effect, the *granularity* with which aspects of the system may be extended and the approach that the system takes to *conflict arbitration* between incompatible extensions.

As an example, consider the design of the contemporary Linux kernel against these criteria. Aside from the simple mechanism of 'extending' the system by re-compiling and booting a new kernel image, Linux supports dynamically loaded modules containing privileged binary extensions. These are typically used to introduce support for new network protocols or new devices. The *trust and failure* of extensions are handled crudely: modules must be loaded by trusted users and thereafter operate with the full privileges of kernel code. Extension failure can have arbitrary effects. An extension may be loaded and unloaded while the system is operating and it consequently has a flexible *lifetime* which need not correspond to machine reboots. The *granularity* with which the system may be extended is fixed for a particular kernel image: modules are supported by hooks in various parts of the kernel and so extensions can only be supported where these hooks are present. The task of *conflict arbitration* must be performed manually by updating configuration files.

Seltzer *et al* also introduce a broad classification of extensibility in the operating systems that they surveyed in 1996.

Static extensibility

A *statically extensible* system is reconfigured at compile time. This allows customization to a specific workload – either by omitting parts of the system which would be unused, or by selecting between different implementations of a particular interface which provide different trade-offs. This is typified by the Scout operating system, designed to handle media streams [Montz95]. A running Scout system comprises modules whose interconnections are specified at build time, allowing cross-module optimization.

Dynamic extensibility

A *dynamically extensible* system allows reconfiguration to be performed while the system is running. The approach taken in microkernel operating systems, such as Mach, is to move functionality from the kernel into user-level server processes [Tanenbaum92]. These server processes may be changed without restarting the kernel. However, the server processes interact with the kernel through privileged interfaces and so system configuration must be performed by a trusted administrator.

Library extensibility

A *library extensible* system goes further in that it moves functionality from privileged server processes into untrusted applications. The system is *library* extensible in that most applications are expected to draw on implementations of this functionality from shared libraries. However, if an application programmer so desires, they may bypass the shared library code and obtain direct access to unprivileged lower-level interfaces. The exokernel design provides one example of such a system, in which it is envisaged that a number of different *libarary operating systems* can be supported (each library operating system resembling a different traditional OS) [Kaashoek97].

Extensibility through downloaded code

Finally, a system based on *downloadable code* takes the opposite approach: rather than allowing functionality to be implemented in user-space it allows applications to download code into the running kernel. This approach is exemplified by SPIN which uses it with the aim of allowing applications to extend operating system interfaces and implementations so that they are better matched to the application's requirements [Bershad95]. This is motivated by examples from the database, distributed-programming, multi-media and fault-tolerant programming

communities. By downloading code into the kernel SPIN aims to co-locate extensions with existing services and thereby support low-cost communication between the two. The safety of the resulting system is ensured by requiring that the extensions are written in a type safe language (Modula-3 [Nelson, editor91]) and compiled using a trusted compiler. This safety allows SPIN to use pointers as capabilities for access to kernel resources. These can be passed to untrusted user-space applications (which need not be implemented in Modula-3) by exporting them as indices into a protected per-process capability table. Extensions have been developed in SPIN to control paging to disk, thread management and specialized network protocol stacks.

Discussion

Two systems, the Xok exokernel and the Nemesis operating system, will be described in more detail in Sections 2.2.1 and 2.2.2. Their design is most relevant to this dissertation because they share the aim of allowing untrusted users to deploy system extensions, rather than simply providing extensibility as a mechanism for administrators to reconfigure or tune an operating system.

A further category of extensibility, for which the recent work post-dates the survey of Seltzer *et al*, are systems that are based on *hierarchical confinement*. These allow individual processes to control many aspects of the environment in which their sub-processes execute. Consequently a process that appears, at one level, to be an application, can appear to its sub-processes to fulfill many of the roles of the operating system. Section 2.2.3 describes this work in more detail.

2.2.1 Exokernel

Xok is an exokernel implementation for Intel-x86 based machines. *ExOS* is a library operating system (LIBOS) for *Xok* which provides a UNIX-like environment, including fork-based process creation, inter-process signals, pipes and sockets [Kaashoek97].

Proponents of exokernels give three reasons for providing applications with low-level resource access [Engler95]. Firstly, an application may take advantage of hardware advances without requiring that the kernel be upgraded. Secondly, an application may gain improved performance by tailoring policies to its own requirements – for example page replacement in a virtual memory system. Finally, designing policy implementations for particular applications rather than for the general case may enable specialization.

These goals were translated into four design principles [Kaashoek97]:

- *Protection and management should be separate*, meaning that the exokernel should provide the mechanisms needed to control access to resources but that management decisions should not be dictated by the kernel.
- *Allocation and revocation should be exposed to applications*, so that they may be designed to operate within a particular resource budget and, if their resource allocation changes, they can control which instance of a particular resource to relinquish.
- *Physical names should be used wherever possible* in order to avoid unnecessary translation steps.
- *System information such as configuration and global performance metrics should be exposed* so that applications may make informed decisions about the likely effects of trade-offs.

Although the exokernel design generally fits in the category of library-based extensibility, its realization in the *Aegis* and *Xok/ExOS* implementations also uses downloadable code techniques. This is to avoid the runtime costs of frequent up-calls to user-space libraries.

Xok uses a number of techniques to protect the kernel from errant library operating systems and to allow one library operating system to control access to its data

structures from the applications that it manages. These include *software regions*, which are areas of memory supporting fine-grained protection by requiring that all accesses are made through system calls, simple *critical sections* which are available in some environments by disabling software interrupts, and *hierarchical capabilities* that must be provided with particular system calls and may be delegated from one LIBOS to the applications running over it.

The disk storage system employs a novel design to retain protection while allowing applications to determine their own on-disk meta-data formats and file-layout policies [Kaashoek97]. A LIBOS developer specifies the behaviour of their algorithm by providing an untrusted deterministic function (UDF) to translate the LIBOS-specific on-disk meta-data format into a standard representation that the kernel understands.

A UDF is deterministic in that the output of the function must depend only on the inputs and not, for example, on other aspects of the system or LIBOS state. Each different meta-data format has a function *owns* which maps from a piece of meta-data (m) to the set of blocks accessible from m. When a LIBOS requests that a block is allocated to m (and, correspondingly, should a block be revoked from m), the kernel evaluates the UDF with the original value of m and with the new value, m', proposed by the LIBOS. The kernel only accepts m' if the result of the UDF reflects the requested update.

2.2.2 Nemesis

Nemesis was conceived as a *multi-service* operating system, meaning that it should be capable of handling a diverse and changing mix of applications and that these applications would extend beyond traditional workstation tasks to include capturing, processing and displaying multi-media streams [Leslie96]. To that end it places a particular emphasis on accurate resource accounting and resource management

because these are particularly important when handling media streams – overload is the common case and so it is necessary to arbitrate between the different tasks.

The primary design principles in Nemesis are that resources should be multiplexed at the lowest level and that physical resources should be exposed to applications wherever possible. As an example of the first of these tenets, if a system is handling network connections then data should be demultiplexed as soon as soon as possible upon receipt and further processing should be performed by the application requiring the data. As an example of the second, applications should be explicitly aware of the amount of processing time and physical memory that they have been apportioned. Although the motivations differ, this design leads to similarities between Nemesis and library-based exokernel systems.

Low-level demultiplexing makes it feasible to account the resources used by each task. This is because when handling de-multiplexed data there is a direct association between the stream of data and the task to which it is being accounted. In contrast, when handling aggregates, it is much harder to account usage back to particular tasks. This leads to QoS crosstalk where application performance depends critically on shared resources whose allocation or scheduling cannot be controlled [Tennenhouse89].

The interfaces defined in Nemesis provide applications with control over many of the resources that are required for their execution:

Disk I/O

In Nemesis, *User-safe disks* (USDs) provide an extent-based interface for data storage [Barham97]. Clients perform I/O operations over stream-based *rbufs* channels [Black94] with which resource allocations or disk scheduling parameters may be associated. Access control is performed on an extent-level granularity by a trusted

user-space filesystem driver. The USD driver maintains a cache of permission checks and makes call-backs to the filesystem driver upon cache misses.

Memory management

Self-paging is used by applications which require virtual memory [Hand98, Hand99]. Separate interfaces are used for allocating (or potentially revoking) physical and virtual address space. The application is responsible for electing which physical frames should back which regions of its virtual address space and for paging data to disk where necessary. This design aids the accountability of resource usage (for example, accounting disk operations to the application requiring them) and local optimization (for example to avoid writing a page containing unallocated space in the heap).

Thread scheduling

Although Nemesis provides applications with firm guarantees of processing time, beyond this it is the responsibility of the application itself to multiplex the time that it receives between different threads of execution. An application does this by providing a *user-level thread scheduler* to implement a particular scheduling policy.

The design is conceptually similar to the *scheduler activations* developed by Anderson *et al* [Anderson92]. However, the interface through which the user-level scheduler interacts with the kernel scheduler is notably different.

The key fact about Anderson's scheme is that the user-level scheduler is informed *explicitly* when it is allocated the CPU. Rather than transparently resuming execution at the point at which it was suspended, the process scheduler invokes the user-level

scheduler by making an up-call to its *activation handler*. It is then the responsibility of the activation handler to select which thread to resume. Separate notifications inform the user-space thread scheduler when a thread becomes blocked or unblocked during I/O.

Anderson's scheduler activations handled preemption by having the process scheduler record the pre-empted state and pass it back to the application when it was next activated. The activation handler could elect to resume from that saved state, or it could switch threads by resuming from a different saved state that it may have stored. This approach is problematic because it is unclear how to proceed if an application is preempted before its activation handler has resumed from a saved context.

The approach taken in Nemesis differs from Anderson's in two ways:

- Firstly, complete state records are not passed between the process scheduler and the user-level scheduler. Instead, each application has an allocation of *context slots* in which saved processor states can be recorded. The activation handler selects which slot is used to hold the processor context next time that application is preempted. This allows the kernel to maintain control over the saved contexts, since they may contain privileged registers which should not be modified by the application.
- Secondly, the user-level scheduler may *disable* activations. If the application is preempted while activations are disabled then the context is recorded in a designated *resume slot*. When the application is next scheduled then it is restarted directly from the resume slot, rather than by activation. Furthermore, whenever activation occurs, the process scheduler atomically disables further activations. This avoids the potential race condition when an application is preempted before the handler resumes from a saved context. An atomic *resume and enable activations* operation is used to resume threads.

This scheme allows Nemesis to support a spectrum of user-level scheduling policies. It also enables lightweight critical sections to be implemented in pre-emptive uni-processor environments: thread switches within an application can be prevented by disabling activations.

Many traditional thread scheduling policies can be implemented directly. For example, the activation handler in a preemptive round-robin scheduler need only maintain a circular list of runnable threads and resume each of these in turn. A non-preemptive version differs in that it continues to resume a particular thread in the list while it remains runnable, at which point the scheduler advances to the next thread. A priority-based scheduler can be developed as a simple extension to these by sorting the threads according to their priorities.

More interestingly, this system enables policies in which threads receive their own soft-real-time guarantees. A number of implementation schemes are possible, the most straightforward of which is for the user-level scheduler to maintain its own accounting information about the requirements of its threads and for it to use this information to select which thread is resumed upon activation. An alarm timer allows the user-level scheduler to cause itself to be re-activated when the timer expires. If the application is preempted before this happens then the process scheduler cancels the timer. The alarm timer therefore allows a user-level scheduler to set an upper limit on the length of time which it can donate to a thread.

2.2.3 Fluke

A common concern over library-based extensibility is that it favours *local optimization* within each application in preference to *global optimization* over the entire system. Kaashoek *et al* attempt to address this in Xok/ExOS by providing mechanisms for *controlled sharing* of data between applications using the same library operating system.

They speculate that superior global performance may be possible by using the newly-enabled intra-application optimization to reduce resource wastage, by allowing groups of applications to perform inter-application management of shared resources and by designing adaptive applications which can employ different algorithms depending on which resources are scarce. However, they admit that 'global performance has not been extensively studied' [Kaashoek97].

Ford *et al* present an alternative approach based on *recursive virtual machines* [Ford96a]. They use the term 'virtual machine' more generally than in this dissertation, using it to denote any execution environment such as that provided by an operating system to the applications running over it and interacting with it through a virtual machine interface (VMI). A recursive virtual machine is therefore one VM running over another, usually customizing the VMI in some way.

This architecture was implemented in *Fluke* [Ford96a]. It enables a modular OS design in which functionality may be implemented at different levels within a series of nested VMs or *nesters*. Unneeded services may be removed and existing services re-implemented or customized. Nesters form a hierarchy and so different applications or groups of applications can access different VMIs depending on the topmost nester over which they are executing. Ford *et al* envisage that many nesters will export VMIs which match the hardware architecture on which the system operates and also that the VMI will only cover low-level operations, such as the instruction set and system calls available to an application. Higher-level operations, such as file-system access, will be via inter-process communication (IPC) to user-level servers. Although this is similar to a *virtual machine monitor* [Creasy81], Ford *et al* claim that the rationale for the design is different: now it is a desire for flexibility rather than a means of multiplexing scarce resources without changing application software.

Fluke is implemented for x86-based hardware. This processor does not provide native support for self-virtualization (that is, for exposing an identical x86 VMI): there are unprivileged instructions which reveal 'global' state to which a nester cannot prevent

access. This problem is avoided by restricting applications to a particular subset of the x86 instruction set. An interface which replaces the missing functionality is made available.

This design allows flexibility between exposing low-level Application Program Interfaces (APIs) to programmers (and structuring applications as in a library-extensible system) and exposing APIs similar to those in a traditional OS. Furthermore, a single running instance of the system can use both approaches concurrently by defining global policies within nesters over which groups of applications run. For example, a shared buffer cache could be provided by a nester over which all applications run, or separate caches could be implemented within each application. This therefore allows system-specific trade-offs between intra-application and inter-application optimization of resource usage.

Ford and Susarla present *inheritance scheduling* as an example mechanism for use in such a system [Ford96b]. Schedulers are organized in a hierarchy in which they receive resources from their parent and donate these, in some scheduler-specific fashion, to their children. Ford and Susarla illustrate the case with a CPU scheduler in which the hierarchy may consist of schedulers representing different administrative domains within an organization. A low-level *dispatcher* implements primitive thread management functions such as marking threads blocked or unblocked and waking threads after timeouts expire. This is the only part of the system which does not operate in user mode. Donation is performed by a *schedule* operation which specifies a target thread, a *wakeup sensitivity* which controls when the scheduler should regain control – for example if the scheduler should be woken if the target to which it donates blocks, or if the scheduler should be woken if another of its client threads becomes runnable. Resource donation need not follow the scheduler hierarchy – the ability to perform arbitrary donation allows a natural approach to handling priority inversion since a thread which blocks may donate the remainder of its time to the thread which is holding the contended lock.

2.2.4 Discussion

Sections 2.2.1-2.2.3 have described approaches taken in the design and implementation of extensible operating systems. As stated in the introduction to this section, one common rationale for the systems presented here – the Xok exokernel, Nemesis, SPIN and Fluke – is that they can allow policy decisions to be deferred until much later in development than with traditional monolithic designs. Engler argues an extreme version of this case in his workshop paper [Engler95].

As we have seen, these policies typically relate to managing resources that must ultimately be controlled by code operating in privileged mode. Thus although many decisions can be delegated to untrusted code within applications some form of policing must be performed on the resulting decision before it can be put into effect. Canonical examples are checking that user-assembled network packets contain correctly formed protocol headers, or that direct-access rendering operations are made only to windows owned by the associated application.

This need both to delegate decision making, enabling policies to be tailored to application-required behaviour, while still policing results to ensure globally safe behaviour, has led to a characteristic separation of control-path and data-path operations in these operating systems. Typically, control path operations – such as opening windows or creating network connections – are expected to be in the minority and can consequently admit a more heavyweight implementation. In contrast, data path operations – such as individual rendering operations or packet transmissions – are expected to be frequent and any checking of application behaviour must be lightweight. An example is the use of downloadable code in the SPIN operating system: when a new extension is downloaded substantial effort may be expended to link it into the operating system kernel (for example, in the optimization phase of compilation).

The design presented in Chapter 4 for an extensible virtual machine will use similar techniques to safely devolve VM policy decisions on a fine granularity to untrusted applications.

Chapter 3

Flexibility in existing VMs

Chapter 2 documented the evolution of virtual machines (VMs). The discussion there described how current systems, typified by the Java Virtual Machine (JVM) and Microsoft Common Language Runtime (CLR), have adopted substantially similar type systems with single-implementation and multiple-interface inheritance.

This chapter concerns the way in which VMs of that design have been used and extended. In particular it considers the trade-offs in flexibility and performance that exist when using such a VM as an execution platform rather than compiling applications directly to native code.

This chapter is divided into three parts. Firstly Section 3.1 describes how current VMs have have formed platforms for the implementation of various languages and language-extensions. Secondly, Section 3.2 identifies a number of areas in which the interface provided by contemporary VMs prevents optimization opportunities that are used in non-VM-based systems. Finally, Section 3.3 relates these areas to the dissertation as a whole. In summary, the theme is that while the interfaces provided by current VMs are adequate for most programming tasks, the only interfaces currently exposed are at a much higher level than their counterparts in an OS. This means that

much functionality is fixed early in the development of a VM and therefore many of the criticisms made of monolithic operating systems may also be made here.

3.1 Support for multiple languages

This section describes recent compilation projects which use the JVM as their target. The range of source languages considered is interesting because it includes many for which tension appears between the language features and the design of Java bytecode. For example, Java bytecode provides strong static typing, whereas Lisp [Steele, Jr.90] and Scheme [Abelson98] are dynamically typed. In addition, the inter-method control flow operations in the JVM are designed for a class-based object-oriented language rather than for supporting higher-order functions, for example, or for selecting between methods on the basis of pattern matching.

3.1.1 MLJ

The MLJ [Benton99] compiler translates the functor-free subset of Standard ML (SML) to Java bytecode [Gosling95]; SML is a functional programming language which supports parametric polymorphism and higher order functions [Milner97]. It is subject to eager evaluation. In this case, the use of Java bytecode as a target language is attractive because it allows SML applications to operate on a broad range of platforms. The low semantic gap between SML and Java (compared, for example, with SML and C) permits safe inter-operability between parts of applications written in the two languages – both for Java code to call methods implemented in ML, or the ML programmer to work with libraries available for the JVM.

The system operates as a three-phase whole-program compiler. The first phase parses and type-checks the structure definitions (SML code modules) which comprise a project and translates them to terms in a typed Monadic Intermediate Language (MIL). The second phase aggregates the MIL terms and transforms them to low-level Basic Block

Code (BBC), a SSA-form intermediate language from which the final phase generates Java `class` files. The majority of the transformations are performed on the MIL representation.

The basic ML data types are represented by the primitive types of the JVM. For example a Java `int` is used in place of an ML `int` and an instance of `java.lang.String` in place of a ML `string`.

Product types in ML are represented by Java classes containing successive fields of the product (some of the Java classes introduced by this mapping could be avoided by generating a canonical representation which is shared between multiple ML product types).

Summation types are represented by defining a Java class for each different type of summand. Each of these is a direct sub-class of a single 'summation' super-class which contains an integer tag field containing different values to differentiate between summands. The MLJ compiler detects two particular idioms which are handled more efficiently. Firstly, a summation type with only nullary constructors is represented by the primitive integer type. Secondly, summations which add a single nullary constructor to an existing data type are translated by introducing a new distinct value of the existing type and using this to represent values of the additional constructor (in many cases the Java `null` value may be used directly). Each ML exception declaration is mapped to a new sub-class of an 'exception' super-class.

Functions in ML are handled in three different ways:

- Functions which occur only as tail calls are placed inline in the generated code. The ML function application is translated to a `goto` bytecode.
- Functions used in a higher-order context are converted to *closures*. For each different function type occurring as a closure an abstract *apply* method is added

to a designated class, `F`. For each different set of free variables accessed within a closure, a new sub-type of `F` is created in which the values of the free variables are held in instance fields. The code within the closure is generated as an implementation of the abstract `apply` method within the appropriate sub-type of `F`. A similar representation is used in Pizza [Odersky97], the Intermetrics Ada 95 compiler [Taft96] and in JPP, a pre-processor which generates Java source code from a Java-like language which is extended to include block closures.

- Other functions are collected as `static` methods on a designated class. They are generally called with the `invokestatic` bytecode, although this can be translated to a `goto` branch in the case of a self-tail-call.

Polymorphic functions are translated multiple times – once for each instantiation of their type scheme. The risked exponential code expansion has not been observed in benchmark applications. Producing multiple translations allows each to be specialized, and for the arguments of primitive types to be passed in an *unboxed* representation. A similar approach is proposed by Odersky and Wadler in their heterogenous implementation of parametric polymorphism for the Java programming language [Odersky97].

The performance of the translator itself was found to be poor when compared with Moscow ML (4-11 times) or with SML/NJ (4-8 times) because of the whole-program analysis performed by MLJ [Benton99]. The performance of the generated code depends on the implementation of the JVM. The developers of MLJ reported that gathering results with early just-in-time compilers were hampered by errors in the JIT implementations. Qualitatively, the performance was found to be good on numerically-intensive benchmarks because basic arithmetic operations continued to operate on primitive types which the JIT compiler would recognise and translate to single assembly language instructions. Conversely, code which exploited higher-order functions behaved poorly. Benton *et al* suggest that this may have been a consequence of poor memory management within the implementation of the JVM [Benton99].

3.1.2 Kawa

Scheme is a non-pure functional language which provides first-class functions, lexical scoping, dynamic typing and eager evaluation with side effects [Abelson98]. Kawa is a toolkit which contains a substantially-complete implementation of Scheme through compilation to Java bytecode [Bothner98].

All Scheme types are mapped to classes in the JVM. This is even necessary for primitive arithmetic types because of the sub-typing relationship that exists within Scheme between successively specific numeric formats (for example, a Scheme real number is considered to be a subtype of a complex number). Similarly, classes are defined for Scheme collection types such as sequences, fixed-length mutable strings, fixed-length vectors, lists and pairs. Scheme symbols are represented by instances of the immutable `java.lang.String` class.

Binary arithmetic operations in Scheme may be based on the type of both of their arguments. This cannot be expressed directly in the JVM which supports only method dispatch based on the receiver class rather than a more general multi-method dispatch based also on the argument classes [Bobrow88, Chambers92]. In general a subtraction `x-y` is implemented by a method invocation `x.sub(y)`. The implementation of `sub` in the existing numerical classes tests the class of its argument and, if it is unknown, returns `y.reverseSub(x)`. This allows the hierarchy of numeric types to be extended without requiring changes at every level. In a similar manner to MLJ first-class functions are implemented as sub-classes of an abstract `Procedure` class.

Kawa provides limited support for continuations. The *call-with-current-continuation* operation is implemented by instantiating a Java exception object and passing this to the called function as the 'continuation'. A continuation is taken by throwing the exception object. The site of the `call-with-current-continuation` operation is covered by an exception handler which compares any exception objects caught against the one that it instantiated. If the exception references are identical then execution may continue from that point. Otherwise the exception is propagated.

If no handler is found then the continuation was generated by a *call-with-current-continuation* operation that is no longer on the stack. This case is not currently implemented because the most direct approach to providing complete support for continuations would require stack frames to be heap allocated if they are captured as part of the state of a continuation. The JVM does not provide access to stack frames as first-class entities.

3.1.3 Pep

Pep is a Just In Time (JIT) compiler for Java which operates by translating Java bytecode for execution on the SELF VM [Agesen97a]. As described in Section 2.1.4, SELF is a prototype-based pure object-oriented language.

Each Java class is compiled to two objects: one representing the static methods and fields, and another defining the ordinary methods and fields as a prototype for any instances that are created. Systematic 'name mangling' is used to reconcile some differences between the two object models, such as the distinction (required in Java, not in SELF) between field and method names, and the fact that a Java field definition may be hidden by a field of the same name defined in a subclass. Most Java methods are translated into two SELF methods. The first is used for normal virtual method dispatch. The second has a modified name, incorporating the name of the defining Java class, to support direct invocation through the Java `invokespecial` bytecode. Synchronized methods require a third translation to handle lock acquisition and release. Compilation to SELF bytecode is performed lazily by installing *stub methods* which trigger the translation of code upon its first invocation.

In general, Java integers must be handled as objects, although small (30-bit) values may be held in an unboxed representation. Floating point numbers are represented directly in SELF, although this diverges from the IEEE semantics used in the JVM because the underlying SELF implementation reserves two bits in each value. This approach necessitates the use of an abstract interpretation when translating the `dup`

and `pop` bytecodes because it is necessary to determine the type of the value on top of the JVM stack [1].

Exceptions are not primitive to SELF and so the control flow that results from throwing a Java exception must be encoded explicitly by translating the `athrow` bytecode into a method invocation which searches a stack of exception handlers. This has the unfortunate effect of introducing a cost for entering and leaving regions protected by handlers (because it is necessary to update the exception handler stack). Extensions to the SELF VM are proposed which would avoid this cost for 'passive' exception handlers.

In the Java programming language any object reference may be used as the target for a `monitorenter` or `monitorexit` bytecode or for an invocation of the `wait`, `notify` or `notifyAll` methods. In contrast locks must be instantiated explicitly in SELF, so each object in the generated code has an associated lazily-allocated lock object that is created when the object is used for the first time as the target of a synchronization operation.

Control flow within SELF VM methods is provided by three mechanisms: closure-like code blocks, a restart-current-method operation (used, for example, in tail call elimination) and non-local return. Branch bytecodes were added to the SELF VM to handle some cases of the more general `goto` operation available in Java bytecode. Some limitations are introduced by assuming that the input to the translator was generated by a particular compiler – for example that exception handling regions are nested.

In general Agesen found that Pep performed well on programs written in an object-oriented style – that is, programs using method dispatch as the primary control flow structure. It performed less well on computationally intensive applications – for

[1] Readers familiar in detail with the JVM may wish to note that it is invalid for generic `dup` or `pop` operations to operate on values of unknown type within a bytecode subroutine, so it is unnecessary to duplicate subroutine code as later proposed by Agesen *et al* for other reasons [Agesen98].

example a slight slowdown was reported when executing `javac` when compared with version 1.0.2 of the Java Development Kit. Agesen suggested that these effects were due to the use of type feedback to optimize method dispatch in the SELF VM and to the use of boxed representations for primitive types.

3.1.4 Ada 95

Intermetrics have implemented an Ada 95 compiler which generates Java byte-code [Taft96]. Each Ada package is translated into a class on which library-level variables are defined as `static` fields. Each record type within a package is translated to a separate Java class. Non-tagged operations on the record become `static` methods. Tagged operations become instance methods. Record extension is implemented by causing the class generated for the extended record type to extend the class generated for the base record type. Record types with names derived in idiomatic ways from the enclosing package are treated as a special case in which, rather than generating a separate class, the fields and tagged operations are generated directly on the class representing the package. This reduces the number of classes generated and allows the generated code to present a more natural interface to other classes executing on the JVM.

Variant record type definitions are translated into sets of classes in which one class, representing the common fields, is extended by class definitions for each of the variant choices. Run-time checked casts are introduced to select the appropriate sub-class when accessing fields within a variant record.

Protected type definitions are translated to separate classes on which the associated protected sub-programs are defined as `synchronized` methods. The semantics associated with Ada entries and queues are implemented by additional code on the exit of each method. Ada tasks map to sub-classes of `java.lang.Thread`.

Each Ada exception is translated to a separate class. These are ultimately sub-classes of `java.lang.RuntimeException` and so their use does not have to reflected in the `throws` clauses of the generated methods. (Ada does not require code to be

flagged with which exceptions it may raise).

Multi-dimensional Ada arrays are *flattened* to a single dimension rather than being represented by multi-dimensional arrays in the JVM. This provides a more efficient implementation since accessing a field within a multi-dimensional JVM array requires a succession of `aaload` bytecode operations – this is necessary for implementing non-rectangular arrays, but these are not available in Ada.

In general it is possible to use the scalar types of the JVM directly. However, this is complicated by the requirement that signed integer arithmetic in Ada 95 signals an exception on overflow (the integer arithmetic operations in the JVM are defined to wrap around using a twos' complement representation [Lindholm97]). The proposed implementation uses wider integer types where overflow is possible and introduces run-time tests to check for out-of-range values. An alternative would be to introduce run-time tests on the parameters of each operation that may overflow.

If a primitive value may be accessed by reference then it is represented in the generated code as a heap-allocated object and accessed through a wrapper object.

3.1.5 NESL

Hardwick and Sipelstein describe using the Java programming language as the target of a compiler for NESL, a high-level parallel language [Hardwick96]. They present three arguments for using Java as a viable intermediate language. Firstly, strong typing in the intermediate language can aid debugging the high-level compiler. Secondly, the inclusion of automatic storage management in the intermediate language can simplify the translation made by the high-level compiler and avoid the need to implement a new garbage collector. Finally, the commercial success of Java and the widespread deployment of the JVM provides a wide base of machines on which the generated code can be executed.

The NESL-to-Java translator is implemented in Perl and replaces an earlier version

which generated native code using the VCODE library [Engler96]. However, both translators generate code that relies on a large library of native vector-manipulation functions for numerically-intensive computations. Java is thus being used essentially as a scripting language to control the computation rather than being used for the direct expression of the program.

3.1.6 AspectJ

Aspect oriented programming [Kiczales97] is a technique for structuring software so that the code implementing each *aspect* of a program's behaviour is grouped together. Examples of different aspects include control over synchronization, error recovery, replication or object migration. The rationale is that separating such code from the main algorithms of a program aids readability (by allowing the reader to focus on one aspect of the system at a time) and aids code evolution (by enabling the implementation of one aspect to be changed while retaining the code implementing other aspects). A complete program is re-constructed from the separate implementations of each aspect by using a *weaver* to re-combine them.

AspectJ is a framework to support such an aspect-oriented approach in the Java programming language. It is intended that the basic algorithms of a program are implemented in standard class definitions which are then woven with a series of aspect definitions that insert code around the methods. A method may therefore be extended many times before reaching its complete definition. AspectJ allows code to be introduced *before* each invocation, *after* each normal invocation, as a *catch* clause to handle exceptions thrown in the method and as a *finally* block which is executed whether the enclosed code completes normally or by throwing an exception. Additionally *new weaves*, applied to classes rather than to methods, introduce further fields or method definitions [Lopes98].

By using AspectJ, Lippert and Lopes achieved a reduction by a factor of 4 in the

number of lines of code related to exception detection and handling in a Java-based framework for interactive business applications [Lippert99]. The candidate application had been created using a *design by contract* convention in which the parameters and results of cross-interface invocations were checked against pre- and post-conditions. Using AspectJ enabled the enforcement of these conditions to be segregated from the main logic of the application. This approach allows checks to be entirely removed after testing. As an example, one common post-condition was that all methods returning object references must not return the `null` value. This test was originally replicated throughout the code. AspectJ allowed it to be written once as a block to execute *after* each invocation of any method whose return signature is a reference type.

3.1.7 Jamie

Jamie is a pre-processor based extension to the Java programming language that implements *automated delegation*. Viega *et al* claim that this provides a viable alternative to using multiple inheritance of method implementations [Viega98]. Jamie allows a class definition to include a `forwards` clause after the specification of its superclass.

This takes the form `forwards Intrfc to Fld` and has the effect that invocations of operations on the `Intrfc` interface are forwarded to the corresponding operation on the instance referred to by the `Fld` field of the class being defined. The programmer must, of course, define a `Fld` field and initialize it with a reference to a class which implements the `Intrfc` interface. Forwarding is implemented by adding delegation methods for each of these operations.

This approach gives some of the benefits that have been claimed for multiple-inheritance of method implementations [Waldo91]. In particular, it allows multiple specialization in cases where different parts of the behaviour of an object are

not closely related – for example an `InputOutputStream` could delegate to separate `InputStream` and `OutputStream` objects that are initialized, upon construction, to refer to the same file. It is still necessary for the programmer to manually avoid (or resolve) conflicts when multiple interfaces contain operations of the same signature.

Jamie does not attempt to allow delegation of field accesses because there is no clear translation of this into Java bytecode.

3.1.8 OpenJava

OpenJava is a pre-processor based extension of the Java programming language [Tatsubori99]. It implements a compile-time Metaobject Protocol (MOP) in which *meta-level* classes define transformations that are applied to any ordinary classes tagged by a new `instantiates` key word and the name of the meta-class. A meta-class can transform the definition of a base class in various ways, including introducing caller-side and callee-side code in method invocations whose target is an instance of a class that instantiates the meta-class. As with AspectJ (described in Section 3.1.6) this can be used to separate error handling from the main logic of the program and as with Jamie (described in Section 3.1.7) it can implement delegation between instances.

An example application of OpenJava is in providing language level support for *design patterns* [Gamma94]. Tatsubori and Chiba describe the use of the MOP to generate glue-code automatically and to make explicit the rôle of each class that participates in a pattern [Tatsubori98]. They illustrate this by considering an *adapter* class that is used to implement a new interface in terms of an existing object which implements an old interface. In the terminology of the pattern, the adapter class contains a field of the type of the adaptee and methods which implement the new interface in terms of operations that are supported by the adaptee. They use an *AdapterPattern* meta-class to generate the majority of the adapter class and to implement methods which may be delegated directly to the adaptee.

3.1.9 VMLets

Folliot *et al*'s *Dynamically configurable, multi-language execution platform* is a flexible virtual-machine-based system designed to be able to execute programs written in any bytecoded language [Folliot98]. An application is 'typed' with the name of an appropriate *VMLet* that describes how the bytecode implementation of the program can be converted into a language-neutral internal representation.

3.1.10 Kimera

The University of Washington's *Kimera* project is described as a *distributed virtual machine* [Sirer98]. It is *distributed* in the sense that the functionality of a system such as the VM is decomposed into separate components such as the verifier, the execution service and the resource management service. This approach may increase the overall integrity of the system by containing the failure of individual components – for example by separating the address spaces within which the components operate. The manageability of the system can be enhanced by requiring that a common verification service is used within an organization, under the close control of the system administrators. Finally, this decomposition hopes to enable performance gains and scalability by performing resource-intensive tasks such as compilation and verification on dedicated machines.

3.1.11 Vanilla

The *Vanilla* project at Trinity College, Dublin [Dobson98] is a system in which parsers, type checkers and interpreters can be constructed from language fragments. Each of these components implements a language feature and describes how the feature is realized in concrete syntax, how it is represented in the abstract syntax tree, how it affects the assignment of types to program fragments and how the language feature should be implemented at run time. A *language definition file* identifies the

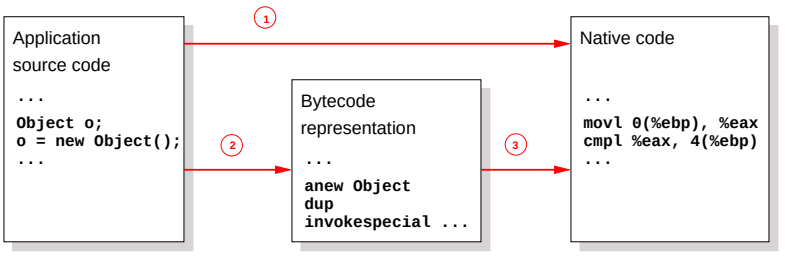


Figure 3.1: Possible execution paths for an application. In *(1)* it is compiled directly to native code, whereas in *(2)* it is compiled to a bytecode representation and subsequently *(3)* to native code.

components that need to be combined to construct the required language.

3.2 Limitations to flexibility

In systems such as those described in Chapter 2 the bytecode format of the VM is being used in place of a native binary format. As illustrated in Figure 3.1, if the VM itself employs a run-time code generator, the compilation of an application is effectively divided into two steps with the format of the VM bytecode defining the interface between the two. As explained in the introduction, such a division of responsibilities allows the evolution and deployment of the two compilation steps to be de-coupled – so long as the bytecode format remains fixed. With that separation of responsibilities comes a need to co-ordinate changes made to the bytecode format. However, in existing systems, that bytecode format provides the *only* mechanism for presenting application code to the VM. The bytecode operations, because they are semantically close to the source language, deal in terms of much higher level entities than contemporary native code formats: for example the Smalltalk-80 VM, JVM and Microsoft CLR all provide operations for object allocation and method invocation. Referring again to Figure 3.1, in a traditional VM-based system, optimizations at stage *(3)* can only be performed by the VM implementor and – at least in the case of untrusted code – there is no option of sidestepping this by re-writing portions of the application directly in native code or in an unsafe bytecode.

This section discusses a number of areas in which the style of interface exposed by existing VMs either restricts their suitability as general-purpose execution environments, or may be detrimental to performance.

It is naïve to view these restrictions, in themselves, as a problem. One of the potential benefits of using a high-level portable bytecode as a distribution format is that it enables optimization decisions to take into account the characteristics of the computer on which an application is being executed, or to allow its execution to benefit from developments in optimization technology. Similarly, if compilation is delayed until run-time then value-based or cross-module specialization may be possible. Franz advocates such a system in which code from various sources is combined dynamically into a single 'quasi-monolithic' image [Franz97].

However, one of the key tenets of the virtual machine proposed in Chapter 4 is that such concerns may be addressed by separating the underlying application from its specialized execution policies – thereby retaining flexibility for both the VM implementor and the application programmer.

3.2.1 Loss of information

In some cases the 'lost' opportunities for optimization may be avoided by passing hints from the bytecode generator to the run-time compiler. For example, although the bytecodes provided by the Mite VM (Section 2.1.6) present the abstraction of an infinite set of virtual registers, a native code generator may assume that numerically lower registers should be allocated to those of the physical processor in preference to higher ones. The JVM provides a general mechanism for including auxiliary information alongside standard class definitions -- in an extreme case this could include full native-code method implementations produced by a trusted compiler in combination with some form of code-signing system.

Without such facilities, other opportunities for optimization may be harder to recover – for example whether data flow dependencies exist between particular instructions (and consequently whether they can be reordered during compilation). Using a single VM as an execution target for multiple programming languages exacerbates this problem: a simple type-based analysis, in which mutually-incompatible object references cannot be aliased, loses precision if distinct types in the source language become coincident after compilation. It is easy to hypothesise various kinds of additional hints that could express such information (in much the same way as very long instruction word (VLIW) processors require instructions to be presented in bundles that may be executed concurrently [Hennessy90]).

3.2.2 Implementation of standard APIs

The comparatively high level of abstraction presented by the VM interface places it above the level at which decisions must be made, for example, concerning which conventions are used for laying out fields within objects or objects within the heap. Similarly, if library implementations are responsible both for implementing abstractions (e.g. of an object-based stream over a byte-based TCP network connection) and for protection (e.g. restricting the network addresses to which a connection may be opened) then both aspects of this behaviour may be moved outside the control of untrusted applications.

Heydon *et al* identified a number of examples within the Java 1.1 core libraries during their implementation and evaluation of a multi-threaded web crawler [Heydon00]. Library implementations of standard data structures were found to have been written to acquire and release mutual-exclusion locks on most operations – for example all insertions and deletions from a hashtable. Presumably intended to aid novice programmers, this degraded performance when the locks were unnecessary – either because the application was not multi-threaded or because access to the data structure was arbitrated in some other fashion.

Heydon *et al* cite particular examples in the use of `StringBuffer` objects during string concatenation, the acquiring and releasing of locks during low-level I/O operations (e.g. on each byte of a multi-byte quantity) and the updating of a cache used during network name-to-address resolution.

In other cases the implementation of the standard libraries leads to larger-than-necessary volumes of allocation on the heap. Various examples are identified, but the `BufferedInputStream` class is typical in that the target of the stream (that is, the lower level stream over which it provides buffering) is set only in the constructor of the object. Their is no way to re-target the buffer without re-allocating the buffer it uses.

In some cases these problems could be circumvented by re-implementing the library operations as part of the application. However, this is not always possible – either because the replacement cannot be implemented using facilities available to the untrusted programmer, or because it is invoked from a class which cannot itself be changed.

3.2.3 Run-time services

As shown in Chapter 2, VM-based environments such as the JVM and Smalltalk-VM typically provide extensive run-time services in addition to the underlying code-execution platform. In these cases, for the user's perception of the system to remain good, it is necessary not only to execute the application itself efficiently but also to ensure that the services on which it depends operate appropriately.

These arguments apply equally to applications executed directly within a computer operating system and they provided part of the motivation for the extensible operating systems described in Section 2.2. In each case the performance observed by the user depends both on decisions taken by the application programmer and also

on decisions taken within the implementation of the enclosing environment. This section describes examples of major VM components for which the selection of an appropriate implementation depends on the application.

Dynamic compilation

Section 3.2.1 described a number of areas in which the native code generated from a portable bytecode may be less efficient than similar code generated directly from the source language. A VM-based environment using a dynamic compiler faces the additional problem of scheduling this compilation work so that it presents acceptable performance at run-time: time spent during compilation can reduce time available to the application itself.

A simple approach to dynamic compilation is to invoke the compiler whenever a body of code is encountered for the first time – for example, in an object-oriented language, upon the first invocation of a particular method implementation. This so-called just-in-time scheme provides two principal benefits over compiling the entire application at load-time: only methods that are executed become compiled and this work is spread across the start-up phase of the application and so may be less perceptible to the user. In fact, facilities such as dynamic class loading make it generally impossible to compile an entire application in a single step.

A number of concerns have been identified with simple JIT compilation:

- Firstly, the quality of the native code generated by the JIT compiler is typically poor compared with the output of a traditional ahead-of-time system. This is due both to the need to perform compilation quickly and also to the piecemeal fashion in which the compiler operates – preventing optimizations that may require a whole-program analysis such as Class Hierarchy Analysis (CHA) [Dean96].

- Secondly, the time spent during compilation may be poorly scheduled within the execution of the application. For example, an interactive application may spend substantial periods of time blocked for user input and then – when input is received – spend time compiling the methods responsible for generating output. It may have been worthwhile to compile those methods in advance during idle time. In contrast, for a CPU-bound application, it is unreasonable to defer compilation until the system is idle because that opportunity will never arise.

- Finally, if an application is executed multiple times, then each instantiation of the VM will need to perform the compilation afresh. This provides opportunities for run-time value-based profiling and for optimizations based on the particular inputs to the application, but such techniques are not currently used widely and – in any case – may not be practicable to implement within the run-time budget available to a JIT compiler.

A number of systems have used heuristics to select which bodies of code should be compiled. Typically these identify *hot-spots* in the application – that is, regions which are executed frequently – and compile those in preference to less-frequently executed sections. This scheme exploits *locality of execution*. Just as those parts may come to be held in the instruction cache when executed as native code, there is a greater benefit to be had when compiling them from bytecode. Implementations differ in exactly how hot-spots are identified and how they are treated when compared with the remainder of the application. A simple technique is to count the number of times each method implementation is called and to invoke the compiler when this attains a threshold value. Some systems use an interpreter for initial invocations while others vary the level of optimization according to whether a method is being compiled for the first time (in which case a fast non-optimizing strategy is selected) or whether it is being re-optimized after identification as a hot-spot.

Recent work has evaluated the benefits of preserving compiled code between runs of a Java VM [Serrano00]. However, in earlier work on the Smalltalk VM, studies found

that the cost of recompilation may be less than the cost of saving and restoring the generated code and ensuring that changes to the bytecode cause re-compilation to occur [Deutsch84].

To evaluate dynamic compilation Hölzle introduced the notion of *pause clusters* – the aggregate delays produced by run-time compilation of multiple methods in quick succession [Hölzle94a]. He defined a pause cluster to be a maximal series of delays in which a specified proportion of the interval was accounted to compilation. With that definition in mind, a run-time compiler was found to perform better if compilation occurred only once a method had been executed several times.

The '*HotSpot*' implementation of the Java Virtual Machine contains separate bodies of code for 'client' and 'server' deployments. In this case the 'client' system is intended for use on a desktop machine with an interactive workload and a consequent emphasis on avoiding long pause clusters whereas the 'server' system is intended for bulk-processing tasks, perhaps with a preference for high-throughput over low-latency responses.

Memory allocation

Barrett and Zorn studied C programs which placed heavy demands on dynamic storage allocation [Barrett93]. They use profile-driven full-run feedback based on observed object lifetimes. Their motivation is to reduce the fragmentation caused by long-lived objects scattered throughout the heap. They are able to reduce the cost of allocating short-lived objects by placing them contiguously and delaying deallocation until large batches become free.

They attempt to correlate short object lifetimes with the most recent n return addresses on the execution stack. They found that there is typically an abrupt step in the effectiveness of prediction when n reaches some critical value. These critical

values varied between applications, but were usually not greater than 4.

The effect of using these predictions was evaluated through simulation with allocation traces. Each entry in the traces contained an identifier representing the object size and the complete call-chain to the allocation site. They estimate that the cost of computing a reasonable approximation to such an identifier may be between 9 and 94 RISC-style instructions for each memory allocation made. Such an overhead is, perhaps, reasonable for a free-list based allocator from the `libc` library. However, the fast-path of a simpler allocator may use fewer than 10 instructions and so even the best-case overhead of a further 9 instructions appears unsatisfactory [Harris01].

Zorn and Seidl used a similar profile-driven approach in which three special categories of objects were identified: *highly referenced* (HR), *not highly referenced* (nHR) and *short lived* (SL) [Zorn98]. A separate heap region was allocated for each of these three categories and the profiling results were used to modify allocation behaviour on subsequent runs. These divisions are designed to improve the program's usage of virtual memory pages. Their intuition is as follows: highly referenced objects should be densely packed together so that the pages they occupy will form part of the working set of the program, non-highly referenced objects should also be held with one another but segregated from other kinds of object, in the hope that the pages they occupy will not form part of the working set. Short-lived objects should also be held separately from the rest of the heap in order to avoid fragmentation in the remainder of the heap.

Cheng, Harper and Lee describe profile-based pre-tenuring in a generational garbage collector for an SML compiler [Cheng98]. They identify allocation sites by their program counter – this is perhaps more effective for ML rather than C because it is not customary for allocations to be made through layers of wrapper functions.

Cheng *et al* do not comment on whether the effectiveness of pre-tenuring is influenced by the usage patterns of heap-allocated data in functional languages (illustrated, for

example, in Stefanovic and Moss' analysis based on SML/NJ [Stefanovic94]). The major differences observed are a higher allocation rate of data records and a reduced update rate of existing data. However, it is unclear whether these language-level differences in the manipulation of data structures will be reflected in the native code generated by an optimizing compiler.

In contrast to such profile-based feedback-directed work, Vo exposes direct control to the programmer through the Vmalloc interface [Vo96]. The heap is divided into separate regions, each of which has a *discipline* (which defines how to obtain further memory with which to extend the region) and a *method* (which defines how the memory in the region should be divided to satisfy allocation requests). The programmer can define new kinds of region and can specify into which region an allocation is placed. A nested structure may be created using a *discipline* to obtain memory from an upstream heap rather than from an operating system allocator.

Vo observed that different definitions were effective with different workloads. A 'never free' policy is appropriate for a region in which the allocation lifetime will extend to the termination of the program. It may be more compact than a traditional 'first fit' or 'best fit' policy because the heap need not track the size of allocated blocks and so per-object headers can be avoided [2]. A futher policy, which allowed only the most recently-allocated block to be de-allocated, was found to work well with workloads which had generally-long-lived structures and needed occasional temporary storage.

Chilimbi *et al* investigated techniques for improving locality of reference by *cache-conscious* handling of data structures [Chilimbi99]. Their analysis considered tree-based *read-mostly* data structures and reorganized them at run-time between their creation and their use. They clustered data structures which are used together onto the same memory pages and used memory *colouring*, based on a description of the cache

[2] Vmalloc was implemented for use with the C programming language which does not require a mechanism for determining the size of an allocated block from its address.

hierarchy, to avoid introducing conflicts in the cache. Previous work used a copying garbage collector to reorganize the heap at run-time [Chilimbi98].

Chilimbi *et al* also investigated further techniques for *cache-conscious structure definition*. This focussed on the internal layout of data within a block of allocated memory rather than on the external organization of allocated blocks within the heap. An implementation for the Java programming language translated class definitions into *hot* and *cold* portions based on profile feedback. As with the HR and NHR regions in the work of Zorn and Seidl, the intent is that hot portions can be co-located to reduce (in this case) the number of cache-line fetches. Fields were also reordered so that those accessed with high temporal affinity were placed together.

Wilson *et al* present an extensive survey of techniques for dynamic storage and attempt to develop a conceptual framework and terminology for the discussion of the subject [Wilson95]. They identify the fundamental dilemma when implementing a storage allocator as the fact that the number and size of live blocks are controlled by the execution of the application: the only possible influence from the storage allocator is to decide where in memory to satisfy a request (or whether to reject the request). It must make such decisions judiciously to avoid fragmenting the heap into isolated blocks. In the general case it is impossible to avoid fragmentation without knowledge of the workload imposed by the application on the allocator [Robson74]. However, Johnstone and Wilson report that the worst-case bounds of fragmentation are not observed in current benchmark applications [Johnstone97].

In their survey paper Wilson *et al* study the effectiveness of different allocation schemes with six allocation-intensive benchmarks. They conclude that:

- *Program behaviour is usually time-varying*, meaning that the characteristics of the allocations requested will vary during a single execution of an application.
- *Fragmentation at peaks is more important*, because peaks in memory usage correspond to times when the allocator used by the application may need to

request further memory from the underlying operating system allocator.

- *Fragmentation is caused by time-varying behaviour*, because an application phase-change may alter the sizes of blocks requested.

- *Known program behaviour invalidates previous experimental and analytical results*, because much of the work Wilson *et al* surveyed were based on synthetic models of allocation requests whose behaviour differed significantly from the benchmark applications studied.

- *Different programs may display characteristically different behaviour*, for example in terms of object lifetimes, the total volume of active objects over the application execution or in terms of the phases exhibited during an application execution.

Garbage collection

Different garbage collection algorithms have very different run-time performance. For example, considering the basic schemes described in Wilson's second survey paper [Wilson92]:

- A *reference-counting* garbage collector associates an integer count with each allocated object, increasing this count when a new reference is created to the object and decreasing it when a reference is removed. An important consequence is that storage space may be reclaimed immediately that the count reaches zero. Disadvantages include heap fragmentation, the inability to reclaim cyclic data structures and the need for count-manipulations when traversing data structures in addition to when modifying them.

- A *mark-sweep* garbage collector traverses the heap, recursively *marking* each object that can be reached by the application. When a fixed-point is reached any unmarked objects are known to be inaccessible to the application and may

be *swept* – i.e. returned to the free list. A mark-sweep collector can reclaim cyclic data structures. However, it requires careful co-operation between application threads and the garbage collector – either suspending the application while garbage collection takes place or by introducing *barriers* on each access that the application makes to the heap. As with reference-counting, work must be performed for each object freed and freed space may be fragmented among retained objects.

- A *mark-compact* garbage collector, as with *mark-sweep*, uses an initial marking phase to identify live objects. It then *compacts* the heap by moving live objects so that they are contiguous. This means that work is performed proportional to the amount of data that remains live rather than the amount of data that is freed. It can enable straightforward memory allocation by placing objects contiguously in that free space.

- A *copying* garbage collector operates by recursively traversing objects reachable in the heap and copying them to a new area of memory. A simple implementation may divide the heap into two equally-sized *semispaces* and copy from one semispace into the other. As with compacting collectors, the work performed is proportional to the amount of live data.

In each of these cases numerous alternative implementations are possible. At a fairly broad level a garbage collector may be implemented so as to be:

- *Stop-the-world*, meaning that application threads must be suspended while the garbage collector operates. This may simplify the implementation of the collector or avoid the need for barriers in the application.

- *Incremental*, meaning that it can operate in a series of small steps – typically so that it is not necessary to suspend all of the application threads during each collection cycle. Reference counting is an example of an incremental scheme.

- *Concurrent*, meaning in addition to *incremental*, that the steps taking by the collector may occur at the *same time* as the execution of application threads [3]. In some cases a concurrent collector may dedicate a separate thread to garbage collection, or it may perform small increments of collection work as part of each object allocation.

- *Parallel* meaning that the collection algorithm is itself implemented using multiple threads.

Consequently, these different schemes provide a range of performance characteristics in terms of the *immediacy* with which space may be reclaimed, the ability to manage arbitrary data structures, the space overhead needed during collection (e.g. for semispaces or housekeeping information), whether work is done in proportion to the amount of live data, live objects, dead data, dead objects or heap size, whether it de-fragments free space and any additional work (e.g. barriers or reference-count manipulation) that may be required of the application threads. Jones provides detailed information in his reference text [Jones96].

As with storage allocation, the selection between different garbage collection algorithms is an area in which the use of a VM can provide both positive and negative effects on application performance. In particular run-time compilation can introduce appropriate barrier code at the point of application accesses to the heap. In contrast, if an application is distributed as native code, then the correct barrier must be selected when the application is compiled.

Conversely, as we have seen elsewhere, current VMs provide not only an *ability* for the garbage collector to be selected at run-time, but also a *requirement* that it takes place at that time, for example by the user invoking the VM with appropriate command-line options. An application with known behaviour cannot directly communicate this to the VM.

[3] Some authors use the terms *concurrent* and *parallel* interchangeably. The definitions used here follow those by Printezis and Detlefs [Printezis01]

Thread scheduling

In a multi-threaded application the thread scheduler is responsible for multiplexing the available processors between the threads which are eligible for execution.

In some applications this decision may appear straightforward – for example there is only one benchmark among those in the SPECJVM98 suite which is explicitly multi-threaded. However, even in the remaining 7 benchmarks, the behaviour of the thread scheduler may become significant where the VM performs housekeeping tasks using multiple threads. In the case of the JVM these include threads dedicated to other system services, such as concurrent garbage collection, run-time compilation or the execution of *finalizer* methods on otherwise-unreachable objects.

It is easy to see how the decisions made by the thread scheduler affect the performance that a user experiences. As an example consider the difference between a mutli-threaded batch-mode application and an alternative application in which each thread performs user-visible interaction. In the first case, if output is only generated upon completion, the primary concern is the overall resource requirements of the program – the user is unaware if the system switches between sub-tasks frequently or rarely (and equally if it switches regularly or sporadically). In the second case, where an application run of the same overall duration may interact with the user, then the goal may be to reduce the latency or variability with which each interaction occurs. The tension here is that any fixed overhead associated with each thread switch will degrade the overall execution time of the application.

More subtle concerns exist in a multi-processor multi-threaded environment. In that case there may be natural groups of application threads which should be scheduled on the same processor and other groups of application threads which should be scheduled concurrently but on different processors. Such situations arise from the communication patterns between threads and the extent to which the performance of a particular thread depends on state in per-processor caches or other resources.

Applications executing directly using native code may have various facilities available for controlling thread scheduling. The Solaris operating system provides a two-level mechanism which is of particular use in multi-processor environments. The application associates threads with Lightweight Processors (LWPs) which, in turn, are scheduled by the OS over the physical processors available to the process. Threads which should be executed on the same processor may be *bound* to the same LWP and, conversely, threads which should execute concurrently on different processors may be bound to separate LWPs. This scheme also enables an application to indicate that it is using multiple threads to simplify its own definition rather than in expectation of genuine parallel execution.

Where these facilities are not available the OS may provide sufficient primitives for a native code programmer to implement a user-supplied thread scheduler – as described, for example, in Section 2.2.2 this may be achieved by using specialized schemes in an extensible operating system. Additionally, and particularly where non-preemptive uni-processor scheduling is acceptable, the programmer may effect manual context save/restore operations by switching between multiple saved register sets. While potentially simple, such implementations are unsuitable for general workloads because the OS treats the entire collection of 'threads' as a single process which may therefore block.

3.3 Discussion

This chapter has described a number of compilers and pre-processors which allow a reasonably diverse set of languages to be executed over VMs designed for other languages. In practice most of these examples form part of the recent body on work on the Java Virtual Machine and consequently use that as a target. The work illustrates how, although the bytecode operations provided by the JVM were originally tailored for the Java programming language, they nonetheless provide reasonable facilities for supporting other languages.

The compilers considered here vary substantially in terms of how 'natural' the generated code appears. One way of assessing this more methodically is the extent to which the translation maintains abstractions that are present. For example:

- Whether data type definitions in the source code generate corresponding type definitions for the target machine, in the way that MLJ translates product and sum ML data types, or the way that Pep maps class definitions in the source language into prototypical objects for the target.
- Whether control flow structures in the source code map to typical control flow structures for the target VM – for example as Kawa maps Scheme operations on numeric types into method invocations on objects in the JVM, or MLJ maps non-tail function calls into static method invocations.
- Whether the generated code is executed directly by the VM or whether the 'translation' introduces a further interpretive layer over the VM. Such an approach was apparently taken originally in the JPython compiler from the Python scripting language to Java bytecode, however it has not been reported in the literature.
- Whether storage management is performed using the mechanisms provided by the VM (as in all of the systems described here) or whether storage space is modelled explicitly using a large array of bytes or similar construct.

Many of the areas in which abstractions are not preserved under translation to Java bytecode are either ones that may be ameliorated by proposed extensions to the JVM (in particular support for parametric polymorphism in the translation of ML [Bracha98, Agesen97b, Myers97, Solorzano98, Odersky97]) or by the additional control-flow operations available in the Microsoft CLR.

More formally, Abadi introduces the term *full abstraction* to describe a translation between two languages that maintains both pairwise equivalence and pairwise

non-equivalence between expressions [Abadi98]. He uses it to illustrate flaws in some simple translation schemes from the spi-calculus [Abadi99] to the pi-calculus [Milner92]. However, that notion appears less relevant here. Firstly, the languages in question are rarely associated with a definition of what it means for two programs to be equivalent. Secondly, experience with existing language-VM pairs (such as Java and the JVM or Smalltalk and the Smalltalk-VM) shows that these cases do not exhibit full abstraction and so it is unrealistic to expect that other languages targetting the same VM would do so.

Component-based virtual machines, such as the VMLets or Kimera systems described in Section 3.1, address some of these concerns of flexibility and extensibility. However, their approaches are more akin to a microkernel design than to systems such as Nemesis or the Xok exokernel. For example, the components from which a language is developed in Vanilla must be designed so that they do not conflict with one another and so that one component does not destroy invariants or security properties on which another depends. Similarly, although the decomposition proposed in Kimera allows a choice over where parts of the VM are implemented, these decisions need to be taken by the system administrator rather than on a per-application basis under the control of the programmer.

In summary, the compilers and translators described in Section 3.1 support the claim that the bytecode formats supported by existing VMs are sufficient for use as the target code for multiple programming languages – even if that is not the purpose for which they were originally conceived. Furthermore, the discussion of Section 3.2 argued that – in spite of this kind of flexibility – the abstractions presented by existing VMs preclude many traditional opportunities for optimization, or for run-time services to exploit application-specific characteristics.

Chapter 4

The design of an extensible virtual machine

Chapter 3 argued that although existing virtual machines can form execution targets for a large set of programming languages they provide little control over low-level aspects of how particular applications come to be executed. Furthermore, Section 3.2 identified a series of areas in which that kind of low-level control can make application execution more efficient. This chapter develops those observations into a design for a prototype eXtensible Virtual Machine (XVM) drawing on the techniques developed for other kinds of extensible system.

The design presented here has four main requirements:

[R1] *Applications should not be required to exercise control over low-level implementation details*. It is often appropriate to use a default policy (e.g. for general-purpose or prototype applications). By analogy, both the Xok and Nemesis library-extensible operating systems provide standard implementations of common OS abstractions.

[R2] *Nave or premature optimization should be discouraged*. That is, once a low-level interface is exposed then it may be that programmers use it inappropriately

or that their own 'optimizations' prove to be ineffective as new techniques are developed or as computer systems evolve. Such usages and optimizations should be discouraged.

[R3] *Code reuse should be promoted by allowing particular policy implementations to be used with multiple applications and, conversely, by allowing a single application to be used with multiple low-level implementations*. This is essentially an argument promoting a modular design in which clear interfaces demarcate boundaries.

[R4] *The new facilities exposed to untrusted programmers should not allow the safety of the* VM *to be compromised*. Informally, new interfaces exposed to the programmer should provide control over *how* the VM executes some portion of an application, without extending the range of possible paths that execution may take. Safety concerns are addressed separately and in more detail in Section 4.6.

4.1 Framework

This section describes the framework within which application-accessible run-time policies have been defined. Subsequent chapters will develop this further by describing the way in which this framework has been applied in particular problem domains.

There are three overall principles on which the design is based:

[P1] *Applications can be used without modification and, if executed using default policies, will behave without significantly degraded performance*. This promotes [R1] by allowing existing applications to be used rather than mandating that they are revised to include low-level implementation details.

[P2] *Policy implementations are expressed separately from application code*. This promotes [R1] because it encourages programmers to distinguish their concerns over application logic from those over execution policies. In so doing it provides

a defence against inappropriate optimization because a policy can be replaced without updating the remainder of the application [R2].

[P3] *Fully-featured programming languages are used, where practicable and appropriate, for expressing policy implementations*. The intent is that using a Turing-complete policy definition language enables the expression of non-trivial policies – for example, the *allocation inheritance* thread scheduler described in Chapter 7. In practice, a conventional programming language is chosen to take advantage of existing development facilities. The design of an appropriate domain-specific language remains a possible area for future research [van Deursen00].

The framework proposed here consists of four kinds of entity: Policy Registries (PRs), Untrusted Policy Implementations (UPIs), Protected Mechanism Implementations (PMIs) and class isotopes. These are described in Sections 4.2-4.5 respectively.

In overview, the UPIs implement application-supplied policies by receiving up-calls from the VM and making invocations on domain-specific PMIs. The PRs associate sections of the application program with the policy implementations that should be used. The fourth kind of entity, *class isotopes*, provide a mechanism for controlling the granularity at which policy decisions are made – they are collections of objects that have the same class but which should be distinguished by run-time policies.

Figure 4.1 outlines this structure and illustrates the communication paths that exist between the application code, VM infrastructure and policy implementations.

4.2 Policy registries

A PR provides a name service that maps from sections of the application to the policy implementations that should guide execution. Separate registries exist for each different aspect of execution over which the application may exercise control.

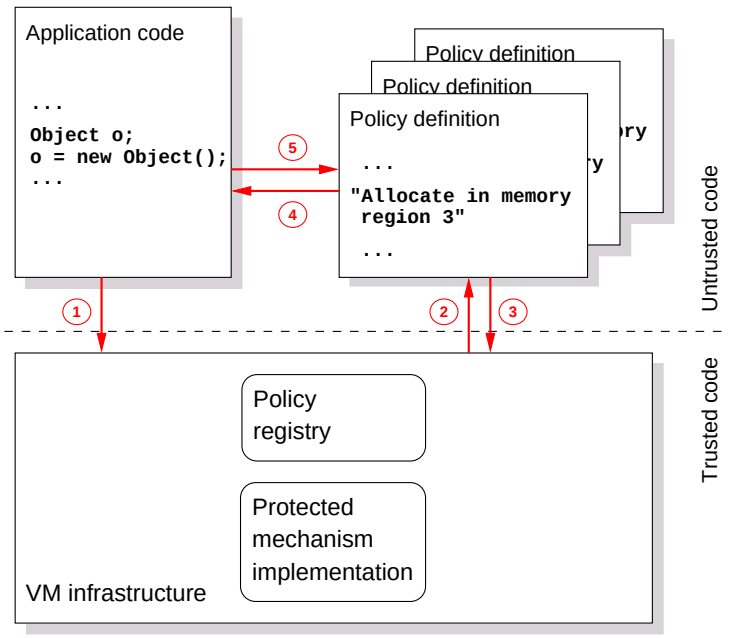


Figure 4.1: Overall system structure. Arrows indicate *(1)* the invocations made by the application on VM services, *(2)* by the VM on the selected policy definition and *(3)* by that definition on a protected mechanism implementation. Additional communication may occur *(4,5)* between the application code and policy definition.

In the extended implementation of the JVM described here, sections of the application are identified by their position within the package namespace. For example one policy may be associated with the `java.lang` standard libraries and another may be associated with code within the `uk.ac.cam.cl.tlh20` set of classes.

Figure 4.2 shows how the PR concerned with run-time compilation is used, along with different compilation policies, to co-ordinate the execution of a particular application. The standard classes whose names begin `java.*` are registered with a policy that loads a pre-compiled implementation – one that was generated off-line with a highly optimizing compiler. Classes whose names begin `uk.ac.cam.cl.tlh20.*` will be compiled in the background (see Section 5.6). The single class `uk.ac.cam.cl.tlh20.UserInterface` will not be compiled at all. Other classes will be handled according to the system's default policy.

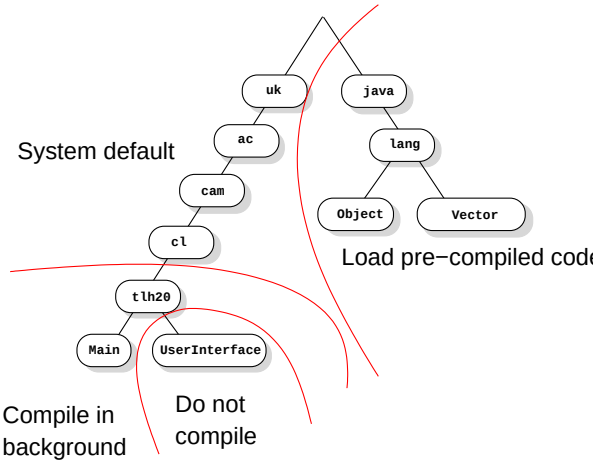


Figure 4.2: The PR is used to control how different sections of the package hierarchy are handled by a run-time compiler. In this example, standard classes are loaded from pre-compiled versions, part of an application is compiled in the background and another part will not be compiled at all. Ambiguity is resolved by selecting the most specific match.

4.3 Untrusted policy implementations

A UPI is responsible for implementing the policy decisions required for the application by the run-time system. A programmer defines a new policy by creating an appropriate UPI and passing it to the policy registry for that resource.

Concretely, policies are defined as classes that implement per-resource interfaces. For example, a `CompilationPolicyIfc` interface defines the operations required of a run-time compilation policy. The policy itself is expressed in arbitrary Java bytecode and therefore subject to the usual security mechanisms of the VM.

4.4 Protected mechanism implementations

The third type of entity are the Protected Mechanism Implementations. PMIs are used by Untrusted Policy Implementations to communicate their decisions to the VM. For example the protected mechanisms used within a run-time compilation policy include operations to invoke a native code generator and to perform various kinds of code-specialization, described fully in Section 5.1. Similarly, the implementation of a storage management policy can use operations for allocating space in various kinds of heap.

These mechanisms are *protected* in that they could not be implemented directly by the application programmer within a UPI. For example, over the JVM, protected mechanisms must be provided by native methods -- indeed they may be implemented and accessed through the standard Java Native Interface (JNI). Section 3.2.2 described how, in an unmodified JVM, the implementation of certain standard APIs could not be changed by the programmer: in the proposed XVM those implementations could be divided into UPIs (which could be changed) and PMIs which would be inviolable.

4.5 Class isotopes

The previous sections have described how *policy registries* map from sections of the application to run-time policy implementations, implemented by UPIs on the basis of the program state at the point at which a policy decision is required. For example, a UPI controlling object allocation would select into which region of the heap the new object should be placed. It could be invoked once for each static allocation site and this decision could be cached within the application – either by inlining the appropriate allocation function during run-time compilation or by updating a bytecode implementation for future use by the interpreter. This enables an efficient implementation because policy invocations are not required on subsequent passes through that section of the application.

However, this binding between policy decisions and static points in the application code introduces substantial limits on the kinds of policy that may be implemented. As an example, suppose that a hashtable data structure is defined using two classes: `Hashtable` provides the external interface to the data structure and, internally, refers to instances of `HashtableEntry` which in turn contain key-value pairs. If this forms part of a library of standard data structures then it may be used in many parts of an application where differing policy decisions are appropriate. In particular, the allocation mechanism for a particular site instantiating `HashtableEntry` may differ according to the mechanism used for the enclosing `Hashtable` – for example so that the buckets are placed in the same region of the heap as the overall table. Such a policy cannot be expressed if a single decision is bound to each static allocation site.

There are a number of possible schemes that could be used to allow more flexibility in the definition of policies:

- Firstly, in this particular case, a new allocation mechanism for buckets could be defined to examine the kind of enclosing `Hashtable` for which a particular bucket is destined to be used. This has many unsatisfactory aspects – it introduces extra complexity on each execution of the allocation (rather than simply on each policy decision), it may require additional facilities for run-time introspection, and it is predicated on the ability to define such an elaborate mechanism.

- Secondly, the programmer could reformulate their application so that separate classes are used wherever different policy decisions may be desired – perhaps, in this example, defining a separate `LongLivedHashtable` class. This approach is unwieldy in any reasonable application. Even in this example it would not be possible simply to define `LongLivedHashtable` as a sub-class of the existing definition. This is because the site allocating instances of the bucket class would still only statically occur once in the application whereas, to implement different policy decisions between the two kinds of table, the

allocation site must occur multiple times. This approach consequently detracts from the retroactive definition of run-time policies and the separation between application code and policy definitions. The need to consider such policy decisions directly within the application code detracts from the modularity and reusability of standard libraries. Furthermore, even if the application and policy classes are developed cooperatively, then there are situations in which the desired reorganization is not possible – for example if the class is *final* or *sealed*, meaning that further sub-classes cannot be derived, of if explicit tests are made on the class of the instances being manipulated.

The alternative approach taken here aims to combine some aspects of each of these schemes while avoiding their respective problems. It provides facilities through which a run-time policy definition may identify static program points which should be handled in different ways in different situations. A separate policy look-up is performed for each of those situations with the intent that, as before, the results may be cached inline.

This organization is achieved by separating the idea of an object's *class* (which defines the behaviour of its methods, the fields available from it and its relationship to other classes in the type system) from the idea of an object's *isotope* (which is the granularity at which run-time policy decisions are applied to groups of objects). Objects with the same class behave in the same way from the point of view of the application programmer. Objects that additionally have the same isotope are implemented in the same way within the VM. Isotopes are subdivisions of classes, so any pair of objects belonging to the same isotope will necessarily belong to the same class. For example, the instances of the `Hashtable` class may be divided between isotopes representing *long-lived hashtables*, *short-lived hashtables* and *unknown hashtables*.

Objects are assigned to isotopes under the control of the allocation policy in force at the point of their instantiation. This is because, as with the choice of allocation mechanism, selecting an isotope when compiling or interpreting an allocation site for

the first time lends itself to a straightforward and efficient implementation through inlining or bytecode rewriting.

A further conceptual decision is whether objects may by *transmuted* during their lifetime (that is, changed between different isotopes belonging to the same class). The running example of segregating long-lived and short-lived data suggests that the ability to make dynamic changes may be desirable in applications that exhibit different phases of execution. Such behaviour has been observed in common benchmark applications for the JVM [Harris01].

Changes to an object's isotope are not directly visible to the application. This follows from the fact that each isotope provides method implementations derived from the same bytecode definitions and that the language type system is concerned only with the class of each object. In contrast, changes to an object's isotope *are* visible within the implementation of the VM and within the policy definitions. Changing the isotope of an existing object may complicate the VM by invalidating the assumptions on which some methods have been compiled. For example, suppose that an object initially belongs to an isotope for which the run-time compilation policy performs extensive inlining of method implementations. If the isotope of one of the target objects is changed then these inlining decisions may be invalidated.

In this particular case the extra complexity is not great because an effective run-time compiler for the JVM must already maintain various kinds of association between compiled methods and the aspects of the application on which they depend. Furthermore, if the desire is to only provide safety in the sense that out-of-date implementations are never used, then compiled code can be discarded conservatively, providing a trade-off between the work required to re-generate it and the volume of dependency information that must be maintained. In the current implementation some control over this trade-off is provided to the policy implementor. This is achieved by recording and invalidating dependencies at the level of individual compiled methods. Since each compiled method is associated

with a (*source method*, *isotope*) pair the programmer may choose to introduce extra functionally-equivalent isotopes for those objects which are likely to be transmuted.

4.6 Safety and security

The safety of the system – when compared with an unmodified VM implementation – depends on controlling the kinds of invocations that may be made between the entities involved and the data that may be exchanged on those invocations.

Figure 4.1 shows the five kinds of interaction that may occur. There are three situations in which communication occurs across the boundary between trusted and untrusted system components. In the first case, labelled *(1)* the bytecode interface exposed by the VM remains unchanged. Similarly, invocations across interfaces labelled *(4)* and *(5)* are subject to the usual controls imposed by the VM for access to one class from another. The second and third cases are more problematic. In the figure these are labelled *(2)* for invocations from the VM to the UPI and *(3)* for invocations made by the UPI on the PMI.

By using the same language for expressing UPIs as for the remainder of an application, any operation performed within the policy implementation may equally have been performed directly by the application: in the JVM the policy implementation is subject to the same bytecode verification process that is trusted for the application classes. The trust placed in PMIs therefore comes from the safe design of their interfaces: from the fact that they control *how* the application is executed rather than *what* the application does. In general, the same techniques are used here as are used to ensure the safety of the system-call interface exposed by an operating system: access to PMI methods is restricted by ordinary access modifiers and the first stage in each method's invocation is to vet the parameters with which it has been called. The design of these PMI interfaces will evidently vary between different policy domains and so examples will be deferred until Chapters 5-7.

However, such a design does not address concerns over invocations made on interface *(2)*. This is because the proposed invocations of the VM on the UPI reveal information about the code being executed, if only through the timing of call-backs to a UPI. For example consider the extent to which a UPI implementing a run-time compilation policy should be permitted to examine the parameters passed to methods. Such information can be used legitimately to guide a policy that compiles specialized versions of methods that are frequently called with the same parameters. Conversely, the information can be used maliciously by an untrusted application to gain references to protected objects within the VM.

Various approaches could be used to tackle this problem. One would be to enforce a stronger isolation between the UPI and the application program. The intuition is that there is no need to be concerned about the UPI obtaining some particular information if there is no way that that information can be communicated externally – effectively disabling communication from the UPI across interface *(4)*. The use of multiple *class loaders* within the JVM provides an attractively simple implementation mechanism for such a scheme. The isolation this would provide is solely at the level of the VM type system: it would be deficient as a complete solution without a thorough analysis of covert-channel communication within and around the VM.

The prototype implementation uses a hybrid approach:

- Firstly, the parameters passed to UPIs are selected on a per-policy-domain basis so that they remain useful to potential UPIs while losing some of their potential for malicious use. For example a storage-allocation UPI does not receive a reference to the instance that has been allocated. Similarly, a run-time compilation UPI can inspect values passed to methods under its control, but it cannot change them. The system can also be configured (at load-time) so sensitive reference values are passed to the UPI as a corresponding integer value (from which the JVM type system prevents reconstruction of the underlying reference). The intuition is that software protection, based on the secrecy

of data manipulated by the program, tends to rely on the use of protected object references as simple capabilities. A more complex scheme could perhaps employ one-way functions to all parameters with the intent that the UPI may still determine where methods are frequently invoked with some particular value, but they may not learn the particular value taken.

- Secondly, where this hiding is insufficient, sections of the package hierarchy can be *finally* bound to particular policies [1]. This may be desired if the information leakage caused by merely calling the UPI is considered unacceptable. In particular, the classes which implement VM services may be bound to system-provided default policies. The user executing the VM is responsible for selecting when this is appropriate.

The current design aims to provide a pragmatic balance between the information available to UPIs and the desire to maintain the secrecy of certain pieces of system information. It also provides a model which is straightforward to describe to UPI implementors and to VM administrators. The ability to define *final* UPI*s* for particular parts of the package hierarchies allows the VM administrator to provide a more restrictive environment where desired.

It is worth emphasising one further point: the concern here is only to ensure that application-supplied UPIs operate *safely* – not that they necessarily obey any particular form of semantic correctness. For example, Chapter 7 will describe the form of UPIs used to define application-supplied thread scheduling policies. The design there logically involves the VM making up-calls to the application-supplied scheduler which, in turn, identifies which thread should be executed. It is considered quite safe for one of those up-calls to enter an endless loop, with the effect that the entire application will cease to make progress.

[1] Borrowing the terminology of the Java programming language in which a `final` field or method cannot be overridden in sub-classes.

4.7 Implementation environment

The work described here has been carried out as extensions to the implementation of the Sun Java Development Kit version 1.1.4, operating on x86 machines with the Linux and Nemesis operating systems. The JVM is particularly interesting in the context of Nemesis because it provides an execution environment which is in tension with the principles underlying the operating system: the VM traditionally exposes high level interfaces through which the resources handled by applications are removed from the physical resources of the machine. In the case of Java bytecode, the CPU is effectively being virtualized in two ways: both in the division of processing time between tasks and the language in which code is expressed. This environment motivates some of the emphasis on using run-time policy definitions to control resource consumption within applications.

4.8 Policy portability

There is, inevitably, a tension between exposing low-level aspects of a system's implementation and the portability of code that exploits that access. This is currently managed within the framework presented here by selecting the operations to expose on PMIs so that they are likely to remain both useful (to applications) and implementable (within the VM) as the environment develops.

In general this means that the entities manipulated by a policy should correspond to concepts already ingrained in the design of the VM. For example Chapter 5 shows how the run-time code generator can be controlled in terms of the times at which methods are compiled (not how processor registers are used), Chapter 6 shows how storage allocation policies operate at the level of objects (not machine words) and Chapter 7 shows how scheduling decisions are made by selecting between runnable threads (not by explicitly saving and restoring processor states). The one new concept introduced -- that of a *class isotope* -- has a clear analogue as a virtual method table within the VM.

Section 3.2 identified a number of areas within which an application may be able to benefit from lower-level interfaces exposed by a VM and it is, of course, possible to extend that argument to further reduce the level at which interfaces should be drawn. However, the designs presented in Chapters 5--7 present suitable facilities for many of the application-specific customizations used in those areas in other languages. These designs hopefully therefore present a pragmatic balance between portability and expressiveness. In the future it may be that the interface between UPIs and PMIs may usefully be replaced by a more easily extensible format.

4.9 Policy composition

If an application contains separately developed code modules then it is necessary to decide how to combine the different policy definitions that may be supplied with the different parts of the system. The same problem emerges when code is re-used between applications: a policy for the entire application must be developed based on the separate policies that might be defined for its constituent parts.

How these problems are resolved is ultimately the responsibility of the programmers concerned. However, the facilities provided within the XVM framework aim to provide support in particular areas.

Firstly, the design principles [P1] and [P2] aim to separate policy decisions from application code so that the decisions of code integration and policy integration may be handled separately and so that the constituent policies can be discarded if they are inappropriate in the new context.

Secondly, the decision initially to associate policies with the Java package namespace means that, in simple cases, the association between a code module and a policy definition remains in place when that module is used in another application: the package namespace is global within the JVM.

Finally, as shown in the examples in Section 6.5, policy definitions can be constructed so that a decision made for one part of the code is propagated to others. In the example a decision to place part of a data structure in one storage heap is propagated using class isotopes to objects allocated elsewhere forming part of the same structure. Such a design aims to simplify policies into a small number of 'decision' points followed by a larger number of points at which existing decisions are followed.

Of course, policy composition does remain a difficult task and one requiring programmer intervention -- but the same is already true, purely from a functional point of view, when combining existing code modules. However, ideally the separation presented here between functional aspects and policy aspects of a system makes this composition simpler than if the two aspects are entangled.

4.10 Common infrastructure

The PRs are implemented in Java using hashtables to map from sections of the package namespace (expressed as strings) to the UPIs which are associated with them. When a look-up is performed for a particular class then it proceeds using a straightforward longest-prefix-match between the fully-qualified class name and the UPI registrations.

The current implementation is simplistic because it is intended that these look-ups are performed rarely since the results may be cached on a per-class basis. The validity of cached results is currently maintained by associating a *sequence number* with each PR and with each result cached in a class structure. If the sequence numbers are equal then the cached result is valid. The per-PR sequence number is incremented when any registration is added or removed.

There are numerous more complex cache-invalidation schemes with various trade-offs between the number of unnecessary look-ups and the amount of work performed when updating the PR. The current implementation reflects the expectation that UPI registrations will occur during the initialization of an application, that the PRs will be

largely static beyond that point and that it is consequently acceptable for registration changes to bear a modest run-time cost in favour of implementation simplicity.

4.11 Discussion

The design exhibits some similarities to existing *meta-class* and *aspect-oriented* systems, described respectively in Sections 2.1.3 and 3.1.6. All three approaches provide some facilities for interposing additional processing during certain stages of the operation of a VM. In a system providing meta-classes it would be conceivable, for example, to use those as the basis for exposing control over run-time policies or object allocation – the particular meta-classes would fulfill the rôle of the UPIs of Section 4.3 and the association between the ordinary and meta levels would take the place of the PRs of Section 4.2. In such a system the run-time environment could invoke a method on a meta-class to implement policy decisions over how associated classes are handled. A similar effect could be achieved using the AspectJ framework by 'weaving' the implementation of policy decisions into the methods over which they preside.

However, there are disadvantages with the tight integration that such approaches would introduce. In particular contrast to an aspect-oriented system, the intent here is not to modify the actual bytecode executed by the VM whenever a method is invoked or an object allocated, but rather for the VM to *query* some policy definition in order to select an appropriate implementation for that operation. Such a decision may be needed at quite separate times from the actual execution of the code.

However, aside from these practical concerns, there are two conceptual problems with the use of meta-classes that motivated the formulation of the current design:

- Firstly, the association between a class and its meta-class is generally static once a class has been created. This is to be expected if classes are viewed as instances

of their meta-classes – it is uncommon in object-oriented languages to change the class of an object once it has been created [2].

- Secondly, each class is associated with exactly one meta-class: the meta-class must therefore combine the functionality required to control multiple run-time policies. It also means that the *same* meta-class must be used for all instances of the base class. Furthermore, in a language where meta-classes are already available to the programmer, any existing use of them by the application must also be incorporated. Previous work has investigated composing meta-classes but this takes the form of programming tools and conventions with which application programmers can design reusable meta-classes rather than schemes for automated composition [Mulet95, Forman94].

These two problems can both be identified as consequences of the fairly coarse granularity at which meta-class systems operate: that is, the fact that a class must be associated with a meta-class at the time of its definition and the fact that a single meta-class must be selected to provide all of the 'meta' information for its associated classes.

The design presented here addresses these problems in three ways:

- The association between run-time policies and sections of the application is dynamic: the programmer may elect whether to combine policy code within the same class definitions that implement the application, whether to provide separate policy code alongside the application classes, whether to select from policies provided by the run-time environment, or whether to defer that decision entirely to defaults. Similarly, the implementor of the PRs may choose whether to accept the policies selected by the application or whether to override them.

[2] One exception is Moostrap in which the link from an object to its meta-object can be inspected or modified dynamically [Mulet95]. In that system each method invocation is decomposed into separate `lookup` and `apply` steps and a meta-object is simply one that accepts a `lookup` message

- Separate mappings from sections of the application to run-time policies are maintained for each different aspect of the implementation that may be controlled. For example, a separate mapping would be maintained for controlling the run-time compiler from that maintained for controlling object allocation. This mapping can be controlled dynamically by modifying the registrations held by the PRS.

- The introduction of class isotopes allows the association between policies and sections of the application to be defined at a granularity finer than that of individual classes.

Chapter 5

Run-time compilation

This chapter describes how the architecture of Chapter 4 has been applied to the definition of application-specific policies for run-time compilation. The intention is to provide a framework which is sufficiently flexible to move the implementation of existing compilation policies from trusted code within the VM into untrusted code within a UPI. Desirable policies may include simple JIT compilation, or the invocation of an optimizing compiler once a method has been called a user-configurable number of times.

There are numerous granularities at which such schemes could operate. At a coarse level, a small amount of functionality could be moved into a parameterized UPI by defining a policy in terms of two threshold counts used to trigger compilation and subsequent optimization. However in keeping with the design principles of Section 4.1, and with the intent of promoting the flexibility to define more expressive policies, the split between the UPI and PMI is made at a lower level.

In general a UPI operates by receiving up-calls from the VM, implementing some form of policy decision on the basis of these inputs and then signalling that back to the VM

by invocations on an appropriate PMI. The design process for constructing UPIs and PMIs for a particular policy domain must therefore be based on the following factors:

- What kinds of information would be required in order to make policy decisions? For example, would a run-time compilation UPI require traces of every method invocation that has been made? Should the parameters being passed to the method also be made available to the UPI? Are compilation decisions made only on local data (pertaining to a single method invocation), or do the 'inputs' to the policy decision also include state retained within the UPI?
- When, and in what form, should the required data be presented to the UPI? For example, should an invocation-based interface be used in which information is passed to the UPI as parameters to a method call, or should a shared-memory interface be used in which the UPI operates as a separate thread of control?
- How should the UPI communicate its decision to the VM? In particular, does the UPI make explicit invocations on an associated PMI, does it update a data structure that is shared between the UPI and the VM, or does it return its decision as a result on an invocation-based interface?

The decisions in each case need to be guided by aspects of the policy domain in question and the existing kinds of policy that may be drawn from previous work. In the case of run-time compilation the information provided to existing policies tends to concern the execution paths through an application (for example, which methods are executed frequently) and the prominence of particular data (perhaps the class of the target object, or the parameters passed with an invocation).

The relationship between selecting appropriate run-time compilation and the basic execution process of the application suggests that policy decisions will be required *before* methods come to be invoked. For example an existing JIT policy cannot be provided unless the decision to compile is taken at the point at which the method is first executed. Equally, the definition of the background compilation policy in

Section 5.6 relies on the ability to invoke the compiler on a method at an arbitrary time after that method's invocation (in that case when the method reaches the front of a queue).

These observations suggest that controllable run-time compilation should be provided using:

- An invocation-based interface from the VM to the UPI in which the UPI is invoked as each method is called for the first time. Simple policies, such as JIT compilation, to be provided within that UPI invocation.
- The run-time compiler and optimizer should be exposed explicitly through a PMI rather than accessed implicitly by returning particular values from the UPI query. This enables the compiler to be invoked (or re-invoked) beyond the point at which the UPI returns, or for a particular invocation of the UPI to trigger the compilation of multiple methods.
- For efficiency, the VM will not in general make further invocations on a UPI for the same method. Additional facilities will be provided so that a UPI which requires further information can elect to receive it (for example, to compile a method when an invocation count reaches a threshold).

The overall structure closely follows the reference design of Figure 4.1 in that a UPI receives up-calls from the VM and implements the policy by making invocations on a PMI. The particular form taken by run-time compilation UPIs is described in Section 5.1 and the corresponding PMIs in Section 5.2.

In addition to these standard components, the infrastructure developed here provides an extended reflection interface through which a UPI may obtain additional information about the execution of application code. This is described in Section 5.3.

Section 5.4 explains low-level details of how this system is implemented within the JVM and Section 5.5 describes the compiler itself, outlining the process by which it translates bytecode into native code.

Section 5.6 illustrates some of the possible UPIs that may be defined within the framework, showing how compilation can be performed with various levels of optimization, how it can be governed by dynamic feedback over which methods are 'hot' and how compilation work can be distributed among threads. Finally, Section 5.7 summarises the main results of this chapter and discusses the applicability of its techniques to systems other than the JVM.

5.1 Run-time compilation UPI

A run-time compilation UPI is implemented by defining a class that implements the `RTCompilationUPIIfc` interface. This interface contains a single method named `implementPolicy` that the VM invokes whenever a policy decision is required for a method.

`implementPolicy` takes a single parameter indicating the method for which a policy decision is required. The method is represented by an instance of `RTPMIMethod`, which is described in more detail in Section 5.2. The policy is implemented by making invocations on the `RTPMIMethod` which change the manner in which the underlying method will be implemented on future invocations. `implementPolicy` does not itself return a result.

Once a method has been passed to a run-time compilation UPI then invocations on the associated `RTPMIMethod` may be made by the policy implementation beyond the point at which it returns. For example, Section 5.6 will present a UPI in which

`implementPolicy` returns quickly after appending the `RTPMIMethod` to a queue of methods. A separate thread compiles these methods in otherwise idle processing time.

The UPI may use the static methods of a further class, `MethodInvocationContext`, to obtain information about the location from which the method was called and about the parameters it was passed. Section 5.3 describes the rationale for this organization and the operations available on `MethodInvocationContext`.

In general, it is intended that `implementPolicy` is called on the first invocation of a method and that the implemented policy controls how that method is handled on subsequent invocations. However, there are two situations in which this is not possible. The first of these occurs when a method is invoked while it is already subject to a policy look-up. This can occur in a multi-threaded application in which one thread triggers the look-up and a second thread executes the same method before the look-up is complete. It can also occur in a single-threaded application if the implementation of the UPI invokes the method itself. In these cases the second and subsequent invocations use the bytecode interpreter until the policy look-up is complete. The second situtation occurs if, during initialization, a policy requires that the compiler compile itself. This is a problem because much of the compiler is implemented in bytecode over the JVM and, although the compiler is designed to be re-entrant, the succession of nested compiler invocations would quickly exhaust the run-time stack. In this case a limited number of compiler invocations are allowed, beyond which further invocations use the bytecode interpreter until the compilation is complete.

An important consequence of these situations is that compilation policies must not be used to implement method-based access control decisions. Some users may be tempted to define 'compilation' policies that implement schemes similar to part of the security architecture proposed by Wallach *et al* [Wallach97]. This is based on performing call-stack inspection on entry to sensitive methods in order to ensure

that they have been reached through some permissible chain of methods. If it is impossible to guarantee that `implementPolicy` is invoked on the first invocation of a method then it is inevitably impossible to guarantee that the stack inspection is performed on all invocations.

There are a number of ways in which the current implementation could be extended to provide that stronger guarantee. One would be to provide a separate `StrictRTCompilationUPIIfc` interface that would be recognised by the PR as a policy that must be evaluated on the first invocation of any associated methods. The programmer would be responsible for ensuring that the use of such policies would not exhaust the stack. Another option would be to allow recursive invocations of policy implementations and merely disallow recursive invocations of the compiler. However, this would complicate the implementation of some of the reflective features available within policy implementations – for example the `StackFrameCursor` objects described in Section 5.3.1 could not be allocated on a per-thread basis because each thread could be associated with multiple cursors.

`implementPolicy` is invoked at most once for any particular method on a given isotope – although, of course, it may be invoked multiple times for any bytecode method that exists in more than one isotope. Once it has been invoked for a particular method then the result of that policy decision is conceptually cached within the state of the `RTPMIMethod` and reused on subsequent calls [1].

[1] In practice the application-visible state in the `RTPMIMethod` is mirrored in part of the internal state of a native `methodblock` structure within the JVM implementation. If the `RTPMIMethod` is updated to wrap or to compile the method then pointers within the `methodblock` structure are correspondingly updated to refer to different entry points for the method implementation.

5.2 Run-time compilation PMI

Each instance of `RTPMIMethod` represents a method being executed by the JVM. It fulfills a somewhat similar function to the standard library class `java.lang.reflect.Method` that is part of the Reflection API [Gosling97b].

`RTPMIMethod` is maintained as a separate class for two reasons:

- Firstly, modifying classes within libraries risks introducing incompatibilities between the extensions developed here and future changes to the standardized APIs.
- Secondly, if an allocation policy defines multiple *isotopes* of some class then each of these will have a separate virtual method table. Methods in different isotopes may be subject to different compilation options and consequently they are represented by separate instances of `RTPMIMethod`. There is consequently a many-to-one relationship between these and the instances of `Method` representing the bytecode methods from which they are initially derived.

The operations available on an instance `RTPMIMethod` may be divided into four categories, listed in Sections 5.2.1-5.2.4.

5.2.1 Inspection operations

Firstly, there are operations for inspecting the state of the `RTPMIMethod`:

- `getIsotope` returns the instance of `AllocationIsotope`. This identifies the isotope to which the method belongs.
- `getSignature` returns a `String` containing the signature of the method.

- `getName` returns a `String` containing the name of the method.
- `getAccessFlags` returns an integer whose constituent bits represent the modifiers present on the method definition.
- `getClass` returns the `java.lang.Class` within which the method is defined.
- `getMethod` returns the `java.lang.reflect.Method` associated with the `RTPMIMethod`.

These operations are implemented as methods, rather than as directly-accessible fields, because the values in question are objects whose instantiation can be deferred until the `get...` method is invoked. This anticipates the design of policies in which a subset of these inspection methods will be used. Similarly, although much of this information is also accessible from the `Method` object, obtaining that object is a comparatively heavyweight operation.

One field is provided, `hash`, that provides an integer hash value representing the method. In practice this is the memory address of the internal method structure and is consequently likely to be unique within a single instance of the VM.

5.2.2 Compilation operations

Secondly, there are operations for causing the current thread to compile the `RTPMIMethod`. Compilation always occurs synchronously. There are two variants:

- `compileMethod` uses a system-default compiler configuration. It takes no parameters and returns no result. The compiler updates the VM-internal data structures so that the compiled result (if any) is used in place of the previous implementation for this `RTPMIMethod`.

- `compileMethodWith` takes an instance of `Compiler` and invokes it to compile the `RTPMIMethod`. This may be used to compile methods with particular combinations of optimization stages.

5.2.3 Assumption operations

Thirdly, there are operations for signalling that particular assumptions should usefully be made when invoking the method. In the current implementation these assumptions only concern the parameters with which the method is likely to be invoked. Any assumptions are incorporated into code generated by the run-time compiler.

Assumptions must be checked on method invocation in order to ensure safety: the policy may elect whether assumption failures are handled by throwing an instance of `AssumptionFailureError` or by reverting, for that invocation, to a separate implementation of the method generated without assumptions.

Four kinds of assumption may be made:

- `assumeArgumentHasValue` indicates that an argument (specified by its integer position in the parameter list) takes a particular value. Separate methods handle values of different types, allowing primitive values to be passed in an unboxed representation. It is anticipated that this may be used to specialize particular isotopes whose instances tend to be used in similar ways – for example the methods of a class implementing the 'singleton' design pattern will always be invoked with `this` referring to the unique instance [Gamma94]. This assumption may be employed to perform virtual-method look-up at compile time or to generate native code that accesses fields directly by their memory locations.

- `assumeArgumentIsNonNull` indicates that a particular reference-typed argument will take a non-`null` value. The assumption is made implicitly for the first argument of non-static methods because a `null`-reference check is performed as part of the instance-method call sequence within the JVM. The assumption may typically be employed to remove `null`-reference checks from inner loops within methods in cases where this cannot practicably be done within the constraints of a run-time compiler.

- `assumeArgumentIsOfIsotope` and `assumeArgumentIsOfClass` indicate that a particular reference-typed argument will refer to an instance of a particular isotope or, more generally, to a particular class. These assumptions are typically used as a looser version of `assumeArgumentHasValue` where instances share a particular isotope or class.

- `assumeArgumentIsOfType` indicates that a particular reference-typed argument will refer to an instance that is compatible with a particular type. The assumption-check follows the definition of the `checkcast` bytecode in its handling of different types and of `null` values [Lindholm97]. The assumption may be valuable where different isotopes of generic classes have been created to handle values of particular types.

5.2.4 Wrapping operations

Fourthly, there are operations for 'wrapping' a particular method. A wrapped method is one for which a further 'enclosing' user-supplied method is called before each invocation.

- `isWrapped` returns `true` if the method has been wrapped.

- `getUnwrapped` returns an instance of `RTPMIMethod` representing the enclosed method. It takes a flag to indicate whether to unwrap the method once or whether to unwrap it until it is no longer enclosed.

- `wrapWith` causes further invocations of the `RTPMIMethod` to be wrapped by a supplied instance of `MethodWrapper`. This is an interface that defines a single method `wrapperPre` to be invoked before the enclosed method. As with `implementPolicy` on a run-time compilation UPI, `wrapperPre` takes a single paramater of type `RTPMIMethod` indicating the invoked method.

The implementation of wrapping operations is therefore similar to that of run-time compilation policies. The intent is that they are used where a policy cannot be expressed directly within the single invocation of `implementPolicy` on the UPI: for example a policy that compiles methods after their first n invocations would wrap the method with one that counts invocations. Of course, the wrapper may itself be compiled. Section 5.6 will illustrate this as an example UPI definition.

5.3 Extended reflection interface

The `MethodInvocationContext` class provides information about the JVM state at method invocation sites. It may be accessed from within run-time compilation UPIs when a method is invoked for the first time and from `wrapperPre` implementations on each subsequent invocation of a wrapped method.

The implementation of `MethodInvocationContext` is important because wrappers could, potentially, be introduced on every method invocation. An original design favoured using explicit `MethodInvocationContext` objects and passing these to each `implementPolicy` or `wrapperPre` call. However, in an extreme case where each method is wrapped it would be untenable to instantiate `MethodInvocationContext` objects because the object constructor would require at least one invocation, leading to infinite recursion. In more realistic cases the allocation of individual context objects would still result in a high object allocation rate. It is also anticipated that many policies may not differentiate their behaviour

on the basis of the invocation context and so it is desirable to avoid the overhead of passing a frequently unused parameter.

Consequently, the operations on `MethodInvocationContext` are implemented as static methods: the class itself is never instantiated. They are defined using native methods that obtain information directly from the JVM state. The implementation of each of these native methods is preceded by a test to ensure that the calling thread is within a `implementPolicy` or `wrapperPre` method and that this method was invoked from the compilation-policy infrastructure.

There are two sets of operations defined on the invocation context. The first, described in Section 5.3.1 is concerned with inspecting the method call-stack at the point of invocation. The second, described in Section 5.3.2 is concerned with inspecting the parameters with which the method has been invoked.

Note that stack inspection and parameter inspection are intentionally not combined: only the parameters passed directly to the method are available. This follows from two concerns. Firstly, the parameters from other stack frames may not be available because the storage they occupied may have been updated as computation progresses in that frame. Secondly, because the security regime described in Section 4.6 is based on restricting the information available to UPI implementations associated with particular classes, it is undesirable to allow UPIs which are granted parameter access for inspecting values passed within the application to access parameters from within its callers. For example, a UPI associated with an application-supplied call-back within the Abstract Window Toolkit (AWT) graphical user interface class may be able to obtain the AWT event queue by tracing up the stack.

5.3.1 Stack inspection

The first operation on `MethodInvocationContext` is `getCallingStackFrame` which returns an instance of `StackFrameCursor` that identifies the calling stack frame. In turn, the `StackFrameCursor` provides native method operations to step up the call stack – that is, from each method back to its caller – and to obtain the name, signature, class, isotope and `RTPMIMethod` associated with each activation record.

As with the `MethodInvocationContext`, separate instances of `StackFrameCursor` are not created each time it is used. A per-thread instance is lazily allocated and this contains, in private fields, the current position of the cursor. Special values are stored in these fields to invalidate the cursor when the associated invocation of `implementPolicy` or `wrapperPre` completes. This ensures that only valid stack frames may be inspected.

5.3.2 Parameter inspection

The second set of operations on `MethodInvocationContext` are used to obtain information about the values passed to the method as parameters. These operations exist in a number of variants but in each case they take one argument indicating the position of the required parameter in the parameter list. They return a value representing the value of that parameter as their result. There is one variant for each kind of parameter type, such as `getParameterAsInt` or `getParameterAsFloat`. All reference-typed parameters are handled by `getParameterAsObject`. These operations throw an `InvalidParameterType` exception if the position does not contain a parameter of the requested type.

A general `getParameter` method can return any kind of parameter by wrapping values of primitive type using the standard wrapper classes from the `java.lang` package.

`getParameterAsHash` returns a hash value corresponding to the parameter. The implementation ensures that the same hash value will be returned whenever the same parameter is passed in the same position. Depending on the security settings it may also guarantee that the same hash value will be used for different positions on the same method, or for invocations on different methods. As described in Section 4.6 transformations may be applied to the parameters during inspection in order to control the relationship between hash values and actual parameter values.

If the inspection facility has been completely disabled for a particular UPI then the methods will throw an instance of `UPIError`: it is expected that UPI implementations will not handle this exception and that the system, on receiving abnormal termination from the UPI will use a default policy.

5.4 Low-level implementation

This section describes how invocations are made on a compilation-policy UPI and on wrapper methods. Within the existing JVM implementation, each method is associated with an 'invoker' function. These functions are implemented in native code and are responsible for updating VM state on entry to the method.

For example, when one interpreted method is called from another, the `invokeJavaMethod` invoker associated with the called method creates a new stack frame, records the interpreter program counter of the caller and updates that program counter to refer to the callee. The invoker then returns and the interpreter continues execution. This organization means that deep recursion within the Java application does not cause a corresponding series of invocations of the interpreter on the native stack. Other invokers exist for synchronized methods (in which case the invoker aquires a lock on the appropriate object) and for native methods (in which case the invoker executes the native code directly before returning to the interpreter).

Invocations on a compilation UPI are implemented by introducing a new function, termed the 'trigger-invoker', within the VM which is used as the invoker for all methods on which policy decisions have not yet been taken. When the VM creates a new isotope then, for each method, the UPI infrastructure sets it to use the trigger-invoker, displacing the original into a separate field. The trigger-invoker performs two functions:

- Firstly, it determines the UPI associated with the method. This is implemented directly as a query on the PR when the first method is invoked on any instance of each class. The result is cached on a per-class basis along with a per-PR sequence number representing the current state of the PR. Subsequent invocations on methods of the same class reuse the cached UPI look-up while the sequence number recorded in the class remains current. The PR sequence number is incremented whenever UPI registrations change.
- Secondly, it constructs an appropriate instance of `RTPMIMethod` to pass to `implementPolicy` on the selected UPI. The trigger-invoker ordinarily restores the saved invoker after `implementPolicy` has been called once on the implementation method.

The wrapping operations described in Section 5.2.4 are implemented by replacing the invoker on the affected method. The *wrapper-invoker* invokes the `wrapperPre` method on the appropriate instance of `MethodWrapper` before calling the original invoker that was associated with the method.

5.5 The native code generator

This section describes the implementation of the native code generator itself[2]. The compiler used here is straightforward: the motivation is to provide a controllable implementation rather than to establish new optimization techniques.

[2] At the time of implementation it was not possible to secure the use of an existing code generator.

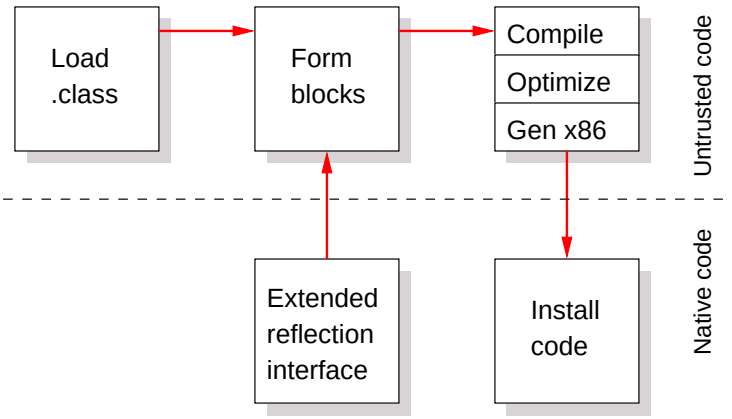


Figure 5.1: The structure of the compiler.

In addition to controlling compilation through UPIs, it is also possible to invoke the compiler directly from within an application. For the reasons argued in Section 4.1, it is not intended that this facility should be widely used: using the non-standardized API exported by the compilation PMI limits the portability of applications. However, although requiring a tight integration between the invocations on the compiler and the main portion of the application, it does enable tight control over which portions of an application are compiled.

5.5.1 Overview

The structure of the compiler is shown in Figure 5.1. There are three phases involved in the compilation of a Java method:

1. Firstly, the bytecode implementation from the `class` file is broken into basic blocks and the contents of the constant pool are resolved as described in [Lindholm97].

2. Secondly, native code is generated for each basic block. This is produced directly from the stack based bytecode operations rather than via an intermediate 3-

address-code form [Aho86]. Forward branches are handled by back-patching the generated code when the target address becomes known.

The code generator consists of three modules – a common section, a simple optimizer and a target-dependent section. The optimizer and target-dependent code generator form a layered structure with an identical interface between adjacent modules. This means that the optimizer can be removed to increase the rate of code generation at the expense of code quality.

3. Finally, the generated code is installed so that it will be executed if the method is invoked in the future.

5.5.2 Register allocation

The Intel Pentium Processor [PPro2] has four 32-bit general-purpose registers (`%eax`, `%ebx`, `%ecx`, `%edx`) and a further four 32-bit registers which may be used with some restrictions (`%esi`, `%edi`, `%ebp`, `%esp`). The compiler reserves these within generated code as follows: `%edi` refers to the current *execution environment* which contains pointers to the current Java stack frame, the current thread and the current exception (if any) that is being propagated, `%ebp` refers to the bottom of the Java operand stack, `%esp` contains the native stack pointer maintained in the conventional manner and `%esi` refers to the bottom of the current stack frame's local variable table.

The location of the Java operand stack and local variable table can be recovered through the execution environment. However they are always held in registers since their values are frequently needed.

5.5.3 Optimization

Without optimization each bytecode instruction is translated to a section of native code which is produced from a fixed template parameterized by the current depth of the Java stack. For example, the code generated for an `iload_1` instruction, when the Java stack already contains two elements, will load the value from the first local variable and then store it into the third slot of the Java stack:

```
movl 4(%esi), %eax  ; Local variable 1 -> %eax register
movl %eax, 8(%ebp)  ; %eax register -> stack slot 3
```

This approach means that the general-purpose registers will only be used within single bytecode instructions. Furthermore, the design of the JVM leads to series of bytecode instructions that transfer values from local variables to the operand stack, then manipulate the operand stack and then transfer the result back to a local variable. These factors mean that the generated code would typically contain large numbers of `movl` instructions performing potentially-unnecessary load and store operations.

Figure 5.2 shows how prevalent such instructions can be: the Java statement `r = r * m` is implemented by four bytecodes `iload_2`, `iload_1`, `imul`, `istore_2`. This structure means that simple expansion of bytecodes to series of native instructions provides very little benefit compared to an efficient interpreter, or to one using threaded code in which an operation is expressed in a semi-compiled form as a series of 'call' statements to more-primitive operations [Bell73].

The optimization layer aims to improve native code by lazily generating `movl` instructions, renaming operands and performing straightforward peephole optimizations. The optimizer maintains details of outstanding `movl` instructions which have yet to be generated. In this example, the effect of the `iload_2` and `iload_1` instructions is to record that local variable 2 has been transferred to stack slot 1 and that local variable 1 has been transferred to stack slot 2 – no native code is generated

```
public class Fact                    Method int fact(int)
{                                      0 iconst_1
  public int fact (int m)              1 istore_2
  {                                    2 goto 12
    int r = 1;                         5 iload_2
                                       6 iload_1
    while (m > 0)                      7 imul
      {                                8 istore_2
        r = r * m;                     9 iinc 1 -1
        m --;                         12 iload_1
      }                               13 ifgt 5
                                      16 iload_2
    return r;                         17 ireturn
  }
}
```

Figure 5.2: A Java method (left) and its bytecode implementation (right).

---

at this stage. These mappings can be used when generating code for the subsequent `imul` instruction – instead of accessing data in the Java operand stack it can use values directly from the local variables. A similar technique is applied within the hardware implementation of the PicoJava microprocessor [McGhan98].

The quality of the generated code is shown in Table 5.1 which compares micro-benchmark scores achieved with and without optimization. Neither the optimizer nor the interpreter will perform inter-block optimization. Measurements with larger applications (such as the `javac` compiler) show that a two-fold speedup is typical.

	Without optimizer	With optimizer
no-op	5.1	6.6
AddInt	2.9	10.0
AddLong	5.4	5.7
AddFloat	1.2	1.2
AddDouble	1.4	1.4
AddByte	2.4	5.8
AddChar	2.5	5.7
AddShort	2.4	5.8
CastToByte	6.5	10.5
CastToChar	6.1	12.0
CastToShort	5.9	11.9
CastToLong	6.0	7.5
CastToFloat	6.2	7.4
CastToDouble	4.9	5.6
CastFromFloat	1.5	1.6
CastFromDouble	1.4	1.5
MethodCall	2.5	2.8
MethodCall (2 arguments)	2.4	2.9
MethodCall (3 arguments)	2.8	2.8
MethodCall (4 arguments)	2.7	2.9
StaticMethodCall	1.1	1.2
InterfaceMethodCall	1.4	1.5
SuperclassMethodCall	1.0	1.1
SynchronizeOnThis	1.2	1.3
CatchSameMethod	1.1	1.1
CallInterpretedMethod	0.9	0.9
NewArray	5.7	6.6
ArrayAccesses	4.9	5.5
NewInstance	0.8	0.8
IterativeFactorial	3.4	8.1

Table 5.1: Microbenchmark results showing the speed-up of individual Java operations when compiled with and without optimization. Results are expressed relative to the original JDK 1.1.4 interpreter which would score 1.0 in each category.

```
public final class NullCompilationUPI
   implements RTCompilationUPIIfc
{
  public final void implementPolicy (RTPMIMethod m)
  {
    /* Nothing */
  }
}
```

Figure 5.3: A UPI that uses the VM-default implementation for its associated classes.

```
public final class JITCompilationUPI
   implements RTCompilationUPIIfc
{
  public final void implementPolicy (RTPMIMethod m)
  {
    m.compileMethod ();
  }
}
```

Figure 5.4: A UPI that performs just-in-time compilation for its associated classes.

5.6 Example policy definitions

This section illustrates some of the ways in which the PMI facilities described in Section 5.2 may be deployed in the implementation of run-time compilation UPIs.

Figure 5.3 is the simplest possible compilation UPI. It causes the VM-default policy to be used because the definition of `implementPolicy` returns without invoking any compilation or wrapping options on the instance of `RTPMIMethod` that is passed to it.

Figure 5.4 shows another trivial policy implementation. It invokes the run-

```
public final class CountedCompilationUPI
   implements RTCompilationUPIIfc, MethodWrapper
{
  public static final int HASH_SIZE = 512;
  public int[] counts = new int[HASH_SIZE];

  public final void wrapperPre (RTPMIMethod m)
  {
    int hash, count;

    hash = (m.hash >> 5) % HASH_SIZE;
    count = counts[hash] = (counts[hash] + 1) % 10;
    if (count == 0)
      {
        m.getUnwrapped().compileMethod ();
      }
  }

  public final void implementPolicy (RTPMIMethod m)
  {
    m.wrapWith (this);
  }
}
```

Figure 5.5: A UPI that aims to compile methods when they have been invoked 10 times. The `counts` array maintains a set of counters recording invocation counts for methods which hash to particular buckets.

---

time compiler, in its default configuration, for each method that is passed to `implementPolicy`. Since methods are generally passed to `implementPolicy` on their first invocation, this policy expresses just-in-time compilation.

Figure 5.5 illustrates the use of a wrapper in order to implement a policy that aims to compile methods after they have been invoked a particular number of times. The rationale for such a policy is that it allows the compiler to focus on frequently executed methods. Doing so may reduce program start-up latency or improve interactive responsiveness. No particular effort is made to avoid collisions within the `counts` array on the assumption that occasional miscompilation of rarely-executed methods is preferable to using a more computationally expensive tracking function.

As a more involved example, and with suitable support from the operating system, it is possible to bound the impact that compilation can have on the progress made by an application by arranging that compilation happens in designated *compiler threads*. This approach relies on the controllable thread scheduler presented in Chapter 7 – it cannot easily be achieved with the normal priority-based scheme. In summary, the thread scheduler allows a thread's CPU allocation to be specified using a (p, s, x, t) tuple in which p is the *period* of the thread, s is its *slice*, x is the *extra time* flag and t is the *priority*. The scheduler aims to provide each thread with s CPU time during each period p of elapsed time. If a thread has its x flag set then it is eligible for an additional allocation of the slack time in the system. Slack time is shared equally between the threads with the highest priority.

Therefore, if a thread does not receive any slack time, its CPU allocation can be used as an upper limit on the resource that it may consume and so on the impact that it may have on other concurrently executing tasks. For example, it is possible to allocate some percentage of the CPU to compilation and a separate percentage to the interpreter. This control, coupled with fine-grained thread switching, means that a user will see their application executing slowly during compilation, rather than stopping completely.

It is possible to have more than one compiler thread – for example one per application – so that the resources used during compilation can be attributed to the application that requested it. This raises a problem with classes which are shared between applications because any methods on these classes would only need to be compiled by one of the applications. However an application could be designed on the worst-case

assumption that it will always need to compile any shared methods that it uses. Additionally, the most highly-shared methods in the standard Java libraries are good candidates for off-line compilation and thorough optimization.

By varying the allocation of CPU time to the compiler thread it is possible to trade interactivity against overall performance. For example running the compilation thread entirely on extra time corresponds to compiling during idle time whereas a 100% allocation provides JIT compilation.

This trade-off is illustrated in Figures 5.6 and 5.7 which compare JIT compilation and interpreted execution against background compilation with a 5%, 20%, 30%, 50% or 75% CPU allocation to the compiler. The JVM as a whole had an 80% allocation of the CPU.

The y-axis shows the percentage of a simple benchmark that has been completed while the x-axis shows the elapsed time in cycles. The interpreter's trace shows a low, straight line which means that the benchmark is being completed slowly but at a steady rate. The JIT compiler's trace shows a steeper line with some plateaux. This shows that the benchmark is being completed more rapidly but that there are pauses during which no progress is made at all. These pauses correspond to sections of the benchmark in which new methods are executed, triggering compilation.

If a background compiler is given a small (5%) CPU allocation then the trace remains steady and is even shallower than that of the interpreter. This is explained by the fact that the compiler is operating slowly and fails to finish compiling methods before execution shifts to another part of the benchmark.

As the CPU time allocated to the background compiler is increased, performance approaches that of the JIT compiler. Note that unlike the JIT compiler, which pauses whenever a new method is encountered, the systems using background compilation merely slow down.

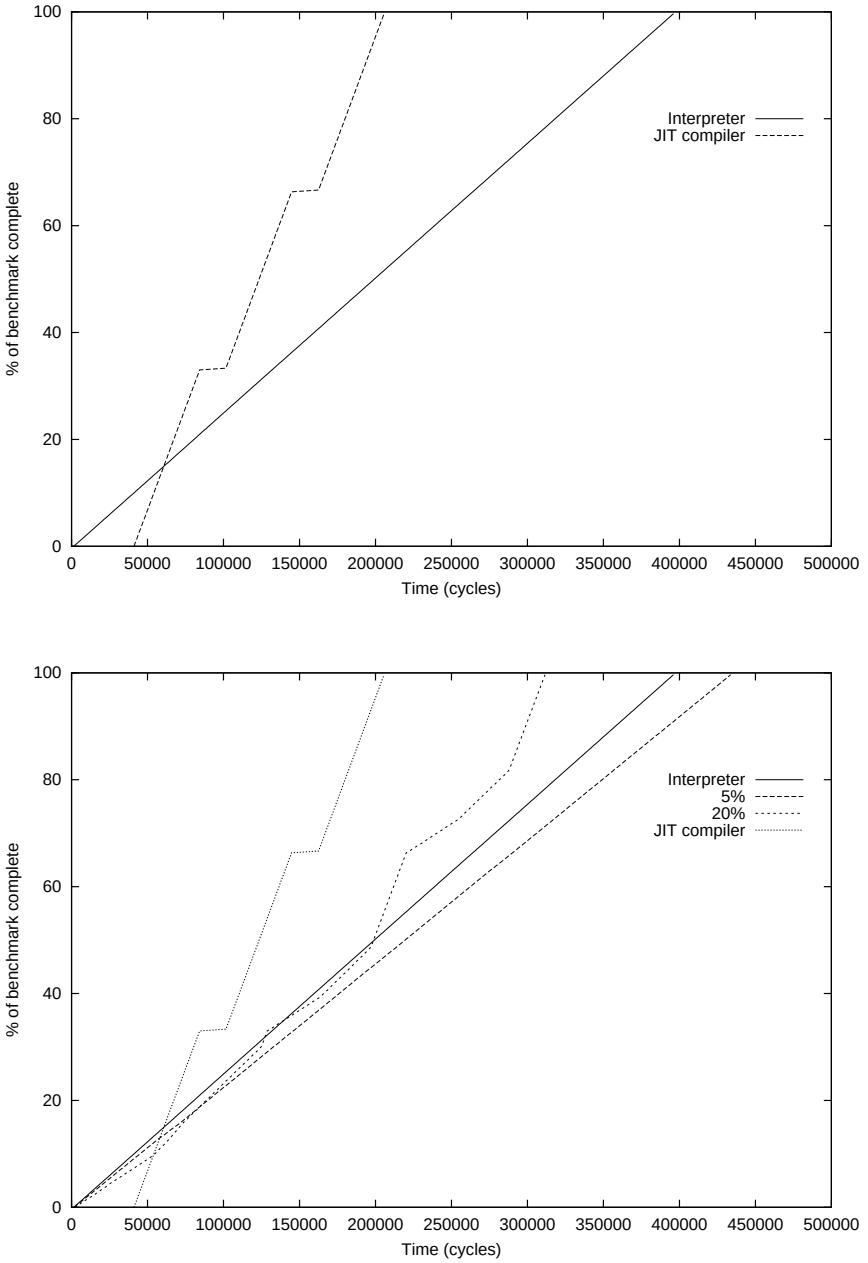


Figure 5.6: JIT compilation and interpreted execution (top), background compilation with 5% and 20% CPU allocation (bottom).

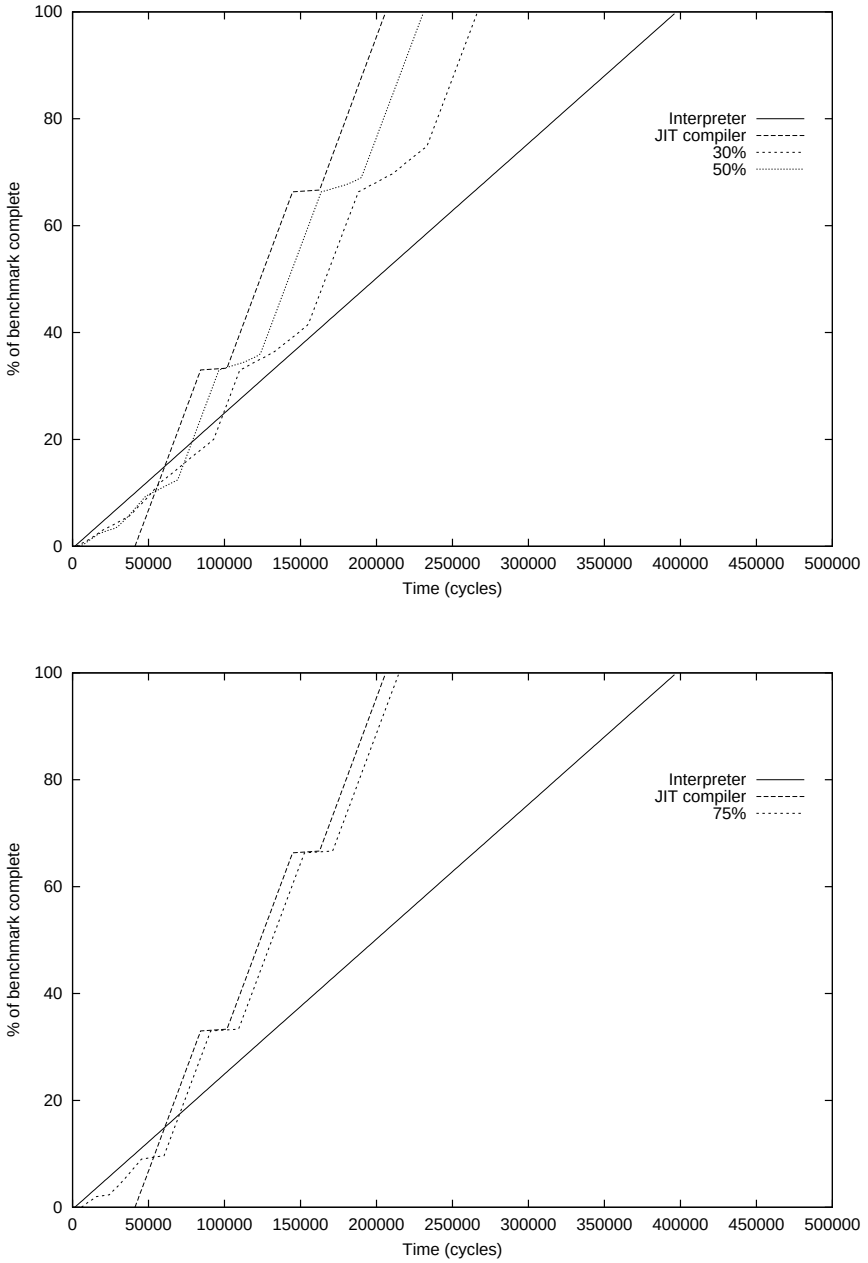


Figure 5.7: Background compilation with 30% and 50% allocation (top), and 75% allocation (bottom). The JVM as a whole was allocated 80% CPU time.

5.7 Discussion

This chapter has introduced the use of the framework of Chapter 4 for the policy domain of run-time compilation. The system presented here, in which policy decisions are made within UPIs and then effected by invocations on PMIs, has been shown to be flexible enough to allow existing policies to be expressed (such as JIT and threshold-based compilation) in addition to the new policy of background compilation.

Although the implementation is based substantially on features of the JVM – such as using method invocation to trigger UPI queries – it appears suitable for deployment in other environments, at least without architectural modifications. One notable area in which the precise details of the interfaces would need to be modified is the way in which optimization parameters are passed to the compiler through invocations on the PMI, as described in Section 5.2.3. In particular, the operations that define checked assumptions for the optimizer all use terminology specific to the JVM and the Java programming language.

Consequently, a potential area for future work is the extent to which policies wishing to make such assumptions may be defined in a way that is more language-neutral. For example, if an application was originally implemented in Scheme and compiled to Java bytecode using the Kawa compiler (described in Section 3.1.2), could its associated policies also be defined in Scheme?

The implementation presented here has adhered closely to the design requirements of Section 4: default policies can be used for applications without particular requirements, naïve or premature optimization can be removed by interposing tests on the application's invocations on the PR and the example policy definitions of Section 5.6 are all reusable between applications. Although the exposed interface to the native code generator is safe, the potential for covert channels based on the up-calls to the UPIs means that security concerns must be managed explicitly when the VM is configured.

Chapter 6

Memory allocation

This chapter describes the application of the extensible virtual machine architecture to the problem of defining policies for application-specific storage allocation. As with the run-time compilation policies described in Chapter 5, the primary motivation for allowing an application to control its heap management is that different applications are suited to different management schemes. Section 3.2 illustrated this with reference to existing surveys of benchmark applications and their performance under the control of different storage allocation or object placement strategies.

However, when compared with run-time compilation, differences between the two problem domains necessitate changes to the interface between application-supplied policies in the form of UPIs and the facilities exposed by the VM as PMIs.

Firstly, the allocation of objects within the heap is a dynamic process: an object may be allocated at any time during the execution of an application, whereas run-time compilation may be expected to reach a fixed state once all of the application has been converted to appropriate native code. If the UPI were to be queried on each allocation then the frequency of policy invocations would be very high indeed.

Secondly, the manner in which object allocations are made is crucial to the type safety of the programming language and to maintaining the expected semantics of the bytecode operations.

The remainder of this chapter considers the impact of these issues and presents the design of the corresponding UPI (Section 6.1) and PMI (Section 6.2). As with run-time compilation, additional contextual information is made available to aid UPIs. This is described in Section 6.4, along with how the UPI implementor may use this to group related objects into *isotopes*. Section 6.5 illustrates how this framework may be used to express practical policies. Finally, Section 6.6 discusses the possibilities for introducing untrusted allocation *mechanisms* in addition to untrusted allocation policies.

6.1 Storage allocation UPI

The high frequency of allocation requests suggests that it is inappropriate for the UPI to be queried each time an object is instantiated.

One possible approach, albeit at a coarse granularity, would be to provide VM configuration parameters to set the size of the heap and to select – at the level of the entire system – between candidate allocation policies. This would provide the same level of control that was examined in Wilson *et al*'s survey of allocation in benchmark applications [Wilson95]. However the same arguments made there, in relation to whole-program behaviour, apply to behaviour that changes within individual application runs – either temporally between different phases of execution, or spatially within the program code. A straightforward example of the latter is when multiple applications operate within the same VM.

The selected approach strikes a balance between the two extremes in that the storage management UPI is invoked whenever a policy decision is first required for each allocation site within the application; the resulting decision is thereafter integrated into the implementation of that operation. This can be performed either by inlining

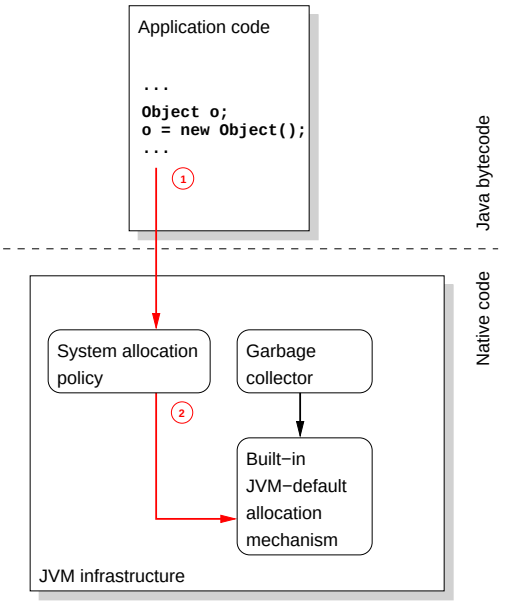


Figure 6.1: Overview of memory allocation within the unmodified JVM implementation.

an appropriate code fragment when the allocation site is compiled to native code, or by replacing the allocating bytecode with an ordinarily-reserved value that summarises the mechanism selected by the UPI [1].

Figure 6.1 illustrates the entities involved in a memory allocation within the unmodified JVM implementation. The untrusted *application code* expresses allocation requests, typically generated from `new` expressions in the Java programming language. These requests are handled by the *system allocation policy* implemented in native code within the JVM. In simple JVM implementations this policy may cause all allocations to be made within a single area of memory – illustrated in the figure by performing all allocations with a single default mechanism. As described in Section 3.2.2 contemporary systems may use feedback-directed techniques to segregate different kinds of object.

[1] Inevitably the use of such bytecodes requires careful co-operation between any possible extensions to the internal instruction set used by the VM. However, the precise value chosen becomes less important where execution time is dominated by compiled code rather than the use of an interpreter. A possible extension to this work – if multiple policy domains contend for 'spare' bytecodes – would be for their allocation to be controllable from the application.

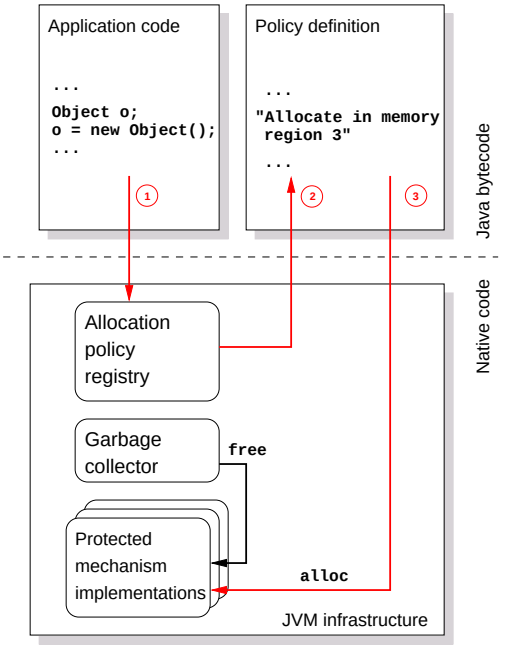


Figure 6.2: Overview of the implementation of an application-specific storage policy. The red arrows indicate the path conceptually taken when an application attempts to perform an allocation.

Such a system would contain multiple allocation mechanisms between which the system allocation policy would select. De-allocation operations are invoked directly by the garbage collector. Furthermore, in addition to the interactions shown here, the allocation mechanism may provide status information to the allocation policy – for example to indicate whether some portion of the heap it manages is nearly full.

Figure 6.2 shows how this structure can be extended to support memory allocation UPIs. As in Chapter 5 the PR maps from portions of the application bytecode to instances of policy implementations. For example, a particular policy instance may be associated with the classes `java.lang.*` and a different policy with the more specific classes `java.lang.reference.*` or the individual class `java.lang.String`.

It is important to realise that these mappings are based on the class *within which* the allocation is being made rather than on the class of the allocated object. The rationale for this decision is as follows:

- Firstly, it reflects the intuition that allocation policies will be designed based on the manner in which objects are *used* and therefore perhaps on the context in which they are allocated rather than according to the class being instantiated. Earlier work on feedback-driven storage management supports this view by showing that the allocation site of an object provides more worthwhile optimizations than merely considering the class involved [Barrett93, Zorn98, Harris01].

- Secondly, it simplifies the modular composition of policies. For example, consider the Java utility class `java.util.Hashtable` and suppose that there may be some hashtables that are accessed frequently and other hashtables that are accessed infrequently. Organizing the PR according to the *allocating* class may enable this distinction to be made during that initial look-up. In contrast, if the PR was organized according to the *allocated* class then the distinction must be made later within the policy implementation. Furthermore the policy implementation used for `Hashtable` would need to be updated whenever new uses of that class are introduced.

6.2 Storage allocation PMIs

The abstract class `AllocationPMI` is used to represent the various PMIs available to storage-management UPIs. The allocation mechanisms themselves are not actually implemented as bytecode within these classes. Instead each of the pre-supplied PMIs contains an identifier which selects between the implementations available within the VM. Section 6.6 will discuss the extent to which these existing allocation mechanisms may be augmented with untrusted low-level allocators supplied by the programmer.

It is important to distinguish clearly between the entities represented by each *sub-class* of `AllocationPMI` and by each *instance* of such a sub-class:

- Each class descended from `AllocationPMI` represents a particular kind of allocation regime. For example a class `VMDefaultAllocationPMI` represents the kind of heap that was implemented in the unmodified VM implementation. Separate classes are provided to expose first-fit placement, best-fit placement, irrecoverable allocation and the use of fixed-size memory blocks.

- In contrast, each *instance* of one of these classes represents a heap that is available to service allocations. It may be parameterized, at the time of instantiation, with class-specific configuration settings such as its size. Therefore, by returning an instance of `AllocationPMI`, the UPI is identifying both the heap in which the allocation is to be placed and the manner in which an appropriate block is to be selected within that heap.

6.3 Expressing allocation policies

To illustrate this scheme, consider how it may be used to express a policy which segregates long-lived and short-lived objects. Such a scheme has been found to be effective for benchmark Java applications [Harris01].

One possible way of organizing this would be to use the PR to map all allocations to a single instance of an allocation policy. That policy implementation would select between two allocation mechanisms corresponding intuitively to 'allocate long-lived' and 'allocate short-lived'. This organization, using a single policy instance, is illustrated in Figure 6.3. In practice the two mechanisms shown there may simply correspond to two distinct heaps with the goal of reducing fragmentation caused by the space occupied by dead short-lived objects between long-lived neighbours – perhaps segregating objects based on the results of run-time sampling using the mechanisms proposed by Agesen and Garthwaite [Agesen01].

However, suppose that certain portions of the application are known to allocate long-lived data structures – based on either the programmer's knowledge or on static

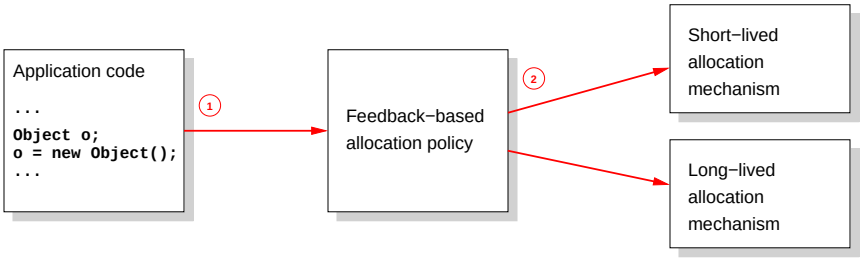


Figure 6.3: Using a single allocation policy for the entire application.

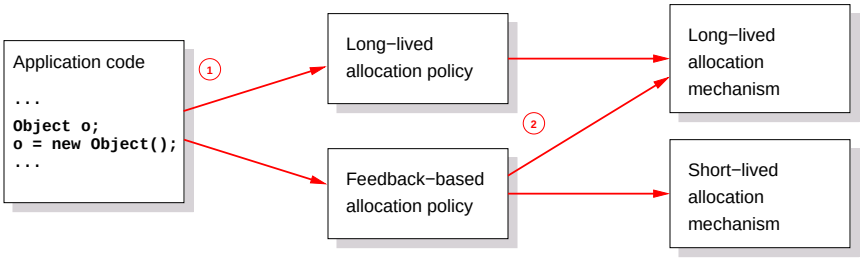


Figure 6.4: Using multiple allocation policies so that sections of the application with well-known behaviour can be handled with simpler policy look-ups.

---

feedback received from previous application runs. Such structures could reasonably be handled directly as long-lived without the overhead of performing run-time feedback.

There are several ways in which such a modification may be made to an existing policy:

- The feedback-based policy could be updated to detect the allocations in question. This approach lacks flexibility because it requires that the existing policy definition is available in a form that permits modification. Even where the existing implementation may be modified, introducing special cases into an existing policy may limit the extent to which the same policy may be reused with different applications.

- A layered implementation could be developed by implementing a new policy to handle special cases before forwarding any other allocations to an instance of the unmodified feedback-based policy. However, the additional tests necessary to detect special cases may harm overall performance. This is particularly likely if there are numerous special cases which develop a long chain of policy implementations.

- The mapping function of the PR can be used to associate different policies with different parts of the application. This organization is illustrated in Figure 6.4, which shows how two allocation policies can be used: one to handle the long-lived objects directly and one to use run-time feedback for all other objects.

Look-ups of policies within the PR are performed on a per-class basis, allowing look-up results to be cached with other per-class information. Of course, it is possible that the desired separation of policies may not correspond cleanly with the organization of code into classes. Such a situation can be resolved using layered policies for just the overlapping sections of the application.

6.3.1 Invocations on memory allocation UPIs

An allocation UPI is implemented by defining a class that implements the Java interface `AllocationUPIIfc`. This interface contains a single method named `getAllocationPMI` that is invoked by the VM whenever a policy decision is required for a particular allocation site. In this case the PMIs that are available correspond to a variety of popular schemes such as first-fit placement (that is, selecting the first block of free space that is big enough), best-fit placement (selecting the smallest block of free space that is big enough), irrecoverable allocation (where blocks cannot be reclaimed once allocated, reducing housekeeping overheads) and allocation within a pool of fixed-size memory blocks (where all blocks within a heap are the same size, again reducing housekeeping overheads).

The behaviour of a UPI is implemented by selecting *between* these candidate allocators, rather than by making direct invocations on them. Each object allocation must be made at most once and this constraint is easily enforced by representing each PMI as a separate object and requiring that `getAllocationPMI` returns a reference to the selected object. If a `null` reference is returned then allocations made at that site are considered to always fail and an instance of `OutOfMemoryError` is thrown if the site is reached during execution.

6.3.2 Timing of policy decisions

As with compilation policies, it is intended that `getAllocationPMI` will be invoked upon the first time an allocation is made at a particular site.

In the current implementation, when the interpreter is used, the first allocation made from a particular bytecode attempts to make the policy decision. The bytecode is subsequently re-written to identify the PMI that was selected. In many cases this is achieved by replacing the `new` bytecode with extra variants, `new_0`, `new_1`, ..., `new_4` which refer to the first five distinct PMI instances encountered within the application. If further heaps are encountered then a general-purpose `new_x` bytecode is used and the selected mechanism is indicated out-of-line by replacing the constant pool index that followed the `new` bytecode. This index refers to a new constant pool entry that identifies both the selected mechanism and the same class as the original bytecode.

In the current implementation, for reasons of application portability, these replacement bytecodes are used only within the implementation of the VM interpreter: the bytecode verifier will reject them if they occur directly in the application classes. It is, however, conceivable that simple allocation policies could be supported by allowing the application programmer (or bytecode-generating compiler) to use a range of `new` bytecode operations in order to identify different heaps. Bytecode-compatibility with unmodified VMs could be maintained by selecting between different `new` operations

in auxiliary tables. These would be distributed with class definitions and identify allocation sites by their position within the bytecode.

If a method is subject to run-time compilation then policy decisions are made for allocation sites encountered at that point. If the method has already been executed by the interpreter then the decision made then is carried over into the native code generated by the compiler. The decision cannot be deferred practicably until the method is executed so that the code that implements the selected PMI may be inlined within the generated native code.

As with compilation policies, the situation is relaxed somewhat during the execution of multi-threaded applications. When performing a look-up operation on a compilation UPI any concurrent invocations of that method default to using the interpreter. In contrast, in this case, each thread performs a *separate* policy look-up for the allocation site and attempts an atomic *compare and swap* processor instruction to update the bytecode with the selected mechanism. This reflects the fact that allocation UPI look-ups are expected to be simple idempotent operations, depending only on the allocation site and allocated class. In contrast a compilation UPI is expected to be non-idempotent and, if the compiler is invoked, longer running.

6.4 Grouping objects into isotopes

As described in Section 6.3.1, a storage management UPI is provided with access to information about each allocation site for which a policy decision is required. Such an allocation context performs two functions:

- Firstly, it provides information about the position within the application for which an allocation-policy decision is required. It is the analogue, for storage allocation UPIs, of the `MethodInvocationContext` class described in Section 5.3.

- Secondly, it provides control over the isotope of the class being allocated. Isotopes were introduced in Section 4.5 as an organization structure between individual objects and classes to allow the programmer to identify groups of objects, instantiated from the same class at the same allocation site, but which should be handled according to different policies at run-time.

As with run-time compilation UPIs, the first of the facilities is provided using a per-thread instance of `StackFrameCursor` to inspect the path through the code that led to the allocation site [2]. When a policy look-up is made during compilation (and therefore no example call stack is known) the contextual information available is inevitably limited to a single frame describing the method containing the allocation site.

The second facility is provided by a further method on an allocation context which sets the *isotope tag* associated with a particular allocation site. It is intended that policies may create isotope tags representing properties such as '*long lived object*' or '*frequently accessed object*'. In the JVM each allocation site generates instances of exactly one class and so the combination of this class and the isotope tag identifies the specific class isotope to instantiate. Class isotopes are lazily created within the VM and are implemented using per-isotope virtual method tables. Internally, isotope tags are identified using object equality between instances of `AllocationIsotopeTag`.

6.5 Example policy definitions

This section illustrates some of the ways in which the PMI facilities described in the previous sections may be deployed in the implementation of storage management UPIs.

[2] Separate per-thread instances of `StackFrameCursor` are used for each kind of UPI in case a storage-allocation look-up is performed within a compilation policy or *vice versa*.

```
public final class NullAllocationUPI
   implements AllocationUPIIfc
{
  AllocationPMI def =
     VMDefaultAllocationPMI.theVMDefaultAllocationPMI;

  public final AllocationPMI
     getAllocationPMI (Class c)
  {
     return def;
  }
}
```

Figure 6.5: A simple storage management policy that selects the default allocation mechanism for all requests.

Figure 6.5 shows the simplest useful allocation policy in which the VM-default allocation mechanism services all requests. The heap takes a default size according to parameters specified at the time the VM was started.

Figure 6.6 shows another trivial policy implementation. It creates a separate heap of 10MByte in size. As before, this heap is managed by the VM-default allocation mechanism. Such a policy may be defined in an attempt to limit the amount of storage space allocated by a particular portion of code within the VM – for example separate instances of `SepHeapAllocationUPI` could be used for each applet running within a shared VM. However, the policy is ineffective because, if registered through the PR with the classes implementing an applet, it will only handle objects allocated directly within those classes. Objects allocated on behalf of the applet in the standard libraries will be handled through the separate policy associated with that library code.

```
public final class SepHeapAllocationUPI
   implements AllocationUPIIfc
{
  AllocationPMI heap =
     new VMDefaultAllocationPMI (10 * 1024 * 1024);

  public final AllocationPMI
     getAllocationPMI (Class c)
  {
     return heap;
  }
}
```

Figure 6.6: A storage management policy that places allocations made in classes associated with it into a separate heap.

Figure 6.7 illustrates one way of addressing this problem. It is a policy that might be applied to the standard libraries shared between applets. It assumes that a separate policy, similar to `SepHeapAllocationUPI`, is applied to the implementations of the applets themselves and that this sets the isotope tag *applet objects* on all of the allocations made within the applet. It might, alternatively, apply isotope tags on the basis of the thread making the allocation. `TransHeapAllocationUPI` propagates this applet tag from objects allocated directly in the applet to any further objects allocated within the libraries.

For example, it propagates the isotope tag from an instance of `Hashtable` (allocated directly by the applet) to the instances of `HashtableEntry` (allocated within the implementation of that table). These objects are allocated within one heap while other objects allocated within the standard libraries are placed in the ordinary heap.

```
public final class TransHeapAllocationUPI
   implements AllocationUPIIfc {
  AllocationPMI heap =
     new VMDefaultAllocationPMI (10 * 1024 * 1024);

  AllocationPMI default =
     VMDefaultAllocationPMI.theVMDefaultAllocationPMI;

  AllocationIsotopeTag ait =
     new AllocationIsotopeTag ("applet objects");

  public final AllocationPMI
     getAllocationPMI (Class c) {
     StackFrameCursor     sfc;
     AllocationIsotopeTag sfait;

     sfc = AllocationContext.getAllocatingStackFrame ();
     sfait = sfc.getAllocationIsotopeTag ();
     if (sfait == ait) {
         AllocationContext.setAllocationIsotopeTag (ait);
         return heap;
     } else {
         return default;
     }
  }
}
```

Figure 6.7: A storage management policy that propagates the isotope tag *applet objects* for one object to all of the objects allocated by its methods. For example, if a `Hashtable` allocated directly in the applet code is created with that tag then it will be passed on to all of the hashtable buckets allocated within the `Hashtable` implementation.

The combination of the `SepHeapAllocationUPI` policy along with the `TransHeapAllocationUPI` definition does not provide a complete solution to the problem of controlling resource usage within a VM. For example, suppose that an array is allocated within a library implementation, filled with data received from the network and then passed to the applet. This buffer would be placed in the ordinary heap because, at the time of the allocation, it is not clear to which applet its resource usage should be accounted. Further, the examples presented here do not allow storage allocations to be 'handed off' when their conceptual ownership changes. These remaining problems relfect those faced in other contexts where resource management is attempted in the presence of sharing. The dilemma of accounting usage within tasks operating on behalf of other resource principals motivated the architects of the Nemesis operating system to develop a structure within which shared tasks could be avoided.

6.6 Discussion

The facilities presented in this chapter allow untrusted policy implementations to select – on a per-allocation-site basis – between different heap storage mechanisms. The resulting framework has been illustrated by implementing a policy to segregate objects with different access characteristics and a policy to ensure particular allocation mechanisms are used for different parts of an application. Furthermore, the use of isotopes as an intermediate organizational structure between individual objects and classes allows the UPI implementor to provide separate policy decisions for different objects produced from the *same* static program point.

As with the work of Chapter 5, the framework presented here fits well with the design requirements identified in Section 4 in that it allows both general-purpose and application-specific policies to be defined and reused and permits the retroactive correction of policy decisions that become unsuitable.

However, the systems presented here still rely on the VM-supplied PMIs in order to implement heap management policies. This is clearly visible in the design of the interface between the VM and each policy implementation: the UPI returns an object identifying one of the available PMIs, rather than making the actual allocation under its own control. An interesting extension of this work is therefore the extent to which it may be possible to support untrusted allocation mechanisms in addition to these existing ones.

The difficulty in providing support for untrusted allocation mechanisms comes from the fact that it is hard to distinguish usefully between *safety* criteria that are sufficient to support the security model of the VM and *correctness* criteria that require allocation and de-allocation operations to have their usual semantics.

Informally, a correct implementation of an allocator must be guaranteed to operate safely and to return the address of a block of memory that may contain an object of a requested size. However, the definitions of *contain* and *operate safely* depend on the kinds of ill effects which are to be prevented. A programmer using a lower-level programming language such as C may expect that a `malloc` implementation returns a currently-unused and appropriately sized block of memory that has been allocated to them by the operating system. If the programmer implements a new version of `malloc` which erroneously corrupts some other part of the process' state then, in the context of C, the isolation between user-level processes will be provided by the hardware protection mechanisms under the control of the operating system. Although the programmer may be concerned that their `malloc` invocations have unforeseen side-effects, the system administrator need not be.

```
class Original
{
 int ival;
}
```

```
class Spoof
{
 Object oref;
}
```

Figure 6.8: If an instance of `Original` can be converted into an instance of `Spoof` then arbitrary integer values could be cast unsafely into object references.

The constraints on the implementation of an object allocator within a type safe VM are more stringent. This is because the software-based protection mechanisms used within the virtual machine depend on the allocator for their own validity. Consider, for example, the class definitions listed in Figure 6.8. If the allocation function allowed the same area of memory to concurrently hold instances of `Original` and `Spoof` then changes to the `ival` field in the instance of `Original` would in all likelihood update the `oref` field in the instance of `Spoof`. This would provide a mechanism for constructing arbitrary object references from integers. Similar problems would occur if the `Spoof` class presented fields with more permissive access modifiers than those of `Original` – for example substituting a `public` field for one originally defined `private`.

It might be tempting to suggest that such 'spoofing' is not a problem for certain data structures – for example between classes forming part of the same application and containing only scalar fields or arrays of scalar types. There are at least two reasons why this is not true. Firstly, the typical representation of an instance of a class such as `Original` will still contain pointer values even if all of the language-level fields are of scalar types – at the very least it will probably contain information about the object's class through which its method table is accessed. Arrays will contain their length for bounds tests. Secondly, the integrity of some checks made within library code may rely on the immutability of (among other examples) strings. In the JVM this immutability is ensured by the implementation of `java.lang.String` which copies the contents of a proposed string into an object-local array of characters.

Effectively the allocator must be guaranteed to return a block of memory that is sufficiently large to hold the proposed object, the block must lie in some portion of an area previously supplied to the allocator's control, and no portion of the returned block of memory may already be used by another allocated object.

There are other properties which may be desired of an allocator but that are not important from the point of view of run-time safety – for example the allocator should not squander memory by returning large blocks for small requests, its implementation should terminate and perhaps it should do so in a predictable amount of time. As argued in Section 3.3 the selection between such goals is the kind of tradeoff which motivates support for application-specific storage management. The choice of when to reuse free storage and when to request new space from a downstream allocator is again the kind of policy decision that is being exposed to the untrusted programmer.

6.6.1 Untrusted deterministic functions

The problem of allocating blocks of memory to objects is superficially similar to that of allocating disk blocks to files. As described in Section 2.2.1 the Xok/ExOS exokernel implementation handles that latter problem by associating an untrusted deterministic function (UDF) with each different meta-data format [Kaashoek97]. Meta-data values represent, for example, the blocks allocated to a particular file. The UDF maps these values onto a common 'set of blocks' format. Updates to the meta-data value can be checked by comparing the results of the UDF before and after the proposed change to the values passed to the UDF: the determinism of the function ensures that the set of allocated blocks cannot depend on other contextual information.

It is not clear whether UDFs provide an effective solution to the problem of managing filesystem meta-data within untrusted code. However, significant differences between that environment and the management of an in-memory heap mean that UDFs do not provide an effective solution here. An attempt to use UDFs to support untrusted memory allocators would require functions which enumerate the free and allocated

areas of the heap: the trusted infrastructure would ensure that the allocation mechanism returns storage space which it previously claimed was free and that it subsequently claims that this space is allocated (without claiming that any other allocated space has become free). This suggests that the data passed between the UDF and the trusted infrastructure would either be O(number of allocated objects) or O(size of heap), irrespective of the size of the particular allocation being made. In contrast, the UDFs previously used within filesystems must merely ensure that the meta-data associated with a particular file is updated to include (or exclude) a specified block.

6.6.2 Linear objects

Baker proposes using *linear objects* as an alternative approach to implementing safe untrusted resource managers [Baker95]. Linear objects must be manipulated in such a way that exactly one reference to them exists at any time. For example, if two variables refer to linear objects then it would be valid to swap the contents of the two variables (preserving one reference to each object) but not to assign the value of one variable to the other (constructing two references to one object and leaving the other unreferenced). Linear objects are inspired by *linear logic*, introduced by Girard [Girard87], in which a proof must use each assumption exactly once. Scedrov presents a survey of that topic [Scedrov95].

Linear objects can be used to naturally model some forms of resource: maintaining exactly one reference to the object reflects the fact that ownership of the resource can be passed between data structures during processing but it cannot, in general, be used twice or discarded at will. Linear objects can also aid optimization because they cannot be aliased and they cannot be accessed concurrently by multiple threads.

In the context of the current example, if linear objects are used to represent blocks of free memory, then the allocation policy which manipulates those blocks would be unable to retain a reference to a block after it has been returned by `malloc` and it

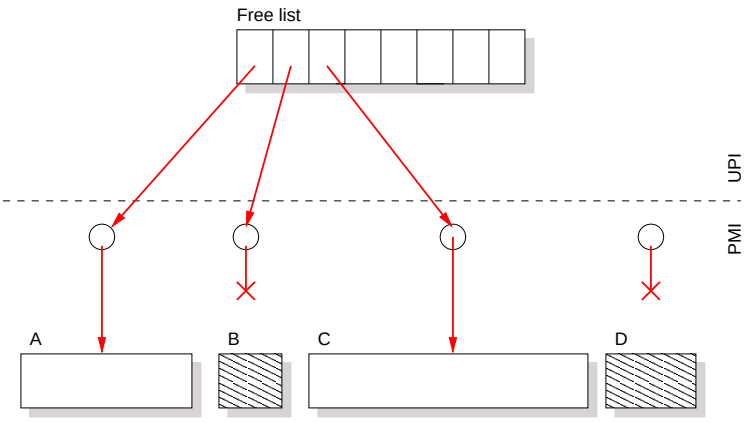


Figure 6.9: Simulating linear objects using another level of indirection. The allocation mechanism in the UPI can access blocks A and C, whereas access to B and D has been revoked.

---

would also be unable to generate references to blocks other than those supplied by the downstream allocator.

Such a primitive scheme does prohibit the expression of many useful allocation policies – such as those which may fragment large blocks in order to satisfy smaller requests, or those that coalesce adjacent blocks in order to satisfy larger requests. These common operations, observed in many existing memory allocation policies, must be provided as trusted operations on blocks.

One possible approach to implementing untrusted allocation mechanisms is therefore to provide a PMI exposing operations to manipulate blocks of memory and to introduce an additional level of indirection to ensure that the UPI can only make a single allocation in each block. This effectively provides a similar scheme to Hawblitzel *et al*'s definition of *capability objects* in which the indirection is used to support revocation in the J-Kernel [Hawblitzel98].

Figure 6.9 shows how this may conceptually be organized. The trusted PMI implementation is able to revoke access to blocks of memory (in the figure, B and D) once they have been allocated or divided into fragments. However, a straightforward implementation of such a scheme appears unsatisfactory for a very frequent operation such as object allocation: each intervening capability object would itself require storage space and each indirection through the object would introduce a run-time penalty over direct access.

Future work hopes to investigate two possible techniques for a more effective implementation:

- Whether extensions to a run-time compiler could be used to inline the PMI implementation within the UPI and, while doing so, identify where memory blocks may be manipulated directly rather than through indirection. Wu *et al* describe a similar technique of *semantically inlining* optimized definitions of library functions (such as arithmetic on complex numbers) within user-supplied methods [Wu98]. Welsh and Culler similarly inline accesses to VIA network interface hardware [Welsh00]. The important difference between these techniques and ordinary method inlining is that the inlined code is supplied from a trusted optimized implementation handled as a special case by the compiler, rather than derived directly from the bytecode implementation.
- Whether a separate low-level bytecode language would be more suitable for expressing this kind of policy. The initial proposal for this language advocates an extensible approach in which different problem domains can use different sets of bytecode operations [Harris99]. Operations would describe both their run-time implementation and their effect on the abstract state at verification time. In the case of memory allocation they could use verification-time reference-counting to ensure that memory blocks remain uniquely referenced.

Chapter 7

Scheduling

Chapters 5 and 6 described the specialization of the XVM architecture to the problem domains of controlling run-time compilation and storage allocation. This chapter introduces a third kind of policy definition by showing how application-specific thread scheduling can be supported. As Section 3.2 identified, it is easy to see how the decisions made by the thread scheduler affect the performance that a user experiences from an application – multi-threaded interactive applications favour a high rescheduling rate to give responsive performance whereas batch-mode applications favour a low rescheduling rate to improve overall throughput through a reduction in the number of thread switches performed.

The efficiency of the designs presented in previous chapters both rely on reducing the number of up-calls made by the VM to an Untrusted Policy Implementation (UPI). This was necessary because the underlying rates of method invocation or object allocation can be high, making it unreasonable for policy queries to be made so frequently. In the case of run-time compilation, the rate of UPI invocations is reduced by invoking the UPI on the first occasion that a method is called and allowing the UPI, at that stage, to elect whether to receive information about subsequent invocations. In the case of storage allocation, the UPI was invoked only on the first execution, or compilation, of each allocation site. The writer of a policy could differentiate

between kinds of allocation made at the same static allocation site by causing different instances of the class containing the site to be placed into different isotopes.

In each of those problem domains a UPI may be queried once and its decision subsequently cached within the VM for reuse on subsequent executions of the code concerned. However, schemes used for providing controllable scheduling tend to require a policy decision on *every* occasion where rescheduling may occur. This is true both with the approach based on *scheduler activations* that the Nemesis operating system takes (described in Section 2.2.2) and with the recursive CPU-donation approach taken in Fluke (described in Section 2.2.3).

The SPIN operating system allows application-provided scheduling policies to be implemented by downloading code from user-space into the kernel [Bershad95, Sirer97]. The rationale is to avoid the perceived overhead of the up-calls made with scheduler activations while retaining the flexibility that they provide. The application-specific thread management code is termed a *strand package*, and is written in a type safe language and compiled with a trusted compiler. In their implementation, Sirer *et al* used Modula-3. The strand package receives up-calls from the kernel when a policy decision is required.

In many functional languages, *continuations* have been used to share processing time between different tasks – Wand's system based on Scheme, and Reppy's for an ML-derived language, are typical [Wand80, Reppy91]. A continuation appears to the programmer as a function of one argument which, when evaluated, returns control to the point of its creation.

Wand shows how a simple thread scheduler can be implemented using a set of continuations, each of which represents one of the threads that is currently runnable. A processor resumes a thread by evaluating its continuation and a thread yields by creating a continuation at the point of its suspension and placing that on the set eligible for scheduling. The same abstraction can be used to implement a

preemptive scheduling policy if a timer interrupt creates a continuation on behalf of the interrupted thread. Queues of continuations can be used to arbitrate the order of resumption for threads blocked on semaphores. Reppy uses similar implementation methods in CML [Reppy91], based on first-class continuations in an extended version of SML/NJ [Duba91].

7.1 Design overview

In order to support application-supplied thread schedulers as part of an extensible virtual machine, the approach taken here combines aspects of scheduler activations, SPIN strand packages and continuation-based systems. Conceptually, the VM makes up-calls to a user-supplied scheduler in much the same way as with scheduler activations. The safety of the resulting system is ensured through a combination of language type safety and checks in the VM on each scheduling decision. However, as with SPIN, an efficient implementation is realized by compiling the application-supplied scheduler to native code into which those checks may be inlined.

The system described here operates over Nemesis and in fact application-supplied thread schedulers operate in a similar manner to a User-Level Scheduler (ULS) in the underlying OS: saved thread states are held in *context slots* and the scheduling policy is responsible for choosing between these. Similarly, the operations exposed by the VM through the thread-scheduling PMI are analagous to the operations that the Nemesis kernel (NTSC) provides to a ULS: for example to resume execution of a thread from a particular saved context.

Consequently, when implemented over Nemesis, there are three entities involved in supporting a particular UPI:

- Firstly, the low-level facilities exported by the NTSC system call interface which are used without modification.
- Secondly, a new ULS which acts as an intermediary between the NTSC system

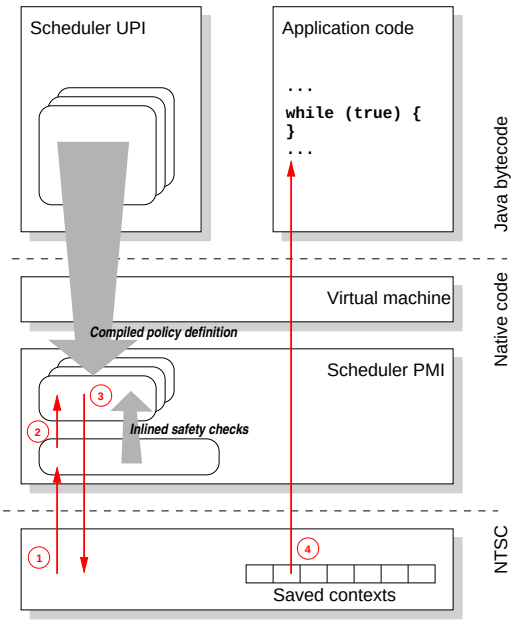


Figure 7.1: Overview of application-supplied thread schedulers. Once the process scheduler has selected which application to resume it passes control by entering the *activation handler* in its ULS. This executes the scheduling policy supplied by the application, polices the result, and resumes the selected context. The policy is compiled to native code in order to allow it to execute in a simpler environment, as well as to avoid the overhead of using the interpreter.

---

calls and an application-supplied UPI. Section 7.4 describes the way in which the ULS ensures that the safety of the VM is not compromised. This exports the thread-scheduling PMI to the application-supplied scheduler.

- Finally, the application-supplied UPI that implements the desired scheduling policy.

Figure 7.1 illustrates the overall integration of these components into the VM. The grey arrows indicate how the conceptual separation of the system into the UPI and ULS is not reflected at run-time because the UPI is compiled to native code into which the checks performed by the ULS are inlined. The red arrows illustrate the four steps which are typically involved in making a scheduling decision:

1. The NTSC process scheduler uses its own system-wide policy to select which process to resume. It passes control to that process' ULS by branching to the associated activation handler.

2. The ULS ensures that it is safe to make a scheduling decision using an application-supplied UPI and, if so, branches to the supplied policy implementation. Section 7.4 describes the situations in which the ULS must make a scheduling decision directly.

3. The UPI selects which thread to resume and passes this selection to the ULS which checks that the decision is valid and invokes an NTSC system call to pass control to the selected thread.

4. The NTSC resumes the selected thread from its saved context.

Subsequent sections illustrate this design and its implementation in more detail. Section 7.2 describes how an application-specific policy is expressed as a UPI. Section 7.3 describes the corresponding interfaces exposed by the thread-scheduling PMI. Section 7.4 shows how the VM ensures that application-supplied scheduling policies operate safely, and also clarifies what it means for a scheduling policy to be safe. Section 7.5 describes how the UPI is actually executed so that it operates efficiently when invoked for every scheduling decision. Section 7.6 illustrates how this framework may be used to implement various scheduling policies. Finally, Section 7.7 discusses how the current uni-processor system may be extended for multi-processor environments.

7.2 Thread scheduling UPI

A UPI is implemented by subclassing and specializing `ThreadScheduler`. The UPI interacts with the system in two different ways. The first is by receiving *up-calls* from the ULS when scheduling decisions are required; for example when threads are created, change state or are destroyed. The second mode of interaction occurs when

the ULS receives *down-calls* from parts of the standard libraries; for example when a thread's name is changed or its priority updated.

Up-calls from the ULS are conceptually delivered by normal method invocations on an instance of `ThreadScheduler`. These methods must be overridden in order to implement an application-supplied particular policy.

- `reschedule` is invoked from the ULS whenever a policy decision is required. It is expected to select a thread, ensure that the thread is associated with a context slot and then resume execution of that saved context by making an invocation on the scheduling PMI.
- `blockThread` and `unblockThread` are invoked from the ULS whenever a thread changes state.
- `createThread` and `destroyThread` are invoked from the ULS to notify the UPI when threads are created or destroyed.

In each of these cases the superclass `ThreadScheduler` defines a number of variants of each method, each defining successively more parameters. The intent is that programmers will override the simplest-possible definition and that this will reduce the cost of making an up-call by limiting the number of parameters that must be passed and computed.

Two versions of `reschedule` are provided. The first takes two parameters, `elapsed` and `now`, indicating the CPU time expended by the most recently scheduled thread and the current system time. The second is a simplified form which takes no parameters: it can consequently be invoked more quickly.

The `blockThread` and `unblockThread` methods have up to four possible parameters. The first of these, `thread`, identifies the `Thread` object which the invocation concerns. The second, `context`, identifies the context slot number (if

any) occupied by the thread. The third, `reason`, takes an integer value indicating why the invocation has been made. The fourth, `object`, identifies the Java-level object that caused the change of state: for example, if a thread blocks while attempting to acquire a lock, it indicates the object concerned.

It is important to realize that all of these operations are only invoked as up-calls from the ULS. This means that the ULS retains knowledge of the intended state of the threads and is consequently able to check that the UPI only resumes from valid saved contexts. In particular, when the implementation of the VM wishes to block a thread, for example because that thread must wait for a mutual exclusion lock, the VM implementation initially invokes a separate `block` operation on the ULS which in turn invokes the `blockThread` operation on the UPI. Similarly, the VM creates a thread by invoking a ULS `fork` operation which is responsible for allocating system resources to the thread, such as space for a native stack, before notifying the UPI with a `createThread` up-call.

In addition to these up-call notifications, there are other aspects of the VM operation that the implementor of a UPI may wish to consider. As before, these notifications are delivered by method invocations on the UPI when the VM implementation changes some part of its thread-related state.

However, it is not anticipated that all application-level schedulers will require this information. Therefore, for each kind of event, there is a separate '`Listener`' interface and a static registration method on the superclass. Requiring an explicit registration step and factoring operations between multiple interfaces avoids the overhead of making unnecessary invocations on the UPI for each different kind of notification. This follows the AWT event delivery model within the existing Java APIs.

Two interfaces are currently defined:

- `ThreadNameChangeListener` which contains just a single method `ThreadNameChanged(Thread t, String old_name, String new_name)` used to notify the UPI that the name of a particular thread has been changed.
- `ThreadPriorityChangeListener` which contains a single method `ThreadPriorityChanged(Thread t, int old_priority, int new_priority)` used to inform the UPI when a thread's priority is updated.

7.3 Thread scheduling PMI

The interface exposed to the application-level scheduler is provided by the class `NativeScheduler`. This provides three sets of operations, implemented as static methods.

The first set of operations is used to manage context slots. It comprises two methods, `cacheThreadContextInSlot` and `decacheThreadContextFromSlot`, which respectively create and destroy associations between Java `Thread` objects and the context slots available to the scheduler. These operations are only needed when there are more threads than available context slots.

The second set of operations is invoked by the application-level scheduler to pass its decisions to the runtime system. There are two pairs of operations: `resumeFromSlot`, `resumeFromSlotUntil`, `blockProcess` and `blockProcessUntil`. The first pair resume the execution of the thread whose state is cached in an indicated slot. The second pair are used to indicate that no thread has been selected to run and so the entire application should block. None

of these operations returns and so it is expected that any reasonable UPI will invoke one such operation each time a scheduling decision is required.

These first and second sets may only be invoked from within the implementation of the application-level scheduler and even then only when the application-level scheduler has itself been invoked by the ULS. Invocation at other times causes the operation to fail by throwing an instance of `NotInSchedulerError`.

The final set of operations provides miscellaneous facilities that are available to the UPI at any time. They include queries to obtain information about the system on which the UPI is operating – for example the current time (using a high-resolution clock) and the number of context slots that are available.

The operations on the UPI are always invoked from the ULS with activations disabled. This serializes their execution because, as explained in Section 2.2.2, disabling activations causes the NTSC process scheduler to maintain the process state in a designated context rather than invoking its ULS. In particular, it allows the notifications made by `blockThread`, `unblockThread`, `createThread` and `destroyThread` to be delivered without concurrent access to the UPI data structures by `reschedule` operations.

The implementor of the UPI must ensure any state shared with the application is correctly managed. An example of such state might be information concerning thread priorities or resource allocations. This can be achieved, to some extent, through judicious use of `volatile` variables and the VM-provided guarantee that updates are performed atomically to values of all types other than `long` and `double`.

Where this is not feasible the programmer may define *critical regions* during which the ULS will resume a particular application thread directly without consulting the application-level scheduler. The programmer delimits critical regions by acquiring and releasing a mutual-exclusion lock on the object implementing the application-

level scheduler, using the ordinary synchronization operations supported by the JVM. As a consequence, the same mechanism can be used to arbitrate between different application threads accessing state within the UPI.

Although this approach to protecting shared data provides conceptual simplicity to the application-level programmer, its implementation is problematic – for example, how should the system behave if one thread blocks while it is within a critical region? However, similar problems will exist in any case because the implementations of the methods within the application-level scheduler itself may themselves block. Section 7.5.1 will discuss the approach taken.

As an example of how this PMI may be used, Figure 7.2 shows the implementation of `reschedule`, `blockThread` and `unblockThread` for a simple round-robin scheduling policy. In this case the scheduler iterates over an array of per-thread information and resumes the first runnable thread that is found. If none of the threads is runnable then the UPI causes the process to block. Neither the `resumeFromSlot` nor `blockProcess` invocations are expected to return.

7.4 Safe application-level scheduling

The NTSC system call interface does not place any restrictions on the context slot that the native ULS selects to resume – uninitialized slots and those corresponding to blocked threads could be resumed.

That approach is appropriate within a ULS because the scheduler and the threads operate within the same protection domain: the threads could, in any case, cause the process to fail by corrupting the scheduler's data structures, or more directly by requesting that the kernel terminate the process.

In contrast, if we wish to retain run-time safety, it is necessary to ensure that the

```
public void reschedule ()
{
  int j = (last_scheduled_thread + 1) % NUM_CONTEXTS;

  for (int i = 0; i < NUM_CONTEXTS; i ++)
  {
    if (state[j] == STATE_RUNNABLE)
    {
      last_scheduled_thread = j;
      NativeScheduler.resumeFromSlot (j);
    }
    j = (j + 1) % NUM_CONTEXTS;
  }

  NativeScheduler.blockProcess ();
}

public void blockThread (int ctx)
{
  state[ctx] = STATE_BLOCKED;
}

public void unblockThread (int ctx)
{
  state[ctx] = STATE_RUNNABLE;
}
```

Figure 7.2: The implementation of a simple application-level scheduler implementing a round-robin scheduler. Other code (not shown here) initializes the `state` array, determines the maximum number of contexts available and updates the `state` array whenever threads are created or destroyed.

thread scheduler resumes only from eligible context slots. For example if the garbage collector operates in a separate thread then concurrent collection techniques generally require that other threads be suspended for some portions of the collection process: if this does not happen then the heap may be corrupted. Similar problems could occur if a thread were scheduled when it was intended to be blocked waiting to acquire a lock.

The ULS must be able to police the decisions made by the UPI in order to ensure that it only schedules runnable threads. This function is performed by interposing a check between the UPI invocation on the `NativeScheduler` class and the subsequent system call.

These checks are implemented by maintaining summary information in the ULS which identifies which context slots contain runnable thread states. The summary state is updated when `block`, `unblock`, `suspend` and `resume` operations are invoked on the ULS. This state is checked in the implementations of the methods of `NativeScheduler` and, since these are static methods, they may readily be inlined into the implementation of the UPI. Although it is intended that the application-level scheduler should not attempt to resume non-runnable threads, if such a failure occurs then the methods of `NativeScheduler` signal this by throwing an exception to the UPI, causing the usual exception-handling mechanism to be invoked.

7.5 Efficient application-level scheduling

Implementing the scheduling policy in Java bytecode above the VM raises obvious concerns over the efficiency of the implementation. There are two aspects of the system that might lead to unacceptably poor performance:

1. If a bytecode interpreter is used to execute the scheduling policy, then it may

simply take a long time for the `reschedule` method to be executed. On Nemesis a scheduling quantum of around 10ms is typical on Intel x86 systems, so if a separate scheduling decision is to be taken each time it is important that this decisions is fast.

2. Before the `reschedule` method is actually executed, the correct environment must be created within which it may operate. For example, the normal execution environment within the JVM implementation provides two stacks (one for use by the bytecode operations within the method and the other for use by any native code it invokes), a region for holding local variable values and some 16 pointers to various structures related to the method being executed. This information is used when implementing some of the more complex bytecodes of the VM, such as acquiring and releasing locks, performing virtual-method look-up or loading classes.

These concerns are addressed in two ways:

- The `reschedule`, `blockThread` and `unblockThread` methods are always compiled to native code when an application-level scheduler is registered with the VM. A separate isotope is created within the class implementing the scheduler so that its methods may be compiled under the assumption that `this` will refer to the unique instance acting as the scheduler. The assumption is always valid when the methods are invoked from the ULS. This allows accesses to fields within the scheduler definition to be bound to particular memory locations at compile time, avoiding checks for `null` references and indirection through an object pointer.

- When these methods are invoked by the ULS, they are executed within a special *deflated* execution environment that reduces the cost of entering the methods. The deflated environment uses static pre-allocated blocks of memory to contain the stacks and local variable table – bytecode verification ensures that the size of the stack frame remains within a per-method bound. Furthermore,

the various references usually maintained from the execution environment to method-related structures are not initialized in the deflated environment. Only 6 assembly language instructions are required to interface the C portions of the ULS to the compiled implementation of the `reschedule` method. In contrast 34 instructions are required to enter an ordinary compiled method.

7.5.1 Supporting arbitrary code within the scheduler

The use of a deflated execution environment places restrictions on the kinds of operation that may be performed directly within the compiled method. It is impossible to acquire or release locks, to throw or catch exceptions, to allocate objects or to invoke other methods (apart from any that may have been inlined by the native code generator).

These restrictions imposed by the spartan environment correspond closely with the separate restrictions that are imposed by execution of an ordinary method from within the scheduler. In particular, any operation that could cause `reschedule` to block must be implemented carefully in order to avoid unintended deadlocks between the application-level scheduler and the threads that it is managing.

It is unrealistic simply to forbid the application-supplied scheduler from attempting to acquire locks. This is because much of the standard library code available from the JVM is written in the expectation that it will be deployed in a multi-threaded environment. Consequently, the existing implementations of simple data structures such as hashtables and extendible arrays are defined to use mutual exclusion locks. In some situations it is possible to perform static analysis to identify locks which are never contended. Unfortunately such analysis is not trivial [Bogda99, Blanchet99, Choi99, Ruf00]. For example, the use of `finalize` methods within a Java program may introduce concurrency into an apparently single-threaded application because the finalization operation may be invoked asynchronously from within a

concurrent garbage collector. The solution currently favoured within commercial implementations of the JVM is to simplify the implementation of uncontended lock operations [Bacon98, Agesen99].

It is insufficient, however, to rely on careful lock-free programming within the scheduler and the use of static analysis to avoid lock manipulation in library code. This is because other facilities within the JVM may acquire internal locks and the application-level scheduler may need to acquire these locks even though it does not manipulate any that are defined at the language level. For example, in the current implementation, some stages of memory allocation require that the allocating thread holds a global lock that arbitrates access to the heap. The UPI may be invoked at a time when this lock is already held by another thread and therefore risks blocking upon memory allocations.

There are two different problem scenarios. The first is that the application-level scheduler performs an operation that cannot be implemented within its limited deflated execution environment. This is handled by *inflating* the execution environment, as described below in Section 7.5.2. The second problem occurs when the application-level scheduler becomes unable to complete an operation because it has blocked. This is handled directly by the underlying ULS, as described subsequently in Section 7.5.3.

7.5.2 Recovering a full execution environment

The execution environment of the application-level scheduler is temporarily *inflated* when it performs an operation that cannot be directly implemented in its usual deflated environment. When inflated the application-level scheduler operates as an ordinary thread within the VM – that is, it has a native stack, a separate Java stack and a fully-initialized execution environment structure.

The application-level scheduler is not used for scheduling decisions while it is inflated – these are handled directly by the underlying ULS. This is consistent with the view that invocations of the `reschedule`, `blockThread` and `unblockThread` operations are serialized: a further up-call to the UPI cannot be made until the existing inflated call has completed. Any `blockThread` and `unblockThread` up-calls that would occur while the UPI is inflated are buffered within the ULS and delivered before the next `reschedule` operation after deflation.

The UPI is deflated whenever an up-call completes, either by making an invocation on `NativeScheduler`, by returning directly or by completing abnormally via an unhandled exception. A slight infelicity is that if the up-call completes by invoking `NativeScheduler` then the resulting decision may be inconsistent because of buffered `blockThread` and `unblockThread` notifications. In that case the invocation is silently discarded and the UPI re-activated after delivering the buffered notifications.

The inflation procedure is implemented by caching an inflated execution environment when the UPI registers with the VM and swapping into this from the deflated environment. A context slot is reserved in the ULS for use by the inflated environment.

7.5.3 Backup user-level scheduler

Three situations have been described in which an application-level scheduler may not be able to make thread-scheduling decisions: firstly, when the implementation of `reschedule` returns to the ULS without invoking a method on `NativeScheduler`, secondly when the implementation of the UPI blocks, and finally when the UPI has been inflated and the process subsequently pre-empted before deflation. In each case the ULS reverts to making thread-scheduling decisions. The intent is that this is a rare occurrence and so a simple policy suffices to enable

progress by the application: to either un-block the UPI or to complete its current invocation and allow it to be deflated.

This policy identifies a *primary thread* which was the direct cause of reversion to the ULS. If the UPI blocked then the primary thread is the one that holds the lock on which it is waiting. If the UPI was inflated then the primary thread is the one that is associated with the inflated environment.

The ULS proceeds by scheduling the primary thread until either it unblocks the UPI, the UPI may be deflated or the primary thread itself blocks. In the first two cases the scheduling policy may revert to that defined by the UPI. In the third case a new thread is selected as the primary. If the primary thread blocked acquiring a lock then the current holder of that lock is selected as the new primary. Otherwise, the runnable threads are scheduled in a round-robin manner. The rationale for this policy is that a primary thread is selected, where possible, as the one that is directly delaying returning to using the UPI and that, where such a thread cannot be identified, a round-robin schedule will avoid starvation.

7.6 Example policy definitions

This section describes thread scheduling policies that can be implemented using the application-level scheduling infrastructure described in the previous sections.

7.6.1 Priority-based scheduling

The original example, shown in Figure 7.2, illustrated how a simple round-robin scheduler could be implemented with a `reschedule` method that cyclically selects one of a number of threads. It is straightforward to extend this with

separate per-priority queues to implement the scheme required by the Java Language Specification (JLS) in which '*when there is competition for processing resources then threads with high priority are generally executed in preference to threads with low priority*' [Gosling97a].

These kinds of simple policy provide few opportunities for application-specific customization. Aside from the obvious consideration of which threads are assigned to which priorities, the only other variable is the choice of how many times the UPI selects a particular thread before it is pre-empted. The provision of the UPI under the control of the application programmer permits variation in rescheduling frequency if a program exhibits different phases of behaviour.

However, the loose specification made in the JLS and provided by a priority-based scheduler is not suitable for all application programs. There are two particularly noteworthy cases: firstly, when the application programmer wishes to use *strict* priority scheduling [1] and, secondly, where the application programmer wishes to use some different description of threads' CPU requirements.

The first of these cases is typified by the use of priority settings in order to achieve mutual exclusion without explicitly manipulating locks: if a thread has the highest priority then the scheduling policy guarantees that it will be the thread that is executed. It has particular practical appeal in highly multi-threaded systems in which one thread wishes to update shared data structures that are generally accessed without locks. However, such an approach lacks merit in many situations. Firstly it assumes a uni-processor environment because the execution of a single high-priority thread by one processor will not exclude other threads from being executed elsewhere. Secondly, using strict priority scheduling to control concurrent access to data structures relies on the high priority thread operating without blocking –

[1] That is, where the scheduler guarantees to schedule the runnable thread with the highest priority, unlike the JLS specification in which the only requirement is that high priority threads *tend* to be favoured.

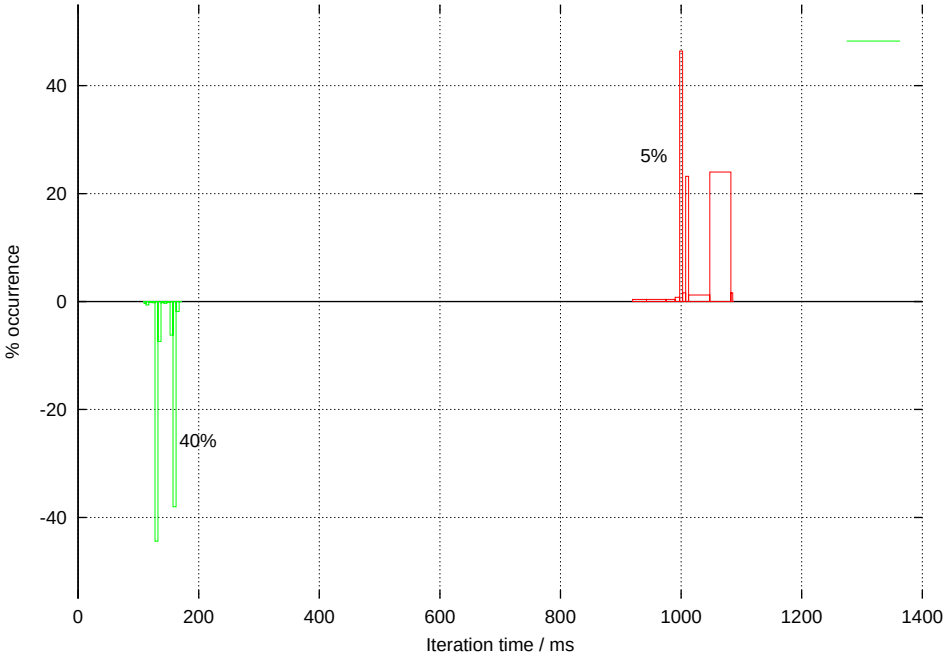
either directly on programmer-defined locks, or indirectly on internal locks within the VM. Finally, if the VM may execute pieces of code within the context of other threads, then those implementations must also be considered – for example this may occur within the JVM if `finalize` methods or class initialization methods are executed in the context of the thread that triggers a garbage collection or class loading. However, despite these concerns with the use of priority scheduling to control access to shared data structures, it would be straightforward to implement as a UPI: all that would be required, over the non-strict version, would be for a `ThreadPriorityChangeListener` to yield if the change raises some other thread priority above that of the thread currently scheduled.

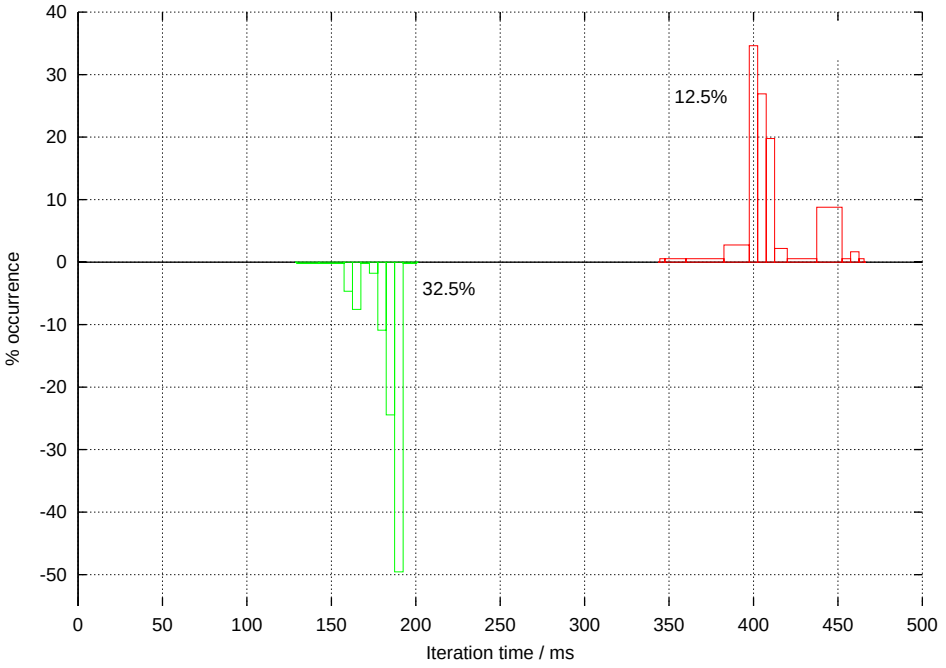
The second situation in which a programmer may wish to use a different policy is where simple priorities are unable to express the desired allocation of resources to the application threads. For example, suppose that a single JVM contains two threads, each running on behalf of a separate Java applet – it is impossible to control the allocation of CPU time at a fine granularity because all that can be adjusted are the relative priorities.

7.6.2 Period-proportion based allocation

A further policy that has been implemented is a *period-proportion* system. Such schedulers have become popular in soft real-time environments for their ability to provide CPU allocations of the form '10ms CPU time every 100ms elapsed time' [Leslie96, Jones97, Steere99].

The implementation in a UPI is again straightforward and follows that of similar process schedulers using the same policy [Leslie96, Roscoe95]. A queue of threads is maintained, ordered according to the end of their current period. These are then treated as deadlines by which each thread must receive $proportion * period$ CPU time.


40
20
0
-20
-40
% occurrence
5%
40%
0
200
400
600
800
1000
1200
1400
Iteration time / ms


40
30
20
10
0
-10
-20
-30
-40
-50
% occurrence
12.5%
32.5%
0
50
100
150
200
250
300
350
400
450
500
Iteration time / ms

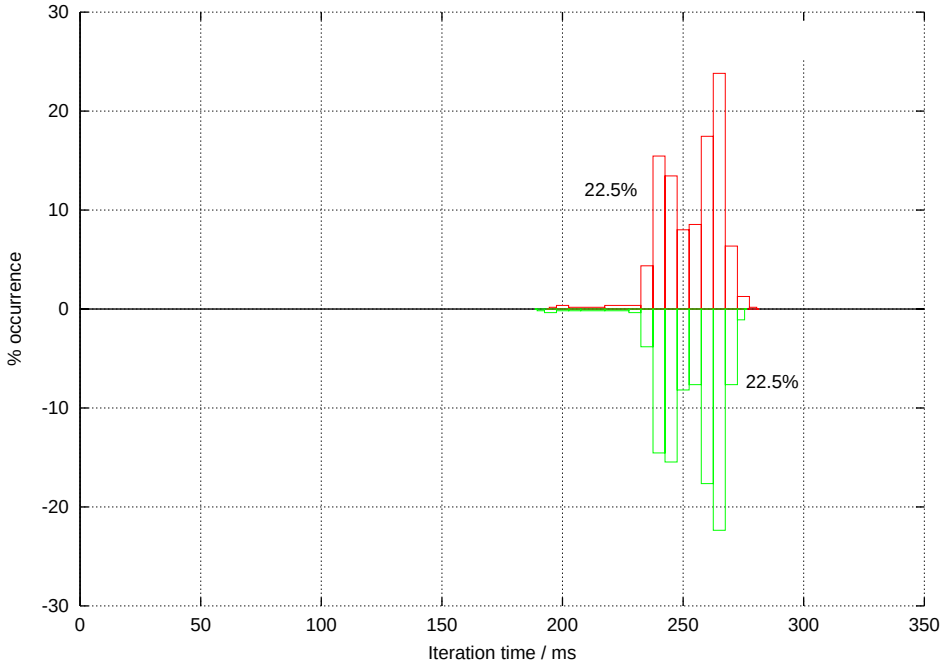


Figure 7.3: The rate of progress achieved by threads running with different CPU allocations. Each plot shows the results from a separate experiment in which two CPU-bound threads executed a loop a number of times, showing the time taken for each iteration of the loop and the number of occasions on which a particular iteration time is achieved. In each case one of the traces, shown in green, is negated to aid comparison with the other.

Figure 7.3 illustrates the behaviour of this scheduler using a simple CPU-bound benchmark. Each iteration of the benchmark is timed and the distribution of these times is plotted for various CPU allocations (all made over a $100ms$ period). Two threads were used in each experiment and a total CPU allocation of 45% was distributed between them, varying from a skew of 5%-40% to an even allocation of 22.5% to each thread. Each plot shows the distribution of iteration times achieved by the two threads, with the result for the green trace negated to aid comparison with the red trace.

7.6.3 Allocation inheritance

When used in the Nemesis process scheduler, the effectiveness of using a period-proportion specification benefits from the structure of the Nemesis operating system: processes are responsible for their own I/O so the performance of one should depend only on its allocation. Locks are only expected to be used between cooperative threads *within* processes.

However, one of the motivations that has frequently been advanced for per-thread resource allocation within a VM is that it allows separate applications to be executed as 'servlets' within a single process [Back98, Hawblitzel98, Czajkowski98, Spoonhower98]. This may allow easy communication between co-operating tasks which, at least for Java applications, would not be possible if they executed in separate instances of the VM. A more complex system configuration may provide other facilities, implemented as trusted native code, that may be invoked by the servlets [Menage99]. That previous work has confirmed that the use of a period-slice based resource specification is problematic: a thread with a low resource allocation may delay well-provisioned threads by executing slowly when holding critical locks. An additional concern, in some environments, is that a malicious programmer could deliberately revoke *all* resources from a thread in such a situation. The problem is similar to that of destroying arbitrary un-cooperative threads: initial versions of the Java `Thread` class provided a `destroy` method for this purpose, but this was subsequently deprecated because a thread could be destroyed while holding system-critical locks, or even be destroyed while internal JVM data structures were in an inconsistent state.

The approach taken in most previous work has been to avoid acquiring system-wide locks while executing code whose resource allocation may be controlled by untrusted code. For example, the J-Kernel (a Java-based system supporting multiple protection domains and resource management) only permits cross-domain calls to designated *capability objects* [Hawblitzel98]. As with Sun's Remote Method Invocation (RMI), parameters and results are generally copied when performing the call – by avoiding

sharing between domains the scheme mitigates the problems of domain termination and of accounting shared resources. Conceptually, a separate thread is used when the cross-domain call executes in the server. This prevents a malicious client from denying resources to its own threads while they hold locks within a server and similarly prevents a malicious server from retaining references to client threads once the cross-domain call has completed. However, threads may still contend for system-provided locks, or for locks on the capability objects themselves because those are directly accessible to multiple protection domains.

Since the J-Kernel is implemented over an unmodified JVM the manner in which a thread's resource requirements are expressed to the system uses the usual priority-based interfaces. It is therefore susceptible to the problems discussed in the introduction: the controls it introduces provide additional forms of isolation between the resources accessible within a protection domain, but do not necessarily provide control over the *amount* of those resources available to each domain.

JRes is an alternative resource management infrastructure for use with the JVM [Czajkowski98]. It aims to provide mechanisms for measuring and limiting resource usage, including CPU time, heap memory and network resources. An example application for JRes is a Java-based web server executing client-supplied code – a more extreme motivating example may be a *XenoServer* executing code on behalf of untrusted clients and billing those customers for the associated resource usage [Reed99].

In the case of thread scheduling, JRes allows a `ResourceManager` to receive notifications when threads are created. This manager can limit the thread's consumption of various resources and receive call-backs when those limits are reached. A CPU resource allocation is expressed in terms of milliseconds per second of wall-time. Consumption is measured from native code by periodically sampling structures maintained by the thread scheduler.

Therefore, although JRes provides mechanisms for controlling resource allocations to threads, it does not expose control over the scheduling policy itself. A separate problem with JRes as proposed by Czajkowski *et al* is that heap memory usage is monitored by using load-time code rewriting to introduce usage-tracking code into constructors and finalizers [Czajkowski98]. An unfortunate consequence of this is that a malicious thread can presumably access these counters directly, or resurrect reachable finalized objects, to acquire unaccounted storage space.

Bernadat *et al* propose another resource management framework based on JVM extensions and APIs for partitioning memory and CPU resources between mutually untrusting applications. For thread scheduling it uses a POSIX round-robin scheduling scheme extended with an interface to measure per-thread CPU time consumption. The scheduling parameters appear to be based on a feedback scheme in which the time received by some threads is shown to take around one minute to stablise on the value intended [Bernadat98].

In a VM-based platform for resource-managed active networks, Menage identifies two schemes for scheduling threads contending for shared resources [Menage00]. The first allows a server to 'underwrite' client threads with its own resource allocations: if a client thread cannot continue while in the server then the resources allocated to the underwriter are used to allow it to continue until it (hopefully) leaves the server. The second scheme allows the server to measure client resource allocations before they make invocations on it: it essentially requires that the client pay 'up front' to prevent it running out of resources part way through executing the server code.

The availability of application-level scheduling, however, permits more flexibility in defining how a particular system should manage the interaction between locking and resource allocation. The particular scheme presented here is a straightforward technique inspired by the standard *priority inheritance* algorithm for avoiding priority inversion.

The basic idea is that if a thread attempts to acquire a contended lock then, instead of blocking, it donates its allocated CPU time to the thread currently holding that lock. The intent is that, as with priority inheritence, the thread favoured by the scheduler elects that the system as a whole should perform work that is useful to that thread, even if the work is not the direct execution of its own code. The initial UPI implementation is straightforward – comprising an extension of the scheduler described in Section 7.6.2 in which threads blocked on locks are maintained in the run-queue and, if selected for execution, the current holder of the contended lock is selected instead of the blocked thread. There are, however, subtle problems which must be addressed:

Accounting donated time

Care is required over how the donated time is accounted to the threads involved. Ford and Susarla describe the implementation of a similar donation mechanism over their user-space CPU inheritance scheduling infrastructure [Ford96b]. They propose that the entire CPU usage is accounted to the thread making the donation since the donation apparently occurs without the acquiescence of the recipient and, more pragmatically, without the knowledge of the accounting mechanism. Intuitively, such an approach appears necessary in the current environment in order to retain the guarantee of schedulability from the underlying Earliest Deadline First (EDF) algorithm because, unless the donor is charged, it will retain its CPU allocation in the current period and ultimately there will be insufficient real time before the end of the period for the allocation to be honoured.

However, without penalising the recipient of a donation it is possible that some threads will be net donors and some will be net recipients – if contended locks are available directly to application threads then some may elect to strategically hold unnecessary locks in order to benefit from donations. In order to promote longer

term fairness, the UPI maintains separate per-thread balances of the net debt held against other threads as a whole. If the UPI selects a debtor thread to be scheduled then the balance of the debt (or the current allocation, if less) is transferred to a thread in credit, and that thread is executed in preference at the debtor's expense. Threads in credit are held on a simple circular list and receive credit in a round-robin manner.

Recursive donation

It is possible that the thread currently holding a contended lock will itself have blocked because it is in turn waiting to acquire a lock. In that case the CPU donation is simply made to the thread that is ultimately delaying the chain of threads. A complication arises from the desire to avoid transforming deadlock into livelock by attempting to find the end of a cycle while analyzing dependencies between threads. In anticipation that `blockThread` and `unblockThread` operations are frequent, whereas allocation donation is rare, it is desirable to avoid introducing deadlock detection on every lock manipulation. Instead, when selecting the recipient of a donation, the traversal algorithm maintains a count of the number of threads that it traverses: a deadlock must exist if the count exceeds the total number of threads in the system, since any sequence of such a length necessarily contains a repeated suffix. The UPI then repeats the traversal, marking the repetitions as being in a `THREAD_DEADLOCKED` state rather than `THREAD_BLOCKED`. Threads in that state are never considered for scheduling or as potential donors of their allocated time.

Blocking for other reasons

The thread currently holding a lock may have blocked for some reason other than lock contention; for example it may be waiting for an I/O operation to complete. There is no clear general strategy for avoiding such problems: if the thread is waiting for keyboard input then there is evidently no way that the UPI can encourage the

operation to complete. Indeed, if the lock protects a shared data structure that will contain the input data then the first thread is effectively also awaiting the user's data entry and therefore blocking both threads is consistent with the view that the UPI is doing whatever *it* can to ensure that the system works towards the goals of the scheduled thread.

Conversely, the second thread may be operating maliciously and have acquired a set of important locks before blocking on a futile I/O operation. It is possible that an extended UPI could address that situation by providing a mechanism for registering objects as providing CPU-*bound locks* and to forbid blocking I/O invocations while holding these – an operation that would block will instead abort with an exception.

However, this policy cannot be implemented over the existing ULS infrastructure because the `blockThread` and `unblockThread` operations act as up-calls after the underlying ULS summary information has been updated: the UPI itself cannot introduce operations, such as throwing an exception, into the threads that it controls. Furthermore, adding such support would not solve this problem in the more general case: the second thread could simply enter an endless CPU-bound loop instead of performing a blocking I/O operation.

The general recommendation is therefore that the server infrastructure that supports untrusted servlets must take care over the locks to which it allows access. The standard access control mechanisms of the JVM prevent internal system locks, such as those controlling access to the garbage-collected heap, from being acquired and released explicitly by the servlets. Therefore the intent is that allocation inheritance limits the effects of contention on those locks rather than providing a 'silver-bullet' solution for sharing data structures between un-cooperative threads.

Discussion

In each of these cases it is important to remember that the policy is implemented within the UPI. Therefore the choices made in the current implementation merely reflect behaviour that is anticipated to be reasonable for a range of situations – the particular UPI that has been developed could equally be replaced by one that makes different trade-offs. A particular example is that the choice of how and whether to charge the recipient of donated CPU time and whether any allocations made should provide an upper bound on the resources expended on a thread, or whether they are viewed as guarantees on the minimum that a thread should receive. In the former case, as implemented here, one expects a thread to be penalised when it receives a donation. In the alternative one may expect a thread to continue to receive its entire allocated time.

7.7 Multi-processor scheduling

The application-level scheduling infrastructure, as currently implemented, operates only on uniprocessor machines because the Nemesis operating system is restricted to such an environment. Moving the current implementation directly to a multi-processor system will introduce additional complexity because there are extra situations that must be considered: in particular the implementation of concurrent `reschedule` operations across more than one CPU.

The current design for multi-processor application-level scheduling is to offer two options to the UPI implementor. The first of these is similar to the uni-processor design in that a single UPI is supplied and that the ULS enforces mutual-exclusion between the invocation of the up-call methods upon different processors. If a second processor requires an up-call while a separate call is already in progress on the first processor then that second processor will wait until the first completes. If, as expected, up-calls form a small proportion of total process execution time then it might be appropriate for the waiting processor to simply spin on a shared lock.

The second option is that multiple UPI instances – that is, multiple instances of the `ThreadScheduler` type – may be used across the processors. Concurrent up-calls would be permitted to different UPI instances, consistent with the view that each up-call conceptually acquires a lock on the target UPI instance. These instances would have to co-operate through the usual facilities provided by the VM in order to share scheduling information. Each UPI would operate on a separate set of context slots and `blockThread` and `unblockThread` notifications would be dispatched to the UPI that currently has the indicated thread. The `NativeScheduler` class would need to be extended to support the migration of context slot values from one UPI to another. However it is anticipated that this migration would occur over relatively long timescales in order to favour process-processor affinity.

In each case the summary information maintained by the ULS would need to be extended to distinguish between *Runnable* processes (those that are eligible for scheduling) from *Running* ones (those that are currently being executed on some other processor).

7.8 Discussion

This chapter has shown how application-controlled thread scheduling can be implemented within the XVM architecture. The system here is able to achieve some of the requirements identified in Section 4 in that it separates the implementation of the scheduling policy from the application itself, thereby promoting policy reuse and allowing policies to be changed independently from applications.

However, the design of Chapter 4 fits less well here than it did in Chapters 5 and 6. This follows from the essentially centralized nature of thread scheduling: one scheduler must be selected for the entire application, whereas decisions for run-time compilation or storage management can be made on a much finer granularity.

Chapter 8

Conclusion

This dissertation has presented an architecture for providing application-accessible extensibility in a virtual machine and shown how this design may be realized in implementation for three different problem domains. This final chapter summarises the work undertaken and presents suggestions for future work that builds on it.

8.1 Summary

Chapter 1 argued that existing VMs, by presenting high levels of abstraction from the underlying hardware, favour program portability over application-specific optimizations. It presented the thesis that applications can benefit through lower-level control of the resources and services that they use.

Chapter 2 described background work relating the development of VMs and the more recent development of extensible operating systems. The former set out to clarify the kind of environment within which the work described here was based. The relevance of the latter came from the analogy with moving processing from an operating system kernel into untrusted user-space code.

Chapter 3 surveyed related work and expanded on the rationale for the new work in this dissertation. It described existing projects which exploit the flexibility that conventional VMs offer and identified areas in which the limited extent of that flexibility presents a barrier to the use of a VM as a ubiquitous execution environment.

Chapter 4 introduced the proposed architecture for an extensible virtual machine. A common framework was developed within which run-time compilation, memory allocation and thread scheduling policies were taken as particular examples. In outline, policies are defined in a general purpose programming language and are implemented by making invocations on *protected mechanism implementations* that are provided by the VM. A *policy registry* records the association between policies and sections of the application.

Chapter 5 described the implementation of this infrastructure over the JVM as a mechanism for defining application-specific policies to control run-time compilation. The primary purpose of such a policy is to define which parts of the application are compiled, when that compilation occurs and what kinds of optimization are attempted.

Chapter 6 described the corresponding implementation of policies to support application-specific memory allocation. In this case the policies may control where objects are placed within the heap -- for example to cluster objects that are expected to be used together.

Chapter 7 described the final area in which this dissertation investigated the use of application-supplied policies. It showed the development of an application-level thread scheduler with which an untrusted program may define the way in which its threads are multiplexed over the CPUs available to the VM.

8.2 Future research

The work presented in this dissertation highlights three problem domains over which an application programmer may wish to exercise control. In each case the architecture presented in Chapter 4 was specialized and implemented according to the particular requirements of that domain -- for example according to the rate at which policy decisions may be required, the inputs that a policy may be expected to use and the safety considerations of the interfaces that have been exposed.

Work such as this presents numerous and obvious scope for extension to additional policy domains.

8.2.1 Garbage collection

Section 3.2 identified the selection of an appropriate garbage collection algorithm as one of the ways in which system performance depended on application behaviour. As with the discussion at the end of Chapter 6 regarding untrusted storage allocation mechanisms, it is difficult to envisage how to provide application control on a remotely fine-grained basis.

The examples shown there illustrated how type safety depended on preventing living objects from premature reuse. In the case of storage allocation it was possible to use linear object references to ensure that the untrusted algorithm did not allocate the same memory on more than one occasion. Similarly, in the case of explicit storage *deallocation*, the system must ensure that the object proposed for reuse is actually free. Effectively, in order to deploy an untrusted implementation of a conventional garbage collector, the system would require a separate trusted collector to operate in order to confirm any deallocations made. There is, of course, also scope for investigating other memory management schemes, such as the capability- and region-based system proposed by Crary, Walker and Morrisett [Crary99].

Current implementations of the JVM do provide a range of garbage collection algorithms and mechanisms for selecting between them at the time the VM is started [Printezis01]. One possible extension to the storage allocation UPI from Chapter 6 would be to investigate the use of different garbage collection algorithms for different areas of the heap -- in much the same way as multi-generational collectors may use one algorithm for managing newly-allocated objects and another for those that have been tenured.

8.2.2 Pre-fetching data access

Traditional APIs provided for I/O in programming languages provide operations either to access buffers of data from within files or to treat the file as a sequential data stream in which successive pieces of data may be accessed. The devices accessed through these interfaces provide substantially different kinds of performance, depending both on the underlying physical storage medium and on the success of caching.

It is sometimes possible to improve I/O performance by issuing appropriate *pre-fetching* requests so that a data transfer is initiated before the resulting data is required. However, this is hard to automate because good performance depends both on future application behaviour and on device characteristics. Furthermore, for some devices it is unsafe to use pre-fetching in case incorrect predictions have user-visible effects such as failure indications for speculative access beyond the end of a file.

A UPI to control data pre-fetching could be based on receiving notification when the application makes a file access. A possible implementation scheme would be to use a bounded-size shared memory buffer for communication between the VM and a thread implementing the UPI -- this may be preferable to an up-call based interface because it reduces policy-decision overhead on each application-made file access: pre-fetching is only useful when CPU utilization is low.

```
static Object o = new Object ();

public void run () {
  while (true) {
    synchronized (o) {
      ...
    }
  }
}
```

Figure 8.1: If multiple threads are instantiated from this code fragment then current APIs do not provide any control over which threads will be able to acquire the mutual-exclusion lock on the single object referenced by `o`.

8.2.3 Lock acquisition order

The mutual-exclusion locks provided by the JVM do not provide any form of guarantee over which threads will be favoured for access. For example, Figure 8.1 shows a code fragment which repeatedly acquires and releases a lock on an object. If multiple threads execute this fragment then there is no way to ensure fair access to the lock. Similarly there is no way to deliberately skew access in favour of some thread. Although a programmer could manually implement a particular locking scheme using condition variables and `wait`/`notify` operations, such an approach is undesirable for a number of reasons. It is likely to incur a substantial penalty in terms of execution time because the programmer must replicate data structures (such as queues of waiting threads) within the application. It also entails modifying the shared code and consequently requires that its source code be available.

The description of policies for lock acquisition order is therefore another candidate for deploying the architecture presented in this dissertation. A reasonable approach may be to follow the thread scheduling UPI of Chapter 7 in using run-time compilation to inline the desired policy within particular implementations of `lock`/`unlock` operations.

Bibliography

[Abadi98] Martín Abadi. *Protection in Programming-Language Translations*. In 25th International Colloquium on Automata, Languages and Programming (ICALP '98), volume 1443 of *Lecture Notes in Computer Science*, pages 868--883. Springer-Verlag, July 1998. Also appeared as SRC Research Report 154 (April 1998). (p 80)

[Abadi99] Martín Abadi and Andrew D. Gordon. *A Calculus for Cryptographic Protocols: the Spi Calculus*. Information and Computation, 148(1):1--70, 10 January 1999. Also appeared as SRC Research Report 149 (January 1998). (p 80)

[Abelson98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. *Revised Report on the Algorithmic Language Scheme*. Higher-Order and Symbolic Computation, 11(1):7--105, August 1998. (pp 52, 55)

[Agesen00] Ole Agesen and David Detlefs. *Mixed-mode bytecode execution*. Technical report SML-TR-2000-87, Sun Microsystems, June 2000. (p 35)

[Agesen01] Ole Agesen and Alex Garthwaite. *Efficient Object Sampling Via Weak References*. In International Symposium on Memory Management (ISMM '00), volume 36(1) of *ACM SIGPLAN Notices*, pages 121--126, January 2001. (p 132)

[Agesen95] Ole Agesen. *Concrete Type Inference: delivering object-oriented applications*. PhD thesis, Stanford University, December 1995. (p 26)

[Agesen97a] Ole Agesen. *Design and Implementation of Pep, a Java Just-in-Time Translator*. Theory and Practice of Object Systems, 3(2):127--155, 1997. (p 56)

[Agesen97b] Ole Agesen, Stephen N. Freund, and John C. Mitchell. *Adding Type Parameterization to the Java Language*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '97), volume 32(10) of *ACM SIGPLAN Notices*, pages 49--65, October 1997. (p 79)

[Agesen98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. *Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines*. In Programming Language Design and Implementation (PLDI '98), volume 33(5) of *ACM SIGPLAN Notices*, pages 269--279, May 1998. (p 57)

[Agesen99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. *An Efficient Meta-Lock for Implementing Ubiquitous Synchronization*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99), volume 34(10) of *ACM SIGPLAN Notices*, pages 207--222, November 1999. (pp 30, 163)

[Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986. (p 115)

[Anderson92] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*. In ACM Transactions on Computer Systems, volume 10, pages 53--79, February 1992. (p 44)

[Back98] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. *Java operating systems: design and implementation*. Technical Report UUCS-98-015, University of Utah, Department of Computer Science, August 1998. (p 170)

[Bacon98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. *Thin Locks: Featherweight Synchronization for Java*. In Programming Language Design and Implementation (PLDI '98), volume 33(5) of *ACM SIGPLAN Notices*, pages 258--268, May 1998. (pp 30, 163)

[Baker95] Henry G. Baker. *'Use-Once' Variables and Linear Objects -- Storage Management, Reflection and Multi-Threading*. ACM SIGPLAN Notices, 30(1):45--52, 1995. (p 145)

[Barham97] Paul R. Barham. *A Fresh Approach to Filesystem Quality of Service*. In Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97), St. Louis, Missouri, USA, pages 119--128, May 1997. (p 43)

[Barrett93] David A. Barrett and Benjamin G. Zorn. *Using lifetime predictors to improve memory allocation performance*. In Programming Language Design and Implementation (PLDI '93), volume 28(6) of *ACM SIGPLAN Notices*, pages 187--196, June 1993. (pp 70, 131)

[Barron81] D W Barron. *A perspective on Pascal*. In Pascal -- the language and its implementation, pages 1--3. John Wiley Sons, 1981. (p 21)

[Bell73] James R. Bell. *Threaded Code*. Communications of the ACM, 16(6):370--372, June 1973. (p 116)

[Benton99] Nick Benton, Andrew Kennedy, and George Russell. *Compiling Standard ML to Java bytecodes*. In International Conference on Functional Programming (ICFP '98), volume 34(1) of *ACM SIGPLAN Notices*, pages 129--140, January 1999. (pp 52, 54, 54)

[Bernadat98] Philippe Bernadat, Dan Lambright, and Franco Travostino. *Towards a Resource-safe Java for service guarantees in uncooperative environments*. In IEEE Workshop on Programming Languages for Real-Time Industrial Applications, pages 101--111, December 1998. (p 172)

[Bershad95] Brian N. Bershad, Stegan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. In Symposium on Operating Systems Principles (SOSP '95), volume 29(5) of *Operating Systems Review*, pages 267--284, December 1995. (pp 39, 150)

[Bertelsen98] Peter Bertelsen. *Dynamic semantics of Java byte-code*. In Workshop on Principles of Abstract Machines, September 1998. (p 29)

[Black94] Richard Black. *Explicit Network Scheduling*. Technical Report TR361, University of Cambridge Computer Laboratory, December 1994. (p 43)

[Blanchet99] Bruno Blanchet. *Escape Analysis for Object Oriented Languages. Application to Java*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99), volume 34(10) of *ACM SIGPLAN Notices*, pages 20--34, October 1999. (p 162)

[Bobrow88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common LISP Object System Specification X3J13 Document 88-002R*. ACM SIGPLAN Notices, 23(9), September 1988. Special issue. (pp 25, 55)

[Bogda99] Jeff Bogda and Urs Hölzle. *Removing Unnecessary Synchronization in Java*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99), volume 34(10), pages 35--46, October 1999. (p 162)

[Bothner98] Per Bothner. *Kawa -- compiling dynamic languages to the Java VM*. In USENIX Annual Technical Conference, 1998. (p 55)

[Bracha98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. *Making the future safe for the past: adding genericity to the Java programming language*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '98), ACM SIGPLAN Notices, pages 183--200, October 1998. (p 79)

[Chambers89] Craig Chambers, David Ungar, and Elgin Lee. *An efficient implementation of SELF, a dynamically-typed object-oriented*

language based on prototypes. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '89), volume 24(10) of *ACM SIGPLAN Notices*, pages 49--70, October 1989. (p 26)

[Chambers91] Craig Chambers and David Ungar. *Making Pure Object-Oriented Languages Practical*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '91), volume 26(11) of *ACM SIGPLAN Notices*, pages 1--15, October 1991. (p 26)

[Chambers92] Craig Chambers. *Object-Oriented Multi-Methods in Cecil*. In European Conference on Object-Oriented Programming (ECOOP '92), volume 615 of *Lecture Notes in Computer Science*, pages 33--56. Springer-Verlag, 1992. (p 55)

[Cheng98] Perry Cheng, Robert Harper, and Peter Lee. *Generational Stack Collection and Profile-Driven Pretenuring*. In Programming Language Design and Implementation (PLDI '98), volume 33(5) of *ACM SIGPLAN Notices*, pages 162--173, May 1998. (p 71)

[Chilimbi98] Trishul M. Chilimbi and James R. Larus. *Using generational garbage collection to implement cache-conscious data placement*. In International Symposium on Memory Management (ISMM '98), volume 34(3) of *ACM SIGPLAN Notices*, pages 17--19, October 1998. (p 73)

[Chilimbi99] Trishul M. Chilimbi, James R. Larus, and Mark D. Hill. *Cache-conscious structure layout*. In Programming Language Design and Implementation (PLDI '99), volume 34(5) of *ACM SIGPLAN Notices*, pages 1--12, May 1999. (p 72)

[Chitnis95] Sachin V. Chitnis, Manoranjan Satpathy, and Sundeep Oberoi. *Rationalized Three Instruction Machine*. ACM SIGPLAN Notices, 30(3):94--102, March 1995. (p 20)

[Choi99] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. *Escape Analysis for Java*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '99), volume 34(10) of *ACM SIGPLAN Notices*, pages 1--19, October 1999. (p 162)

[Coglio98] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. *Toward a provably-correct implementation of the JVM bytecode verifier*. In Proceedings of the OOPSLA 98 workshop on the formal underpinning of Java, October 1998. Also appeared as Kestrel Institute Technical Report KES.U.98.5. (p 29)

[Collberg98] Christian Collberg, Clark Thomborson, and Douglas Low. *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs*. In Proceedings of the twenty-fifth ACM Symposium on Principles of Programming Languages (POPL '98), pages 184--196, San Diego, California, January 1998. (p 29)

[Crary99] Karl Crary, David Walker, and Greg Morrisett. *Typed Memory Management in a Calculus of Capabilities*. In Proceedings of the twenty-sixth ACM Symposium on Principles of Programming Languages (POPL '99), pages 262--275, January 1999. Also appeared as Cornell Technical Report TR2000-1780. (p 181)

[Creasy81] R. J. Creasy. *The origin of the VM/370 time-sharing system*. IBM Journal of Research and Development, 25(5):483--490, September 1981. (pp 16, 47)

[Cytron89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. *An efficient method of computing static single assignment form*. In Proceedings of the sixteenth ACM Symposium on Principles of Programming Languages (POPL '89), pages 25--35, January 1989. (p 33)

[Czajkowski98] Grzegorz Czajkowski and Thorsten von Eicken. *JRes: a resource accounting interface for Java*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '98), volume 33(10) of *ACM SIGPLAN Notices*, pages 21--35, October 1998. (pp 170, 171, 172)

[Davidson87] J. W. Davidson and J. V. Gresh. *Cint: a RISC interpreter for the C programming language*. In Symposium on Interpreters and Interpretive Techniques, volume 22(7) of *ACM SIGPLAN Notices*, pages 189--198, July 1987. (p 34)

[Dean96] Jeffrey Dean. *Whole-Program Optimization of Object-Oriented Languages*. Technical Report TR-96-11-05, University of Washington, Department of Computer Science and Engineering, November 1996. (p 68)

[Dean97] Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. *Java Security: Web Browsers and Beyond*. Technical Report TR-566-97, Princeton University, Computer Science Department, February 1997. (p 29)

[Deutsch84] Peter Deutsch and Alan M. Schiffman. *Efficient Implementation of the Smalltalk-80 System*. In Proceedings of the eleventh ACM Symposium on Principles of Programming Languages (POPL '84), pages 297--302, January 1984. (pp 23, 70)

[Dobson98] Simon Dobson. *A first taste of Vanilla*. Technical Report TCD-CS-1998-20, Trinity College, University of Dublin, September 1998. (p 63)

[Duba91] Bruce F. Duba, Robert Harper, and David B. MacQueen. *Typing first-class continuations in ML*. In Proceedings of the eighth ACM Symposium on Principles of Programming Languages (POPL '91), pages 163--173, 1991. Also appears as CMU Technical Report CMU-CS-90-184. (p 151)

[Engler95] Dawson R. Engler and M. Frans Kaashoek. *Exterminate all operating system abstractions*. In Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-V), pages 78--83, 1995. (pp 41, 49)

[Engler96] Dawson R. Engler. *VCODE: a Retargetable, Extensible, Very Fast Dynamic Code Generation System*. In Programming Language Design and Implementation (PLDI '96), volume 31(5) of *ACM SIGPLAN Notices*, pages 160--170, May 1996. (p 60)

[Folliot98] B. Folliot, I. Piumarta, and F. Riccardi. *A Dynamically Configurable, Multi-Language Execution Platform*. In Eighth ACM SIGOPS European Workshop, September 1998. (p 63)

[Ford96a] Bryan Ford, Mike Hibler, Jay Lapreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. *Microkernels meet recursive virtual machines*. In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96). The USENIX Association, October 1996. Also appeared as University of Utah, Department of Computer Science Technical Report UUCS-96-004. (p 47)

[Ford96b] Bryan Ford and Sai Susarla. *CPU Inheritance Scheduling*. In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), pages 91--105. The USENIX Association, October 1996. (pp 48, 173)

[Forman94] Ira R. Forman, Scott Danforth, and Hari Madduri. *Composition of Before/After Metaclasses in SOM*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '94), volume 29(10), pages 427--439, October 1994. (p 97)

[Franz97] Michael Franz. *Run-time code generation as a central system service*. In Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VI), pages 112--117, May 1997. (p 65)

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994. (pp 62, 107)

[Girard87] Jean-Yves Girard. *Linear Logic*. Theoretical Computer Science, 50:1--102, 1987. (p 145)

[Goldberg83] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983. (pp 16, 22, 25)

[Gosling95] James Gosling. *Java intermediate bytecodes*. ACM SIGPLAN Notices, 30(3):111--118, March 1995. (pp 28, 52)

[Gosling97a] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997. (pp 22, 25, 28, 166)

[Gosling97b] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface. Vol. 1: Core packages*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997. (pp 28, 105)

[Hand98] Steven M. Hand. *Providing quality of service in memory management*. PhD thesis, University of Cambridge Computer Laboratory, November 1998. (p 44)

[Hand99] Steven M. Hand. *Self-Paging in the Nemesis Operating System*. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), pages 73--86. The USENIX Association, February 1999. (p 44)

[Hardwick96] Jonathan C. Hardwick and Jay Sipelstein. *Java as an Intermediate Language*. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, August 1996. (p 59)

[Harris01] Timothy L. Harris. *Dynamic Adaptive Pre-Tenuring*. In International Symposium on Memory Management (ISMM '00), volume 36(1) of *ACM SIGPLAN Notices*, pages 127--136, January 2001. (pp 71, 89, 131, 132)

[Harris98] Timothy L. Harris. *Controlling run-time compilation*. In IEEE Workshop on Programming Languages for Real-Time Industrial Applications, pages 75--84, December 1998. (p 15)

[Harris99] Timothy L. Harris. *An Extensible Virtual Machine Architecture*. In Proceedings of the OOPSLA'99 Workshop on Simplicity,

Performance and Portability in Virtual Machine Design, November 1999. (p 147)

[Hawblitzel98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. *Implementing Multiple Protection Domains in Java*. In 1998 USENIX Annual Technical Conference, pages 259--270, June 1998. (pp 146, 170, 170)

[Hennessy90] John L. Hennessy and David A. Patterson. *Computer Architecture -- A Quantitative Approach*. Morgan Kaufmann, Los Altos, CA 94022, USA, 1990. (pp 21, 66)

[Heydon00] Allan Heydon and Marc Najork. *Performance limitations of the Java core libraries*. Concurrency: Practice and Experience, 12(6):363--373, May 2000. Also appeared at the 1999 ACM Java Grande conference, Palo Alto. (p 66)

[Holyer98] Ian Holyer and Eleni Spiliopoulou. *The Brisk Machine: A Simplified STG Machine*. Technical Report CSTR-98-003, Department of Computer Science, University of Bristol, March 1998. (p 20)

[Hölzle92] Urs Hölzle, Craig Chambers, and David Ungar. *Debugging optimized code with dynamic deoptimization*. In Programming Language Design and Implementation (PLDI '92), volume 27(7) of *ACM SIGPLAN Notices*, pages 32--43, July 1992. (p 26)

[Hölzle94a] Urs Hölzle. *Adaptive Optimization for SELF: reconciling high Performance with Exploratory Programming*. Thesis

CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994. (pp 26, 27, 70)

[Hölzle94b] Urs Hölzle and David Ungar. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*. In Programming Language Design and Implementation (PLDI '94), volume 29(6) of *ACM SIGPLAN Notices*, pages 326--336, June 1994. (p 26)

[Hölzle95] Urs Hölzle and David Ungar. *Do Object-Oriented Languages Need Special Hardware Support?* In Walter G. Olthoff, editor, *European Conference on Object-Oriented Programming (ECOOP '95)*, volume 952 of *Lecture Notes in Computer Science*, pages 283--302. Springer-Verlag, August 1995. (p 27)

[Ingalls84] Daniel H. H. Ingalls. *The evolution of the Smalltalk virtual machine*. In Smalltalk-80: Bits of History, Words of Advice, pages 9--28. Addison-Wesley, Reading, 1984. (pp 22, 23)

[Johnstone97] Mark S. Johnstone and Paul R. Wilson. *The Memory Fragmentation Problem: Solved?* In OOPSLA '97 Workshop on Garbage Collection and Memory Management, volume 34(4) of *ACM SIGPLAN Notices*, pages 26--36, October 1997. (p 73)

[Jones96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. (p 76)

[Jones97] Michael B. Jones, Daniela Rosu, and Marcel-Ctlin Rosu. *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities*. In Symposium on Operating Systems

Principles (SOSP '97), volume 31(5) of *Operating Systems Review*, pages 198--211, October 1997. (p 167)

[Kaashoek97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. *Application Performance and Flexibility on Exokernel Systems*. In Symposium on Operating Systems Principles (SOSP '97), volume 31(5) of *Operating Systems Review*, pages 52--65, October 1997. (pp 39, 40, 41, 42, 47, 144)

[Kiczales97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. In European Conference on Object-Oriented Programming (ECOOP '97), volume 1241 of *Lecture Notes in Computer Science*, pages 220--242, 1997. (p 60)

[Kistler96] Thomas Kistler and Michael Franz. *Slim Binaries*. Technical Report 96-24, Department of Information and Computer Science, University of California, Irvine, June 1996. (p 36)

[Krasner84] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, 1984. (p 22)

[Lawless91] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: the Common Lisp Object System*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1991. (p 25)

[Leslie96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden.

The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal on Selected Areas In Communications, 14(7):1280--1297, September 1996. (pp 42, 167, 167)

[Liang98] Sheng Liang and Gilad Bracha. *Dynamic Class Loading in the Java Virtual Machine*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '98), volume 33(10) of *ACM SIGPLAN Notices*, pages 36--44, October 1998. (p 28)

[Liedtke98] Jochen Liedtke, Nayeem Islam, Trent Jaeger, Vsevolod Panteleenko, and Yoonho Park. *An Unconventional Proposal: Using the x86 Architecture As The Ubiquitous Virtual Standard Architecture*. In Eighth ACM SIGOPS European Workshop, September 1998. (p 34)

[Lindholm97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997. (pp 16, 28, 28, 59, 108, 114)

[Lippert99] Martin Lippert and Cristina Videira Lopes. *A study on exception detecting and handling using aspect-oriented programming*. Technical Report P9910229 CSL-99-1, Xerox Palo Alto Research Center, December 1999. (p 61)

[Lopes98] Christina Videira Lopes and Gregor Kiczales. *Recent developments in AspectJ*. In European Conference on Object-Oriented Programming (ECOOP '98), volume 1543 of *Lecture Notes in Computer Science*, pages 398--401, 1998. (p 60)

[McDowell98] Charlie E. McDowell and E. A. Baldwin. *Unloading Java Classes That Contain Static Fields*. ACM SIGPLAN Notices, 33(1):56--60, January 1998. Also available as University of California, Jack Baskin School of Engineering Technical Report UCSC-CRL-97-18. (p 28)

[McGhan98] Harlan McGhan and Mike O'Connor. *PicoJava: A direct execution engine for Java bytecode*. Computer, 31(10):22--30, October 1998. (p 117)

[Mehl95] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. *An Abstract Machine for Oz*. Lecture Notes in Computer Science, 982:151--168, 1995. Also appears as Research Report RR-95-08, Deutsches Forschungszentrum für Künstliche Intelligenz. (p 20)

[Menage00] Paul Menage. *Resource control of untrusted code in an open programmable network*. PhD thesis, University of Cambridge Computer Laboratory, June 2000. (p 172)

[Menage99] Paul Menage. *RCANE: a resource controlled framework for active network services*. In Proceedings of the First International Working Conference on Active Networks (IWAN '99), volume 1653 of *Lecture Notes in Computer Science*, pages 25--36, July 1999. (p 170)

[Milner92] Robin Milner, Joachim Parrow, and David Walker. *A Calculus of Mobile Processes, Part I/II*. Information and Computation, 100(1):1--77, 1992. (p 80)

[Milner97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML (revised)*. MIT Press, Cambridge, MA, USA, 1997. (p 52)

[Montz95] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting. *Scout: A Communications-Oriented Operating System*. In Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-V), May 1995. (p 38)

[Mulet95] Philippe Mulet, Jacques Malenfant, and Pierre Cointe. *Towards a Methodology for Explicit Composition of MetaObjects*. ACM SIGPLAN Notices, pages 316--330, October 1995. (p 97)

[Myers97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. *Parameterized Types for Java*. In Proceedings of the twenty-fourth ACM Symposium on Principles of Programming Languages (POPL '97), pages 132--145, Paris, France, 15--17 January 1997. (p 79)

[Nelson, editor91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991. (pp 22, 33, 40)

[Nori81] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and Ch. Jacobi. *Pascal-P Implementation Notes*. In Pascal -- The Language and its Implementation, pages 125--170. John Wiley Sons, Ltd., 1981. (p 21)

[Odersky97] Martin Odersky and Philip Wadler. *Pizza into Java: Translating theory into practice*. In Proceedings of the twenty-fourth ACM Symposium on Principles of Programming Languages (POPL '97), pages 146--159, Paris, France, 15--17 January 1997. (pp 54, 79)

[PPro2] *Pentium Pro Family Developer's Manual, volume 2, programmer's reference manual*. Intel Corporation, 1996. Reference number 242691-001. (p 115)

[Printezis01] Tony Printezis and David Detlefs. *A Generational Mostly-Concurrent Garbage Collector*. In International Symposium on Memory Management (ISMM '00), volume 36(1) of *ACM SIGPLAN Notices*, January 2001. (pp 76, 182)

[Proebsting97] Todd A. Proebsting and Scott A. Watterson. *Krakatoa: Decompilation in Java*. In Conference on Object-Oriented Technologies and Systems (COOTS '97), pages 185--197. The USENIX Association, 1997. (p 29)

[Qian99] Zhenyu Qian. *Least types for memory locations in (Java) bytecode*. In Proceedings of the sixth International Workshop on Foundations of Object-Oriented Languages (FOOL 6), January 1999. Also available as part of Kestrel Institute Technical Report KES.U.99.6. (p 29)

[Reed99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. *Xenoservers: accounted execution of untrusted code*. In Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII), 1999. (p 171)

[Reppy91] John H. Reppy. *CML: A Higher-Order Concurrent Language*. In Programming Language Design and Implementation (PLDI '91), volume 26(6) of *ACM SIGPLAN Notices*, pages 293--305, June 1991. (pp 150, 151)

[Richards79] Martin Richards and Colin Whitby-Stevens. *BCPL -- the language and its compiler*. Cambridge University Press, 1979. (p 20)

[Robson74] J. M. Robson. *Bounds for Some Functions Concerning Dynamic Storage Allocation*. Journal of the ACM, 21(3):491--499, July 1974. (p 73)

[Roscoe95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. Technical Report 376, University of Cambridge Computer Laboratory, August 1995. (p 167)

[Ruf00] Erik Ruf. *Effective synchronization removal for Java*. In Programming Language Design and Implementation (PLDI '00), volume 35(5) of *ACM SIGPLAN Notices*, pages 208--218, May 2000. (p 162)

[Scedrov95] Andre Scedrov. *Linear Logic and Computation: A Survey*. In H. Schwichtenberg, editor, *Proof and Computation, Proceedings Marktoberdorf Summer School 1993*, pages 379--395. NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1995. (p 145)

[Seltzer97] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. *Issues in extensible operating systems*. Technical Report TR-18-97, Harvard University EECS, November 1997. (p 38)

[Serrano00] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. *Quicksilver: a quasi-static compiler for Java*. In Object-Oriented Programming, Systems, Languages &

Applications (OOPSLA'00), volume 35(10) of *ACM SIGPLAN Notices*, pages 66--82, October 2000. (p 69)

[Sirer97] Emin Gün Sirer, Przemyslaw Pardyak, and Brian N. Bershad. *Strands: An Efficient and Extensible Thread Management Architecture*. Technical Report TR-97-09-01, University of Washington, Department of Computer Science and Engineering, September 1997. (p 150)

[Sirer98] Emin Gün Sirer, Robert Grimm, Brian N. Bershad, Arthur J Gregory, and Sean McDirmid. *Distributed Virtual Machines: A System Architecture for Network Computing*. In Eighth ACM SIGOPS European Workshop, September 1998. (p 63)

[Solorzano98] Jose H. Solorzano and Suad Alagie. *Parametric Polymorphism for Java: A reflective solution*. In Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98), volume 33(10) of *ACM SIGPLAN Notices*, pages 201--215, October 1998. (p 79)

[Spoonhower98] Daniel Spoonhower, Grzegorz Czajkowski, Chris Hawblitzel, Chi-Chao Chang, Deyu Hu, and Thorsten von Eicken. *Design and Evaluation of an Extensible Web & Telephony Server based on the J-Kernel*. Technical Report TR98-1715, Cornell University, Computer Science, November 1998. (p 170)

[Steele, Jr.90] Guy Steele, Jr. *Common lisp: The Language*. Digital Press, Bedford, Massacusetts, 2nd edition, 1990. (p 52)

[Steere99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. *A Feedback-driven*

Proportion Allocator for Real-Rate Scheduling. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99), pages 145--158, Berkeley, CA, February 1999. The USENIX Association. (p 167)

[Stefanovic94] Darko Stefanovic and J. Eliot B. Moss. *Characterisation of Object Behaviour in Standard ML of New Jersey*. In Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming, pages 43--54. ACM Press, June 1994. (p 72)

[Stroustrup97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 3 edition, 1997. (p 22)

[Taft96] Tucker Taft. *Programming the Internet in Ada 95*. In Ada-Europe International Conference on Reliable Software Technologies, volume 1088 of *Lecture Notes in Computer Science*, pages 1--16. Springer-Verlag, June 1996. (pp 54, 58)

[Tanenbaum92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliff, NJ, 1992. (p 39)

[Tatsubori98] Michiaki Tatsubori and Shigeru Chiba. *Programming support of design patterns with compile-time reflection*. In Proceedings of the OOPSLA '98 workshop on reflective programming in C++ and Java, pages 56--60, October 1998. (p 62)

[Tatsubori99] Michiaki Tatsubori. *An extension mechanism for the Java language*. Master's thesis, Graduate School of Engineering, University of Tsukuba, February 1999. (p 62)

[Tennenhouse89] David L. Tennenhouse. *Layered Multiplexing Considered Harmful*. In Proceedings of the 1st International Workshop on High-Speed Networks. Elsevier Science Publishers, IFIP, May 1989. (p 43)

[Thomas99] Reuben Thomas. *Mite: a basis for ubiquitous virtual machines*. PhD thesis, University of Cambridge Computer Laboratory, September 1999. (p 31)

[van Deursen00] Arie van Deursen, Paul Klint, and Joost Visser. *Domain-Specific Languages: An Annotated Bibliography*. ACM SIGPLAN Notices, 35(6):26--36, June 2000. (p 83)

[Viega98] John Viega, Bill Tutt, and Reimer Behrends. *Automated delegation as a viable alternative to multiple inheritance in class based languages*. Technical report CS-98-03, University of Virginia, 1998. (p 61)

[Vo96] Kiem-Phong Vo. *Vmalloc: A General and Efficient Memory Allocator*. Software --- Practice and Experience, 26(3):357--374, March 1996. (p 72)

[Waldo91] Jim Waldo. *Controversy: The Case for Multiple Inheritance in C++*. In Computing Systems, volume 4, pages 157--172. The USENIX Association, 1991. (p 61)

[Wallach97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. *Extensible Security Architecture for Java*. In Symposium on Operating Systems Principles (SOSP '97), volume 31(5) of *Operating Systems Review*, pages 116--128, New York, October 1997. (p 103)

[Wand80] Mitchell Wand. *Continuation-Based Multiprocessing*. In Conference Record of the 1980 LISP Conference, Stanford University, pages 19--28, New York, NY, 1980. ACM. (p 150)

[Webster81] C. A. G. Webster. *Pascal in education*. In Pascal -- The Language and its Implementation, pages 37--48. John Wiley Sons, Ltd., 1981. (p 21)

[Welsh00] Matt Welsh and David Culler. *Jaguar: enabling efficient communication and I/O in Java*. Concurrency: Practice and Experience, 12(7):519--538, May 2000. (p 147)

[Wilson92] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. In International Workshop on Memory Management (IWMM '92), volume 637 of *Lecture Notes in Computer Science*, September 1992. (p 74)

[Wilson95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. *Dynamic Storage Allocation: A survey and critical review*. In International Workshop on Memory Management (IWMM '95), volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, UK, 1995. Springer-Verlag. (pp 73, 128)

[Wirth71] Niklaus Wirth. *The Programming Language Pascal*. Acta Informatica, 1(4):35--63, 1971. (p 21)

[Wu98] Peng Wu, Samuel P Midkiff, José E Moreira, and Manish Gupta. *Improving Java performance through semantic inlining*. Technical report RC-21313-(96030), IBM Research Division, October 1998. (p 147)

[Zorn98] Benjamin G. Zorn and Matthew L. Seidl. *Segregating Heap Objects by Reference Behavior and Lifetime*. In Architectural Support for Programming Langauages and Operating Systems (ASPLOS '98), volume 33(11) of *ACM SIGPLAN Notices*, pages 12--23, November 1998. (pp 71, 131)