Revolutionizing Embedded Software

Kasper V. Lund and Jakob R. Andersen

Master’s Thesis

Department of Computer Science
University of Aarhus
Denmark

Abstract

More than 90% of the microprocessors produced today are used in embedded devices. With the current development tools, it is exceedingly difficult to debug, profile, and update code running on embedded devices in operation. This leaves developers unable to diagnose and solve software issues on deployed embedded systems, something that is unacceptable for an industry where robustness is paramount.

In this thesis, we show that it is possible to build a fully serviceable software platform that fits on memory-constrained embedded devices. We use virtual machine technology to enable full serviceability even for system software components. At the bottom of the software stack, we have replaced real-time operating systems with an efficient 30 KB object-oriented virtual machine. The virtual machine contains a reflective interface that allows developers to debug, profile, and update code running on embedded devices even in operation. The serviceability extends to system software components, including interrupt handlers, device drivers, and networking protocols. Like any other components, the system software components are implemented in safe, compact virtual machine instructions.

Our virtual machine uses an interpreter to execute both system software and applications. On average, our interpreter is more than twice as fast as the closest competitor for low-end embedded devices. It even outperforms the fastest Java interpreter available. Compared to other object-oriented virtual machines, our compact memory representation of objects allows us to reduce the amount of memory spent on classes, methods, and strings by 40–50%. The result is that our entire software stack fits in less than 128 KB of memory. This way, our platform enables serviceability on a wide range of industrial and consumer devices; something we believe will revolutionize the way embedded software is developed and maintained.

Preface

The work presented in this thesis was done in OOVM A/S, a small startup company consisting of the two authors and Lars Bak. The mission of OOVM A/S is to improve reliability, availability, and servicability of embedded software by introducing a new software platform. The platform consists of several components. The design of the components is the result of animated discussions at the whiteboard between the three of us. This thesis will focus on the virtual machine and the system software, both of which were implemented by the authors. The programming environment, source code compiler, and garbage collector were implemented by Lars Bak.

We wish to thank our thesis supervisor, Ole Lehrmann Madsen, for encouraging us to focus on the written part of the thesis in addition to the software implementation. We also wish to thank Lars Bak, as well as Steffen Grarup who has recently joined the OOVM team. Both have made themselves available for technical discussions, and have provided useful feedback on the different parts of this thesis. We look forward to continuing to work together with you in the future. Furthermore, Mads Torgersen deserves special thanks for many enlightening discussions on object-orientation, reviewing the thesis, and for always bringing cake to our meetings. Finally, we wish to thank all the others who have read and provided feedback on this thesis. We really appreciate your efforts in helping us ensure the accuracy and readability of this thesis.

Aarhus, May 2003

Contents

List of Figures

Chapter 1

Introduction

This thesis presents a new platform for embedded software development. The platform is based on a small object-oriented virtual machine, which runs directly on hardware without the need for an underlying operating system. The platform is fully serviceable; developers can debug, profile, and update code running on embedded devices in operation. The serviceability extends to system software components, including interrupt handlers, device drivers, and networking protocols. Like any other components, the system software components are implemented in safe, compact virtual machine instructions.

1.1 Motivation

More than 90% of the microprocessors produced today are used in embedded devices. It is estimated that each individual in the United States interacts with about 150 embedded systems every day, whether they know it or not [Gat03]. The embedded systems include everything from refrigerators to mobile phones, so naturally there are many variations on embedded hardware architectures. Despite the variations, embedded systems generally need software to function. The embedded industry spends more than 20 billion dollars every year developing and maintaining software for its products.

Developing software for embedded devices has traditionally been slow. The source code is compiled and linked on the development platform, and the resulting binary image is transferred onto the device. If the source code is changed, the entire process must be restarted from compilation. This way, it can easily take several minutes to effectuate a change. This severely limits software productivity and it is a huge problem in an industry where time-to-market can make the difference between success and failure.

Another problem that faces embedded software developers is the lack of serviceability. With the current development tools, it is exceedingly difficult to debug, profile, and update code running on embedded devices in operation. Debugging and profiling is sometimes supported during development. It is achieved by instrumenting the compiled code before downloading it to the device. The instrumented code is typically larger and slower than the non-instrumented version. For this reason, the support for debugging and profiling is removed as a part of the release process. This leaves developers unable to diagnose and solve software issues on deployed embedded systems, something that is unacceptable for an industry where robustness is paramount.

The Java 2 Micro Edition platform has been proposed as the solution to the serviceability problems for embedded devices. It comes with a debugging interface that allows developers to diagnose problems even on operating devices. However, there are no remote profiling capabilities and it is impossible to update software without restarting most parts of the running system. Furthermore, the runtime environment for Java requires more memory than what is available on most low-end embedded devices. For these reasons, many producers of embedded systems are reluctant to use Java as the foundation for their software.

1.2 Goals

The purpose of this thesis is to show that it is feasible to use virtual machine technology to solve the serviceability issues inherent in existing embedded software platforms. We want to show that it is possible to design and implement efficient virtual machines that fit on memory-constrained embedded devices, and that such virtual machines can enable full serviceability even for devices in operation. Furthermore, we want to show that it is both feasible and beneficial to replace existing embedded real-time operating systems with system software components running on top of a virtual machine.

1.3 Philosophy

We have designed and implemented a complete embedded software platform, including a programming environment, an object-oriented virtual machine, and the system software to run on top of it. The system is available in several configurations spanning two different hardware architectures. See appendix A for further details on the configurations. We have managed to implement a complete system because we have favored simplicity over complexity. This is by no means as trivial as it may sound. Systems that evolve get increasingly complex. Simplicity is achieved only through conscious design.

There are many different situations where complexity may arise. Most optimizations add complexity to the system. However, it is well known that in many cases at least 80% of the running time is spent in at most 20% of the code. For that reason, it is important to avoid premature optimizations and the unnecessary complexity that follows. The key is to avoid adding any complexity to the large fraction of the code that is not crucial for performance. This strategy also results in much less code; something that is vital for memory-constrained embedded devices.

Optimizations require measurements. To improve performance it must be known where time is spent. Measurements are much more accurate if they are performed on complete systems, rather than on proof-of-concept systems or prototypes. There are many cases where measurements done on prototypes have exaggerated the importance of certain low-level optimizations. This is one of the reason why we have implemented a complete system, focusing initially on completeness rather than performance.

1.4 Overview

Our platform for embedded software development is based on an object-oriented virtual machine. The following chapters will describe the design and implementation of our platform, and compare it to state-of-the-art virtual machines and embedded software development platforms.

Chapter 2 describes the high-level programming language we have chosen as the primary interface for our software platform. We have chosen a simple yet powerful language that allows us to keep the virtual machine small.

Chapter 3 describes our virtual machine in details. We discuss object models and execution models as well as several performance optimizations. The discussion covers state-of-the-art virtual machine technology using both Smalltalk and Java^{TM} implementations as examples.

Chapter 4 describes how we have implemented a programming environment, which uses the virtual machine to improve software development for embedded systems, by enabling high software productivity.

Chapter 5 describes the design and implementation of the system software for an embedded device. The system software allows us to run our virtual machine directly on hardware without the need for an underlying operating system.

Chapter 6 compares the performance and footprint of our platform to other available platforms. The results of the experiments in this chapter are used to substantiate our claims throughout the thesis.

Chapter 7 concludes our thesis by summarizing our contributions and the conclusions from the preceeding chapters. In this chapter, we also provide a glimpse of future work and research directions.

Chapter 2

Programming Language

The choice of programming language is important, because it has implications for most parts of our system. This includes the virtual machine, the software development tools, and the system software. For this reason, we have a number of requirements to the programming language.

Compactness The runtime support required to run programs written in the language must be able to fit on small embedded devices. The language must be simple and extensible, without too much built-in functionality. A simple language will allow us to build a simple and compact virtual machine, which fits on memory-constrained embedded devices.

Object-orientation The language must support abstracting, factoring, and code reuse. Most object-oriented languages support these. Furthermore, programmers familiar with object-oriented programming should be able to take advantage of their skills and immediately start developing for embedded devices.

Serviceability The language must support incremental software development. It must be possible to do fine-grained updates at runtime. This includes changing methods, classes, and objects without having to restart the system. A platform with such functionality is an improvement over current solutions for embedded software development.

We have chosen Smalltalk as the programming language for our embedded software platform. Smalltalk is a dynamically typed, pure object-oriented language developed by Xerox Palo Alto Research Center (PARC) in the beginning of the 1970s. Compared to other object-oriented languages, we have found that the syntax and semantics of Smalltalk are simple and very consistent, and we will show that Smalltalk meets all the criteria outlined above.

2.1 Smalltalk

Smalltalk is a pure object-oriented language based on *message passing*. Computation is done by sending messages to objects. The objects respond to messages by executing methods. Everything in Smalltalk is an object, even integers and booleans. Therefore, the message passing metaphor extends to integer arithmetic, which is performed by sending messages such as `+` or `/`.

Smalltalk is dynamically typed. This means that variables have no type annotations and that all variables can refer to objects of any type. Dynamic typing speeds up the development process at the cost of type safety. However, it is possible to add static type systems to Smalltalk. The Strongtalk type system described in [BG93] is an example of this. It is an optional, incremental type system for Smalltalk-80. We have not yet experimented with such static typing for our system.

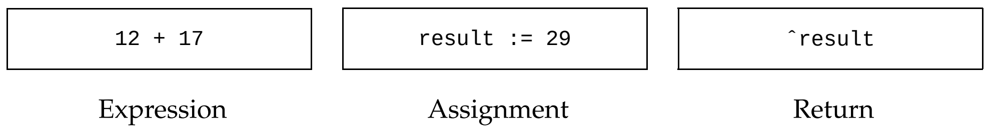
It is important to emphasize that we have implemented only the Smalltalk language; not a full Smalltalk system. Smalltalk systems have extensive class libraries with support for reflection. We have chosen to minimize the class libraries by removing functionality that is seldom used. Furthermore, we have moved the code that handles reflection to the programming environment. Since our programming environment runs on the development platform, this makes our libraries much more suitable for use on memory-constrained embedded devices.

2.1.1 Syntax and Semantics

This section gives a brief introduction to the syntax and semantics of Smalltalk. In the interest of readability, the description of the syntax has been simplified. See [Com97] for an accurate Smalltalk grammar. Methods in Smalltalk consist of a sequence of statements separated by periods. Each statement is either an expression, an assignment, or a return statement. Figure 2.1 on the facing page shows the syntax of each of the three.

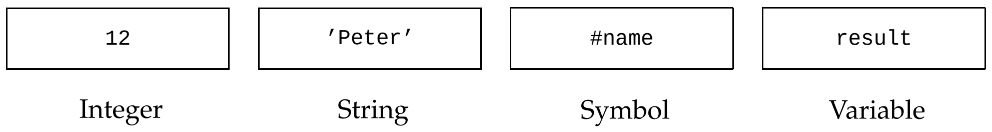
Expressions are built from the constructs shown in figure 2.2 on the next page. Notice that strings are enclosed in apostrophes. Quotation

Figure 2.1 Statement syntax in Smalltalk



marks are reserved for comments. Smalltalk insists that variables are defined before they are used. Variables local to a method must be defined at the beginning of the method. Such a definition consists of a whitespace-separated list of variables enclosed in vertical bars. To define `a` and `b` as method-local variables, insert `| a b |` at the beginning of the method.

Figure 2.2 Simple expression syntax in Smalltalk



The simple expressions from figure 2.2 can be used in *message sends*. Message sends are expressions used for invoking methods. The messages are sent to objects. The *receiver* of a message responds by invoking one of its methods. The receiver determines which method to invoke by looking at the message *selector*. Figure 2.3 on the following page shows the three different forms of sends supported by Smalltalk. The receiver is always the first part of the send. It is followed by the selector and the arguments if any. The selectors are shown in bold. Notice how the arguments to keyword sends are interleaved with the keywords.

Two variables have special meanings in Smalltalk: `self` and `super`. The `self` variable always refers to the current receiver. Methods that do not contain a return statement implicitly return `self`. The `super` variable is similar to `self`, but it can only be used as the receiver in send expressions. It is used to invoke methods that are otherwise inaccessible due to subclass method overriding.

The Smalltalk-80 specification only defines syntax for methods. This is because the standard development tools for Smalltalk-80 are based on graphical user interfaces that rely on non-textual interaction for defining

Figure 2.3 Send syntax in Smalltalk

`12 negated`	`12 + 17`	`array at: 12 put: 17`
Unary send	Binary send	Keyword send

classes. We have defined a textual syntax for classes. Figure 2.4 shows the definition of a point class, which inherits from the object class. The `| x y |` defines two object variables: `x` and `y`. The rest of the definition consists of method declarations. The `new` method is special. It is a method defined for the point class, not for its instances. The method is used in unary sends, such as `Point new`, to allocate new instances of `Point`.

Figure 2.4 Class definition for the point class

```
Point = Object (
  | x y |

  x = ( ^x )
  y = ( ^y )

  ...

  initialize = ( x := 0. y := 0 )

) class (

  new = ( ^super new initialize )

)
```

During the development of our system, it has been convenient to have a textual representation of classes. It has allowed us to use existing revision control systems, which work well on textual files. This has enabled us to have multiple developers working on our Smalltalk source code simultaneously. As our programming environment improves, the need for a textual class representation decreases. However, we will use the textual representation in code examples throughout this thesis.

2.1.2 Blocks

Smalltalk blocks are statements enclosed in square brackets. Figure 2.5 shows two expressions that contain such enclosed statements. The statements in a block are only evaluated when `value` is sent to the block. This is also illustrated by figure 2.5, where the benchmark is only run by the expression to the left. The result of evaluating the expression to the right is simply the block itself. Blocks are *closures*: The statements in a block are evaluated in the context that created the block; not in the context that sent `value` to the block.

Figure 2.5 Evaluation of blocks

```
[ Benchmark run ] value
```

Benchmark is run

```
[ Benchmark run ]
```

Benchmark is *not* run

Blocks are used for implementing control structures. Figure 2.6 shows the Smalltalk variant of *if-then-else* conditional processing. The condition expression `self < 0` is evaluated and the result is either `true` or `false`. The `ifTrue:ifFalse:` method on `true` only sends `value` to the first block. Thus, if the receiver is less than zero, the statements in the second block are never evaluated. Similarly, the `ifTrue:ifFalse:` method on `false` only sends `value` to the second block.

Figure 2.6 Computing the absolute value of an integer

```
abs = (
  self < 0 ifTrue: [ ^self negated ] ifFalse: [ ^self ]
)
```

The statements in a block are usually evaluated outside the method they are written in. Figure 2.7 on the following page shows a method that tests if a collection includes a given element. The `do:` method iterates over all elements in the collection. The evaluation of the return statement in the inner block is initiated by the `ifTrue:` method on `true`. According to the semantics of Smalltalk, the evaluation of the return statement in the block must return from the `includes:` method. This kind of return is a *non-local return*, because `includes:` is not the method that initiated the evaluation. Blocks that do not contain return statements implicitly return the value of their last expression to the sender of `value`. This kind of return is a *local return*. The last return statement in the `includes:` method is also a local return. It returns to the sender of `includes:`.

Figure 2.7 Testing collection membership

```
includes: element = (
  self do: [ :e | element = e ifTrue: [ ˆtrue ] ].
  ˆfalse
)
```

We have extended the Smalltalk-80 syntax with static type annotations for blocks. By enclosing a local variable or an argument in square brackets, the programmer expresses that it will always contain a block. Figure 2.8 shows an example of an annotated argument. The annotations allow us to improve the performance of block-intensive code considerably. The performance impact is explored in more details in section 3.2.2.2.

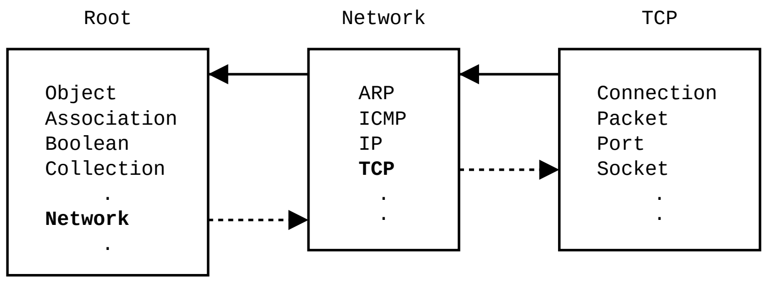
Figure 2.8 Static type annotation for block arguments

```
do: [block] = (
  1 to: (self size) do: [ :i | block value: (self at: i) ].
)
```

2.1.3 Namespaces

In standard Smalltalk-80, all classes are global; they are visible from everywhere. This visibility is excessive and problematic. It can easily cause class naming conflicts. To solve this problem, we have extended Smalltalk with hierarchical namespaces. Namespaces contain classes and optionally nested namespaces. Figure 2.9 on the facing page illustrates this by showing some of the namespaces used in our TCP/IP implementation. Class visibility follows common scope rules: All classes declared in a namespace are visible from any nested namespace. For the TCP/IP implementation this implies that `Collection` is visible to code in `Network` and `TCP`. In our system, namespaces are classes; they can have both instances and behavior. In many ways, our design is similar to the subsystems design described in [Boy96]. The main difference is that we have chosen to resolve class names statically when compiling methods. The subsystems resolve class names dynamically at runtime.

Figure 2.9 Hierarchical namespaces for TCP/IP



Sometimes, it is necessary to access classes in one namespace from another namespace. To support this, we have extended the Smalltalk-80 syntax with the scope resolution operator `::` known from C++. To illustrate this, consider figure 2.10 which shows part of our implementation of an *echo service*. The echo service runs as a separate thread. It accepts incoming TCP connections and sends back the input it receives on these connections. In the implementation, it is necessary to access the `Port` in the `TCP` namespace from another namespace.

Figure 2.10 Remote access to `Port` in the `TCP` namespace

```
run = (
  | port |
  port := Network::TCP::Port new bind: EchoPort.
  port listen: 5.
  ...
)
```

We have found that it is convenient to have namespaces in Smalltalk. We have used namespaces extensively for structuring our libraries. Furthermore, we have used them for source code configuration management. To achieve this, we have added configuration information to the namespaces. Each namespace is either global, in the sense that it is available in all configurations, or local to a specific configuration. We allow multiple namespaces with the same name as long as they are for different configurations. As an example, the `Driver` namespace for our Linux configuration consists of a `SLIP` (Serial Line IP) driver, whereas the same namespace for our CerfCube configuration includes drivers for `CS8900A` (Ethernet) and `GPIO` (General Purpose I/O). The source code compiler takes the current configuration into account when it resolves class names. Appendix A describes the hardware and software configurations we support.

Chapter 3

Virtual Machine

Virtual machines are runtime environments that execute programs written in non-native instructions. As such they are software implementations of platforms for executing software. Inspired by silicon-based platforms, the defining property of a virtual machine is its binary-encoded instruction set. The instruction set acts as an interface between high-level language compilers and the execution engine of the virtual machine.

There are many advantages to implementing platforms in software. Software implementations can be ported to different hardware platforms, thus enabling users to choose the most cost-effective solution. Software implementations are also more manageable, and since experimenting with source code is less costly than changing hardware, it allows for more innovation. These are some of the selling points not only for virtual machines, but for platform independent high-level languages and runtime systems in general. As an example of constraints imposed by hardware, consider the ARM® Jazelle™ technology. Jazelle is a hardware implementation of an interpreter for most of the instructions used in embedded Java. The hardware interpreter is intended to be used with existing virtual machines as an accelerator, but because the hardware logic assumes certain memory layouts for various structures, such as the execution stack, it is difficult for virtual machine programmers to change and improve fundamental parts of their implementation.

During the last decade, *object-oriented virtual machines* have gained a lot of attention, mainly due to the popularity of Java. In the past, object-oriented virtual machines have been used for implementing languages such as Smalltalk and SELF. The execution engine of an object-oriented virtual machine is similar to that of other virtual machines, except that support for dynamic dispatching and type checking is included. The distinguishing aspect of object-oriented virtual machines is the presence of an object-based memory model, coupled with a garbage collection system for doing automatic storage reclamation.

The notions of object models and execution models are central to object-oriented virtual machines. The object model describes how objects are represented in memory, whereas the execution model handles the execution of instructions. In the following sections, we will discuss the design and implementation of these two models in the context of state-of-the-art object-oriented virtual machines. Furthermore, we will describe and evaluate the virtual machine that constitutes the bottom layer of our software stack for embedded devices.

3.1 Object Model

An object model is a description of the state and structure of individual objects and the relations that group them into logical entities. In this section, we will highlight different implementation strategies, and give a detailed description of the design and implementation of the object model upon which our virtual machine rests.

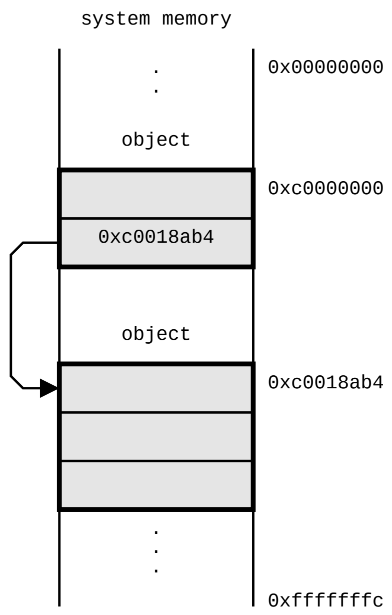
Object are stored in a part of the system memory known as the *object heap*. References between objects, normally called object pointers, come in two variants: Direct and indirect. In the direct pointer model, an object pointer is the memory address of the object being pointed to. This pointer model is used in C++. Figure 3.1 on the facing page shows an object containing a direct pointer to another object in the system memory.

Using direct object pointers is simple and performance-wise efficient, but it complicates relocation of objects. Relocating an object involves copying its contents and changing its memory address, and therefore all object pointers that point to the object have to be updated. The number of such object pointers is only bounded by the total number of object pointers in the heap, which means that in the worst-case scenario the time it takes to move an object is proportional to the used size of the heap.

The original Smalltalk-80 implementation represents object pointers as indexes into an object table. The object table holds direct object pointers. The situation is shown in figure 3.2 on page 16. With indirect pointers, the time it takes to move an object is bounded by the size of the object. Apart from copying the contents of the object, there is only one object table entry that needs to be updated.

Even though an object table seems like a simple solution, there are many problems associated with it. When allocating and deallocating objects, object table entries must also be allocated and deallocated. This con-

Figure 3.1 Object representation using direct pointers

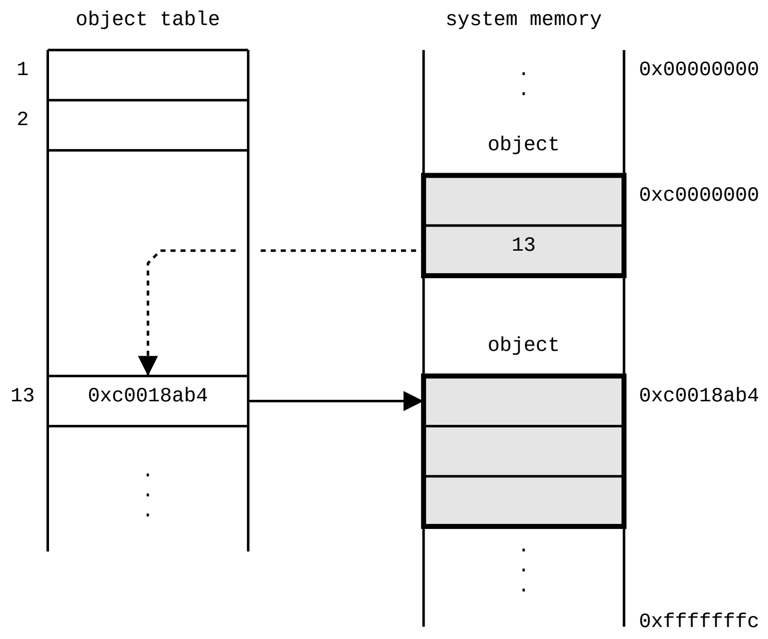


sumes time and complicates garbage collection strategies based on copying, since the non-live objects must be identified and their associated table entries must be deallocated. Furthermore, having a table of pointers means having one more resource to manage. In general, it is very difficult to come up with a good heuristic for keeping the object table at a reasonable size. The following table summarizes the problems with the two pointer models discussed:

	Direct	**Indirect**
Object access requires extra indirection		•
Object table entry must be allocated for all objects		•
Moving an object requires unbounded updates	•	
Copying garbage collection must treat dead objects		•
Object table must be maintained and resized		•

The direct pointer model is the most commonly used. It is used in most modern implementations of Java and Smalltalk. Based on the listed problems, we have chosen to use direct 32-bit object pointers. Further implications of direct and indirect pointers on garbage collection will be discussed in section 3.1.5.

Figure 3.2 Indirect pointers through object table



3.1.1 Objects and Classes

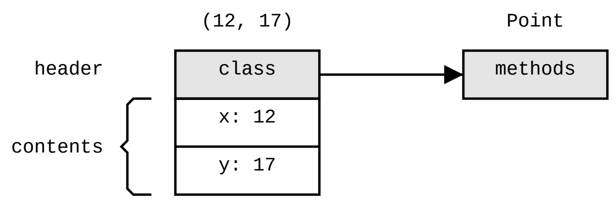
The use of objects in programming languages dates back to Simula, where objects were used to model the phenomena involved in system simulations. Throughout this thesis, we will consider objects as abstract data types, which encapsulate both state and behavior.

There are two kinds of object-based systems: *Class-based* and *prototype-based*. Most modern systems are class-based. In such systems, objects are created from classes. Classes enforce structure; all objects of a given class share the same behavior and representation. In prototype-based systems, objects are created from other objects via cloning. The cloned objects can be customized; there is no enforced structure. Most practical implementations of prototype-based object models include objects that represent immutable shared state for a group of objects. In the implementation of the SELF system, these shared state objects are known as *maps*, and as noted in [CUL89] they bear close resemblance to classes. The distinguishing differences between class-based and prototype-based object models exist primarily at the syntactical and semantical levels. Since prototype-based languages are seldom used, we will consider only class-based object models.

Further implications of the use of prototypes in object-oriented languages are discussed in [Bor86].

Even though there are many variations on the structure of object models, there is a common foundation. Objects are divided into two parts. The first part is known as the *object header*. It consists of fields needed by the virtual machine. Normally, the object header cannot be accessed directly from within the programming language. The second part is known as the *object contents*, and it contains fields for variables described by the programming language. Figure 3.3 shows the relations between a simple object, a point, and its class. The point has two coordinate variables, `x` and `y`, that constitute the state of the point. In addition to the coordinates, the point holds a reference to the point class, which describes the shared behavior of all points through a set of methods. Methods are described thoroughly in section 3.1.3.

Figure 3.3 Relations between a point and its class

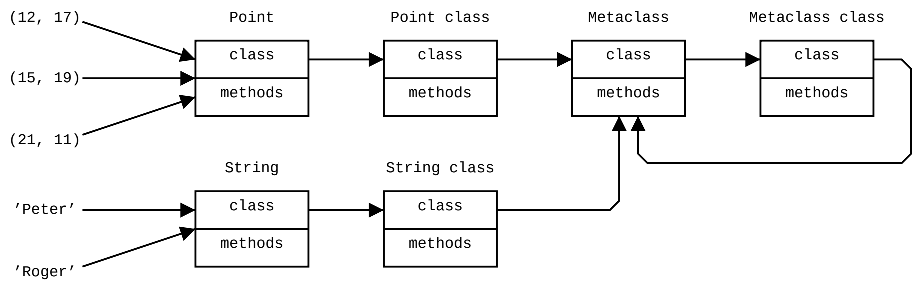


Some languages *require* classes themselves to be represented as objects. In Smalltalk, classes are used as factories for their instances, and therefore it is convenient that they have both state and behavior. Since object behavior is defined in classes, Smalltalk classes have classes of their own. Figure 3.4 on the next page shows the metaclass hierarchy that follows from the recursive definition, using points and strings as examples.

Classes in Java have no first-class runtime representation. Instead, they are accessed through instances of `java.lang.Class`. For that reason, the classes themselves are not required to be objects. However, representing classes as objects enables uniformity in the resource management of the virtual machine. Both the Java Hotspot™ and the CLDC Hotspot™ Implementation virtual machines have a metaclass hierarchy similar to the one shown in figure 3.4 on the following page at the implementation level.

Objects tend to be small. In [DH99], it is reported that the average size of instance objects created by the SPECjvm98 benchmarks is 16–23 bytes.

Figure 3.4 Metaclass hierarchy for points and strings in Smalltalk-80



In our experience, many real-world applications exhibit similar allocation behavior. For that reason, it is important to minimize the size of the object headers. An extra word in the header of each object can easily increase the total memory requirements by 20%. For Java, it has been shown that header compression techniques based on heuristics can give space reductions of 7–21%; see [BFG02].

Most object-oriented languages impose special requirements on the object model. In the rest of this section, we will focus on four such requirements: inheritance, sizing, hashing, and synchronization. These requirements are common to many languages, and a study of the object models satisfying them provides insight into state-of-the-art virtual machine implementations.

3.1.1.1 Inheritance

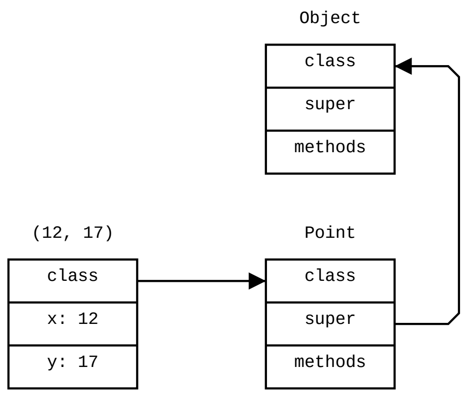
Inheritance is a mechanism for specializing state and behavior. It exists in almost all object-oriented languages. In this section, we will consider inheritance as a relationship between classes only, even though the term inheritance can be used to describe relations between objects; see [CUCH91].

Inheritance is used to define a *subclass* as a specialization of a *superclass*. The behavior of an instance of the subclass is an extension of the behavior of an instance of the superclass. Some languages support *multiple inheritance* by allowing classes to have more than one superclass. Since both Smalltalk and Java only have single inheritance, we will focus on object models that support this simpler kind of inheritance.

Figure 3.5 on the next page shows an example of single inheritance. The point class has been extended with a `super` field, which refers to the general object class. In effect, all instances of the point class, including

`(12,17)`, will behave as objects. As an example of the consequences of this, we turn to the draft ANSI Smalltalk Standard [Com97]. In Smalltalk implementations conforming to it, the object class must define a method for testing object identity. The effect of the inheritance hierarchy is that all points can be used in identity test expressions.

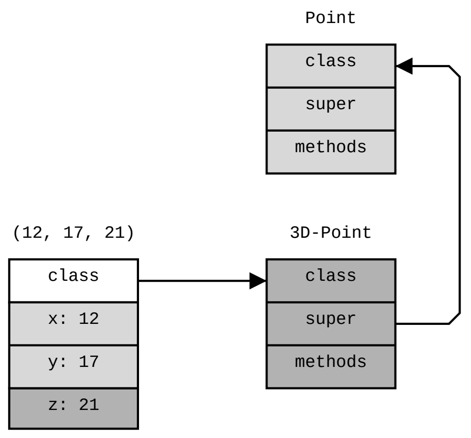
Figure 3.5 Single inheritance for points



In the object-orientation community, invoking a method on an object is often referred to as *dispatching*. Because instances of different classes may respond differently to the same request, the method to invoke at a given call site cannot be known statically; it must be looked up dynamically. The combination of dynamically looking up a method and dispatching to it, is known as *dynamic dispatching*. More details on efficient implementations of dynamic dispatching are described in section 3.2.

The point class may define methods for accessing the `x` and `y` coordinates. When defining new subclasses of the point class, it is vital that every instance of a subclass have both `x` and `y` fields. Otherwise, the inherited methods cannot access these fields, and execution will fail as a consequence. Therefore, most object-oriented languages enforce simultaneous state and behavior inheritance, in the sense that allocating instances of a subclass automatically allocates the fields needed by superclasses. Consider a subclass of the point class that is extended with a third coordinate: `z`. As shown in figure 3.6 on the following page, the `(12,17,21)` instance of the new three-dimensional point class has three coordinates, two of which are inherited from the point class.

Figure 3.6 Extending state by inheritance

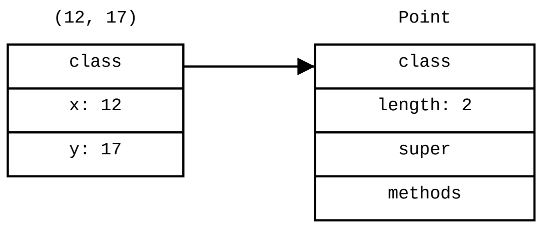


3.1.1.2 Sizing

The virtual machine needs to know the size of objects for purposes such as allocation and garbage collection. When creating objects, the correct amount of memory must be allocated, and when collecting garbage, objects must be traversed and possibly moved. This cannot be done without knowing the size of the objects involved.

The size of an object depends on the number of fields it contains. The number of fields of the objects treated so far is entirely defined by their classes. Such objects are referred to as *statically sized* objects. The obvious solution to determining the number of fields of such an object is to have an explicit `length` field in its class. The size of a statically sized object is easily deduced by multiplying the length by the size of each field and adding the size of the header. Figure 3.7 shows this solution.

Figure 3.7 Length field in the point class



Classes must have encoded lengths that are at least as large as the lengths encoded in their superclasses. This is due to the inheritance of state. In the example in figure 3.6, the point class has a length of two, whereas the class of three-dimensional points has a length of three. The length in a subclass is only identical to the length in its superclass if the subclass does not extend the state of instances.

3.1.1.3 Hashing

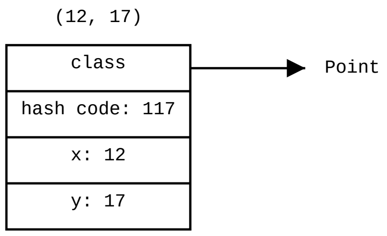
Hash codes are immutable integer values associated with objects. They are typically used to implement efficient lookup in hash tables. Both Smalltalk and Java require the virtual machine to provide default hash codes for all objects, but some classes of objects have specialized hash code implementations. The reason for this is that hash codes are closely coupled with equality: If two objects are equal, they are required to have the same hash code. For objects, such as strings, that have specialized equality tests, this mandates specialized hash code implementations.

The hash code implementation in the virtual machine is seldom used. The vast majority of allocated objects are not used as keys in hash tables, and the objects that are used as keys often have specialized hash code implementations. In [Age99], the results of some measurements of the frequency of hash code assignments in Java programs are reported. The *highest* reported percentage of all allocated objects with hash codes assigned by the virtual machine is only 0.51%.

When using compacting garbage collectors, the address of an object may change during execution. This makes it impossible to use the address of an object as its hash code. In systems with indirect pointers, it is possible to use the object table index of an object as its hash code. The most straight-forward hash code implementation with direct pointers is depicted in figure 3.8 on the following page. The idea is to allocate a field in all objects for storing the hash code, and assign a random number to the field at allocation time. The disadvantage to this approach is that every object allocation requires an extra field to be allocated and initialized. The performance of object allocation is compromised, and the pressure on the garbage collector is increased.

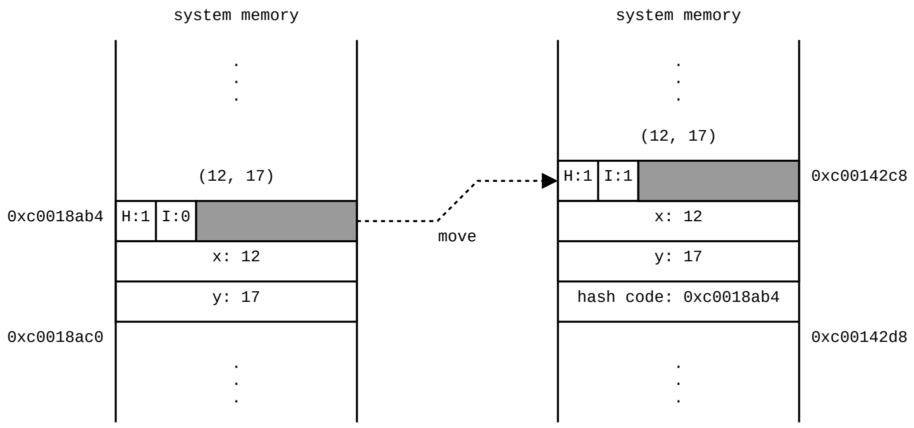
The Hotspot virtual machine uses this approach, except that the field is not used exclusively for the hash code. The other uses of the field include synchronization support, aging information, and forwarding pointers during garbage collection. Furthermore, the hash code is not computed and assigned until it is accessed. This allows fast object allocations.

Figure 3.8 Straight-forward hash code implementation



To avoid the memory overhead inherent in the straight-forward implementation, more sophisticated hashing techniques have been invented. In [Age99], a technique based on lazy object extending is described. The technique is not explicitly named by the author, but we will refer to it as *on-demand internalizing*. Using this technique, all objects will use their memory address as hash code, until they are moved due to garbage collection. When moving an object, the object is *internalized* by appending an extra field containing the old address of the object to it. All internalized objects use the contents of this extra field as hash code. Figure 3.9 shows the internalizing process for a point. The two bits shown in the object header are used to keep track of whether the object has had its hash code accessed (`H`), and whether the object has been internalized (`I`).

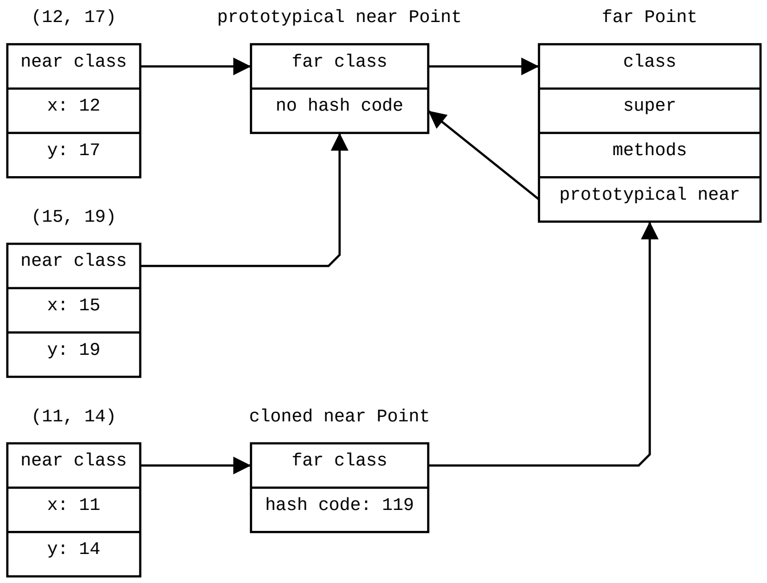
Figure 3.9 On-demand internalizing of a point



An important observation is that the internalizing step involved in the move can be avoided if the hash code of an object has never been accessed. If an object is moved before getting its hash code accessed, the object is not internalized. In this way, the hash code of an object is always the address of the object at the time when the hash code was first accessed.

Another technique for reducing the memory cost of hashing is implemented in a recent Java virtual machine implementation for embedded devices. CLDC Hotspot introduces a concept known as *near classes*; see [BG02]. Near classes represent state shared by most, but not necessarily all, instances of a class. To avoid confusion, the behavior-carrying class has been renamed to *far class*. Using this terminology, the near classes are inserted between the instances and their far classes. The situation is shown in figure 3.10.

Figure 3.10 Near classes in CLDC Hotspot



When an instance of a class is allocated, the near class field in the instance is initialized with a reference to the prototypical near class of the far class. When the need to assign a hash code to an instance arises, the prototypical near class is cloned, the near class field in the instance is updated to point to the cloned near class, and a random number is put in the hash code field of the clone.

The CLDC Hotspot hashing implementation has several advantages over on-demand internalizing. First, and perhaps most important, the internalizing process requires two bits of information in each object header. Even an object that never has a hash code assigned must have an object header with two spare bits. In an object model with one-word object headers, the class pointer must be encoded in the same word as the hashing bits. This means that the class of an object cannot be determined without bit-masking. Second, the object layout changes when an object is internalized. This means that the size computation routines must be extended to handle this case. The result is increased code complexity and loss of size computation performance. The disadvantage of the CLDC Hotspot approach is that an extra indirection is needed to access the far class. The following table summarizes the problems associated with the different hashing schemes: Straight-forward hashing (SFH), on-demand internalizing (ODI), and CLDC Hotspot hashing (CHH).

	SFH	**ODI**	**CHH**
Object layout changes due to hashing		•	
Extra space is required in non-hashed objects	•		
Class access requires bit-masking		•	
Class access requires extra indirection			•

In the design of our virtual machine, we have decided *not* to provide any hash code implementation. The scarce usage of the default hash code implementation does not justify the complexity it adds to the virtual machine. We have implemented separate `hash` methods in Smalltalk for the objects normally used as dictionary keys. These objects include strings, symbols, and integers. Furthermore, the straight-forward hashing scheme can easily be implemented for selected classes. Figure 3.11 shows an example of a class that maintains a hash variable for its instances.

Figure 3.11 Implementing a class for hashable objects

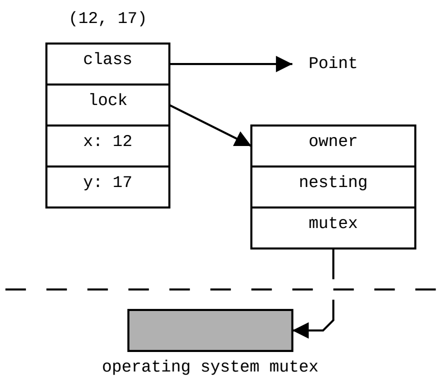
```
Hashable = Object (
  | hash |

  hash = (
    ^hash isNil ifTrue: [ hash := Random next ] ifFalse: [ hash ]
  )
)
```

3.1.1.4 Synchronization

Synchronization is used to gain exclusive access to resources in multi-threaded environments. The Java programming language allows methods and statements to be synchronized. Synchronized methods always try to obtain exclusive access to the object they are invoked on, whereas the synchronization target for synchronized statements can be arbitrary objects. This distinguishes Java from most other languages that only allow synchronization on dedicated objects. To avoid memory overhead and performance degradation most object models for Java employ sophisticated synchronization techniques.

The straight-forward solution for handling synchronization is to equip every object with a lock of its own. Locks are typically implemented on top of mutexes provided by the operating system. To keep track of the object-to-lock mapping, an extra field is needed in every object. This is shown in figure 3.12. It is possible to allocate the lock lazily. This is beneficial since most objects are never used as synchronization targets. The measurements done in [ADG+99] indicate that a lazy approach avoids allocating the lock and creating the operating system mutex for more than 80% of all objects.

Figure 3.12 Straight-forward synchronization implementation



The straight-forward implementation can be further improved by noticing that when the synchronization is uncontended, it never causes threads to block. Consequently, there is no need to allocate an operating system mutex for the lock. This observation allows the virtual machine implementation to avoid expensive calls to the operating system in the common case.

An improvement to the straight-forward technique, known as *thin locks*, is described in [BKMS98]. The idea is to avoid allocating locks for objects that have never been used in contended synchronization. As long as the object is used exclusively for uncontended synchronization, the owner thread and the nesting level are encoded directly into the lock field in the object. This encoding is known as a thin lock, and it is shown in figure 3.13. In case of contention, a lock and an operating system mutex are allocated, and the thin lock is inflated. To distinguish a thin lock from an inflated lock, the most significant bit is set during lock inflation.

Figure 3.13 Lock field encoding for thin locks

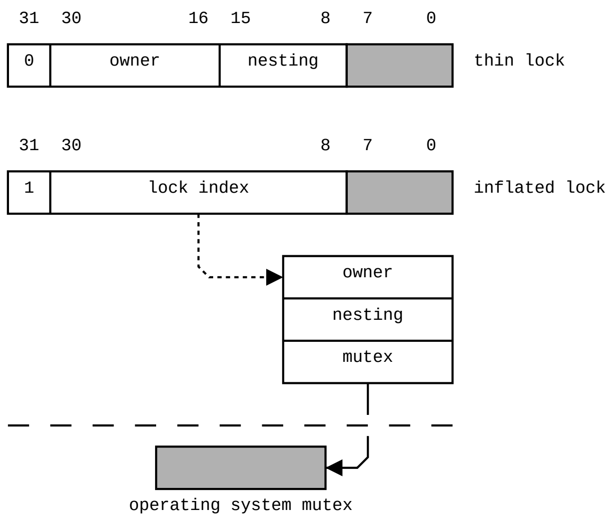
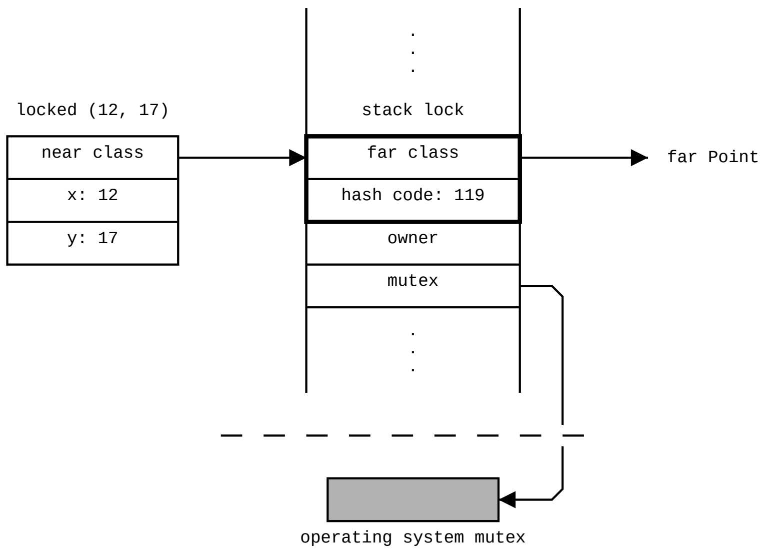


Figure 3.13 also shows that only 24 bits are used in the thin lock encoding of the lock field. The reason for this is that the original thin lock implementation was done on top of an existing Java virtual machine that uses the least significant 8 bits for other purposes.

The thin lock technique uses 24 bits in the header of every object. Many objects are never used as synchronization targets and thus the bits never contain anything but zeroes. CLDC Hotspot avoids wasting precious memory and preserves the one-word object header previously described. The technique used for achieving this is an extension of the synchronization mechanism implemented in Hotspot. The Hotspot implementation relies on the fact that synchronization in Java is block-structured. Objects are always unlocked in reverse locking order. This means that locks can be allocated on the stack. As it was the case for hashing, CLDC Hotspot uses near classes to avoid wasting memory for non-locked objects.

In CLDC Hotspot, the lock is allocated on the stack and it consists of a near class, an owner thread, and an operating system mutex, which is only allocated in case of contention. To avoid confusion, the stack-allocated lock is referred to as a *stack lock*. When locking an object, a stack lock is allocated and the near class of the object is copied to the stack lock. To indicate that the object has been locked, the near class pointer of the object is redirected through the stack. An object is locked if and only if its near class is on the stack. Figure 3.14 shows a locked point object, and illustrates that access to the far class and to the hash code is independent of synchronization. Both far class and hash code in the near class are accessed in exactly the same way, regardless of whether the object is locked or not.

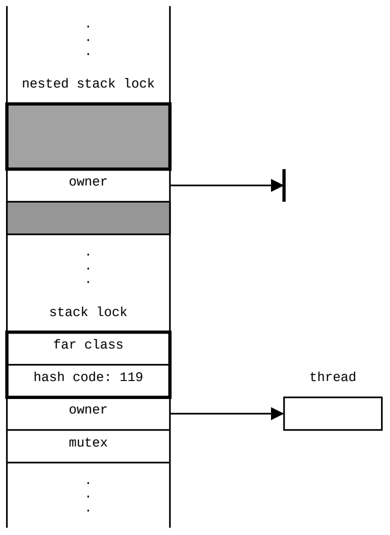
Figure 3.14 Synchronization in CLDC Hotspot



Since Java allows the same thread to lock the same object more than once, both the straight-forward synchronization technique and thin locks explicitly keep track of the current lock nesting depth. It is possible to use the stack for implicitly keeping track of lock nesting by allocating the locks on the stack. In this case, the lock nesting depth is equal to the number of stack locks on the stack. Figure 3.15 on the following page shows how a reentrant stack lock is allocated in CLDC Hotspot. Notice how an owner of zero is used to indicate that the lock is reentrant, and how the near class and the mutex in the stack lock are not initialized. The following table summarizes the problems associated with the different synchronization schemes: Straight-forward synchronization (SFS), thin locks (TL), and CLDC Hotspot synchronization (CHS).

	SFS	TL	CHS
Extra space is required in non-locked objects	●	●	
Nesting depth must be maintained explicitly	●	●	
Locks are required for non-contended locking	●		●
Class access requires extra indirection			●

Figure 3.15 Nested stack locks in CLDC Hotspot

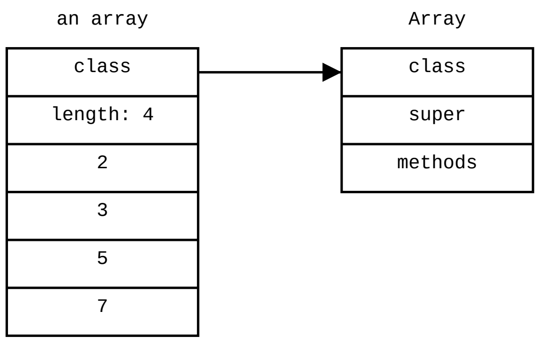


In the design of our virtual machine for Smalltalk, we have decided *not* to support using common objects as locks. Instead, we have provided virtual machine support for implementing synchronization in the high-level language. This support comes in the form of an atomic test-and-store instruction. Section 5.4 will reveal more details on this instruction, and show how semaphores, mutexes, and monitors can be implemented on top of it.

3.1.2 Arrays, Strings, and Symbols

Most programming languages support creating objects where the number of fields is specified as a part of the allocation. Such objects are commonly referred to as *arrays*. Arrays are often used when the number of fields needed for a specific task is only known at runtime. Furthermore, arrays are indexable; the *elements* of an array can be accessed using integer indexes. In safe systems, the virtual machine must verify that the indexes used are within range. To accommodate this, the number of elements is encoded in a `length` field in the header of each array. Figure 3.16 illustrates how an array containing the first four primes is represented.

Figure 3.16 Length and element fields in an array of primes



There can be different types of arrays, depending on which type of elements they contain. Common choices of element type include objects, characters, and bytes. Arrays of characters can be used to represent *strings*. Unlike other arrays, strings are usually immutable. This means that once a string is created its contents cannot be altered. Consequently, the virtual machine can use a single object for all strings with the same contents.

Common uses of strings include method selectors and names of classes. In Smalltalk, strings used as selectors are tested for equality extensively during method lookup. To speed this up, most virtual machines canonicalize all such strings. The canonicalized strings are known as *symbols*. By using a symbol table the virtual machine guarantees that all references to symbols with identical contents refer to the same symbol object. Comparing for equality is thus reduced to a single pointer comparison. The hard work of scanning strings to determine equality is done once when creating the symbol rather than every time it is used.

In our system, strings hold Unicode characters encoded in 16 bits. We have chosen to limit selectors and class names to 8-bit ASCII characters. Therefore, the symbol contents take up less space than string contents. Since the contents of symbols is seldom accessed, it is possible to compress the contents using more elaborate techniques. For debugging purposes, any such technique used on symbols must be non-destructive, so the original symbol contents can be restored.

We use about the same amount of space for strings and symbols as a similar Smalltalk system, but 47–77% less than a Java implementation. Details are provided in section 6.1.3.

3.1.2.1 Sizing Revisited

With the introduction of arrays, there are two different kinds of objects: Statically sized objects and *dynamically sized* objects. Instances of the array classes are all dynamically sized. If the classes for dynamically sized objects contained the `length` field, there would have to be a separate class for each possible length. Therefore, the `length` field for such objects is placed in the instances; see figure 3.16 on the preceding page. Just as for statically sized object, the total size of an object is determined by multiplying the contents of the `length` field by the field size and adding the size of the header.

The object size computation routine must know the location of the `length` field. For that reason, the virtual machine must be able to identify dynamically sized objects. CLDC Hotspot has a straight-forward solution to this problem. Each class has a `layout` field that describes its instances, by indicating if the instance size is static or dynamic. To save space in the classes, the length of statically sized objects is also encoded in this field. Figure 3.17 shows the `layout` field in the point class. By reading it, the virtual machine can tell that instances of the point class are statically sized instances with two fields.

Figure 3.17 Layout field in the point class

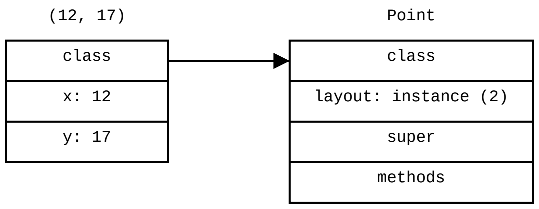
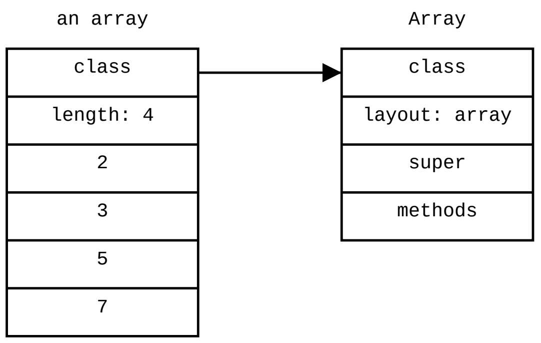


Figure 3.16 does not show the `layout` field in the array class. Figure 3.18 on the facing page remedies this, and it shows how the array of primes has an array layout. When determining the size of the array of

primes, the virtual machine will fetch the `length` field from the array instance; not from the class.

Figure 3.18 Layout field in the array class



Instead of manually encoding layouts in class fields, it is sometimes possible to let the virtual machine implementation language handle the diversity in layouts. Most object-oriented implementation languages support defining virtual behavior for instances of specific classes. Figure 3.19 shows how the virtual member functions of C++ can be used to determine the length of statically and dynamically sized objects.

Figure 3.19 Determining the length of an object using virtual behavior

```
class Object {
 public:
  virtual int length() { return class()->field(Class::length_index); }
  ...
};

class Array: public Object {
 public:
  virtual int length() { return field(Array::length_index); }
  ...
};
```

In C++, virtual member functions are implemented using *dispatch tables*. Every object with virtual behavior has a pointer to a dispatch table. The table holds the code addresses of the virtual member functions. In the example shown in figure 3.19, all instances of `Object` have a pointer to a table containing the address of the `Object::length()` function. The dispatch tables of all `Array` instances contain the code address of the

`Array::length()` function. Calls to virtual member functions are performed indirectly through the dispatch table of the object for which the function is invoked.

The Hotspot and SELF virtual machines rely on C++ for dealing with different object layouts. Since it takes up too much memory to have a dispatch table pointer in every object, they both move the virtual behavior to the classes. This way only classes need an extra pointer. Figure 3.20 outlines the implementation. Classes are capable of determining the size of their instances. The size of a given object is computed by passing `this` to the virtual `length_of_instance()` member function of its class. In addition to the classes shown, at least two classes derived from `Class` must be defined: One for arrays and one for the other objects. These two derived classes must implement the virtual `length_of_instance()` function in a way similar to the two `length()` functions shown in figure 3.19.

Figure 3.20 Moving the virtual dispatch table pointer to the class

```
class Object {
 public:
  int length() { return class()->length_of_instance(this); }
  ...
};

class Class: public Object {
 public:
  virtual int length_of_instance(Object* instance) = 0;
  ...
};
```

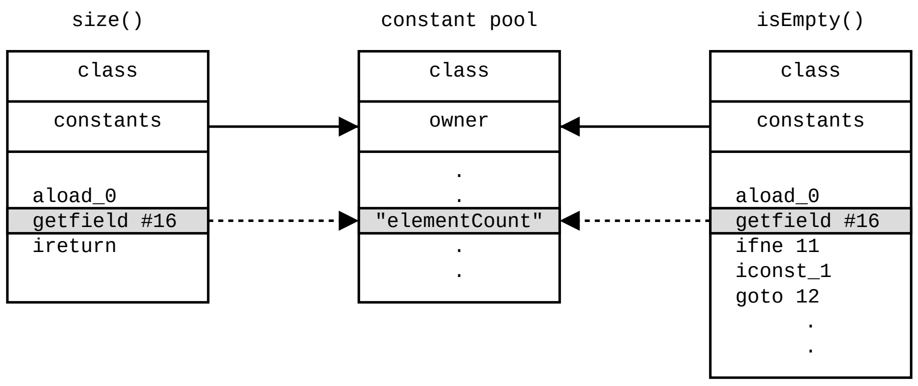
We have chosen to mimic the CLDC Hotspot solution. This has the advantage that we use only one word in each class for describing the instance layout. Using virtual member functions requires two words in each class with statically sized instances: One for the dispatch table pointer and one for the length. Furthermore, porting our virtual machine to embedded platforms that do not have a C++ compiler is less time-consuming. The details of our layout encoding is described in section 3.1.6.

3.1.3 Methods

Methods describe behavior and they are an essential part of the execution model. In this section, we will focus on how methods are represented in memory, and leave the details on *instructions*, the actual behavior description, for section 3.2 on page 48.

In Java, the source code constants are not put directly inside the method that refers them. Instead, they reside in objects known as *constant pools*, and accessing them is done using pool indexes. The situation is depicted in figure 3.21. In an attempt to save space, constant pools are shared between all methods of a given class. Unfortunately, this makes adding, removing, or changing methods dynamically difficult, because the garbage collector must find unused constant pool entries and free them. Once the entries have been freed, the constant pools must be compacted, and that requires a rewrite of all methods that use the constant pool.

Figure 3.21 Constant pool for methods in `java.util.Vector`

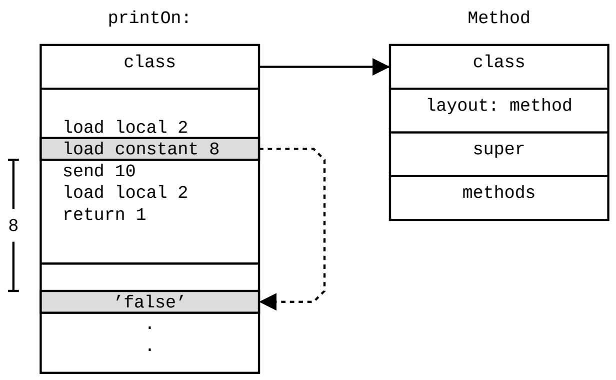


There are several ways to solve the problems related to dynamic code updates inherent in the Java model. In the SELF system, the problems were solved by *not* sharing the constant pools, thus enforcing a one-to-one relationship between methods and constant pools. Notice that sharing of constants is still possible, but only within single methods.

To speed up constant access, some systems avoid pointer indirections by placing constants at the first properly aligned addresses after the instructions that access them. Even though this scheme is fast, it also makes it impossible to share constants within methods, and it complicates pointer traversal of methods during garbage collection.

We have implemented a hybrid solution, where all constants are placed in the method, just after the *last* instruction. This is shown in figure 3.22 on the next page. Our scheme makes method-local sharing of constants possible and pointer traversal trivial. Furthermore, access to constants is fast, since it can be done relative to the current instruction pointer; see the `load constant` instruction in section 3.2.3 for more details.

Figure 3.22 Method for printing `false` on a stream



For uniformity reasons it may be beneficial to equip methods with a class reference, as shown in figure 3.22. This enables methods to be allocated in the object heap, which means that they will be subject to garbage collection when they are no longer referenced.

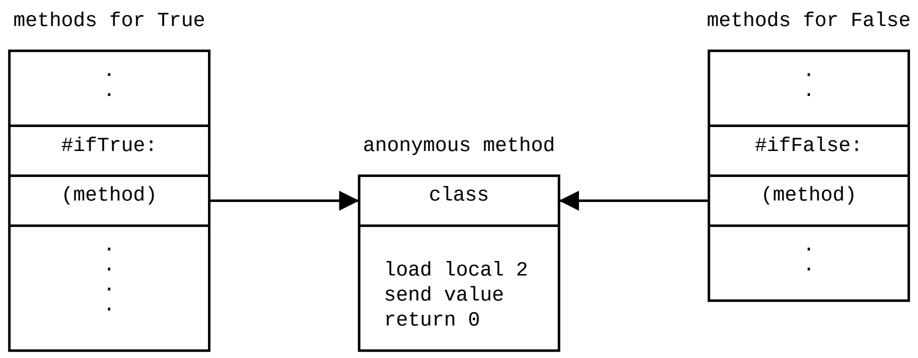
To save space, we have implemented sharing of identical methods. Methods with identical instructions and constants may have different selectors. To enable sharing of such methods, the selector is stored along with the method reference in the set of methods of the class. The method itself does not know which classes use it or which selectors it has. Figure 3.23 on the next page shows how the `True` and `False` classes share an implementation of `ifTrue:` and `ifFalse:` respectively. The sets of methods are implemented as arrays of (selector, method) pairs. Method sharing saves 3–11% of the space used for methods. In total, we use 57–63% less space for methods than equivalent code on a Java virtual machine. See section 6.1.4 for further information.

3.1.4 Integers

In this section, we will discuss the role of integers in pure object-oriented systems. In such systems, everything, including integers, are objects. This is the case in Smalltalk. Even though an integer is just an object, integers deserve special treatment due to their frequent use.

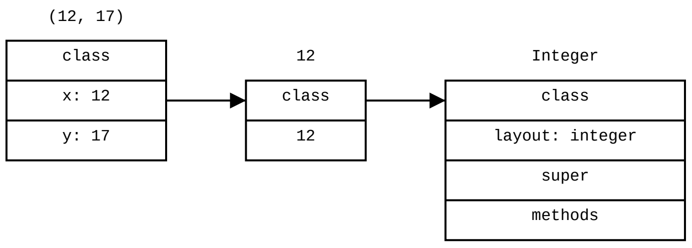
When evaluating arithmetic expressions like `a + b`, where `a` and `b` contain integers, the contents of `a` and `b` are expected to be left unchanged. As a consequence, the result of evaluating the expression must be a new integer object. However, allocating a new object for every arithmetic oper-

Figure 3.23 Method sharing for `ifTrue:` and `ifFalse:`



ation performed on an integer would put unnecessary strain on the resource management system and result in poor arithmetic performance. Since arithmetic operations on integers are widely used in software, some effort should be invested in making these fast and memory-efficient. Figure 3.24 shows an instance of the point class, which contains references to two integers.

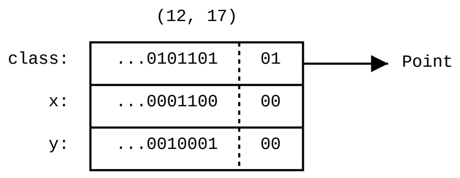
Figure 3.24 Point with explicit integer object coordinates



Most integers used in software can be represented in less than 32 bits. On modern systems, this means that small integers can be encoded in pointers. Unlike other pointers, a pointer with an encoded integer does not refer to any object in memory. Therefore, the virtual machine has to handle these small integers differently than other pointers. Access to the class of a small integer object cannot be done by dereferencing the pointer. The virtual machine must have some way to identify which pointers contain small integers and which do not.

In [Gud93], various techniques for *tagging* pointers are discussed. Pointer tagging is a way of associating different pointer types with recognizable bit patterns in the pointers. We have chosen to use the two least significant bits of all pointers for tagging purposes. This makes sense because objects allocated in the heap in our system are aligned on 32-bit boundaries. Thus, the two least significant bits of pointers to such objects are always zeroes. As long as the virtual machine always ignores these bits when dealing with pointers, they can be used to hold the pointer tag. To make it possible to add small integers without masking off the tag bits, we have chosen to let small integers use `00` as tag. Pointers to objects allocated in the heap use `01`. The two remaining tags are reserved for other optimizations. Tagging does not slow down access to fields in objects, since most native load and store instructions are capable of adding an immediate offset to the base pointer before dereferencing it. By subtracting one from this immediate offset the tag is effectively ignored. Figure 3.25 shows the point from figure 3.24 in the presence of pointer tagging. The class pointer is tagged, but it still refers to the class object. The coordinates are encoded in the coordinate pointers, and do not rely on explicit integer objects anymore. With such pointer tagging, it is possible to handle arithmetic on 30-bit integers without any object allocations.

Figure 3.25 Point with tagged integer coordinates



It should be noted that pointer tagging also works with indirect pointers. The original Smalltalk-80 implementation is based on 16-bit indirect pointers. Only one bit in each pointer is used for tagging. Indirect pointers with the least significant bit set to `0` are used to index objects in the heap through an object table entry, whereas 15-bit small integers are represented with tag `1`.

Explicit tagging makes it easy to identify pointers just by looking at their bit patterns. Some systems that do not use tagging also try to find the set of all pointers by looking at bit patterns. Systems that employ *conservative* garbage collectors approximate the set of live objects by treating any pointer-like bit pattern as actual pointers while traversing the object graph. The resulting approximation consists of all live objects along with objects that have been reached through integers that look like pointers. Transitively, such integers may keep an arbitrary number of objects artificially alive.

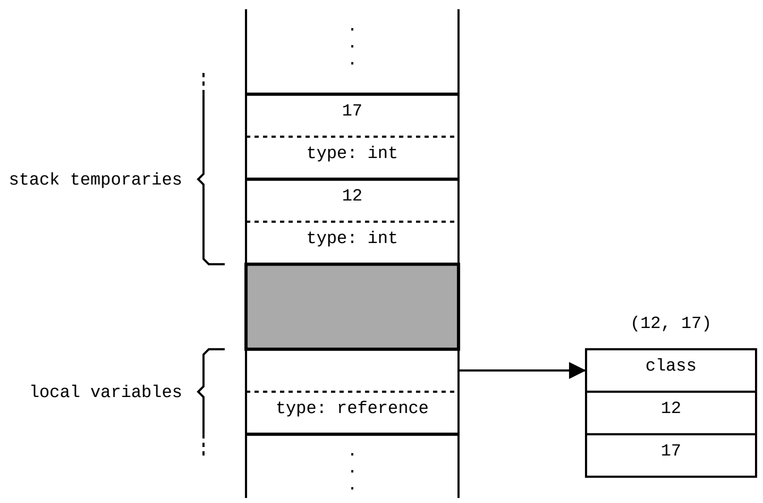
To avoid memory leaks, most garbage collectors for modern object-oriented languages are *precise*; if there exists no references to an object, it will eventually be deallocated. With precise garbage collection, even non-pure object-oriented systems are forced to keep track of the locations of all pointers. If the system is statically typed, it is possible to use *pointer maps* to convey this information. A pointer map is a data structure that tells the virtual machine which parts of objects and activation records contain pointers. Pointer maps are generated from the static type annotations of variables in the source code. Pointer maps for activation records are commonly referred to as *stack maps*.

Some systems do not have explicit type annotations for all parts of activation records. In Java, the local variables and stack temporaries of an activation record may have changing types during the execution of a method. However, as described in [Gos95], it is possible to calculate the types of the stack elements at each bytecode in the method, using straight-forward data flow analysis. The type calculation problems associated with the `jsr` and `ret` bytecodes are solvable; see [ADM98].

The type calculations can be driven by demand. This is used by the Hotspot virtual machine to avoid keeping unneeded stack maps around. Whenever the garbage collector needs a stack map for a bytecode in a method, the stack map is calculated using data flow analysis based on abstract interpretation. To avoid repeated calculations the stack maps are cached. Even with this kind of optimization the stack maps take up much memory. In the industry standard Java virtual machine for embedded devices, KVM, the stack maps are precalculated and up to three times as large as the maximum amount of used stack space; see section 6.1.5.

To save memory, it is possible to exploit the relatively low number of activation records by reverting to tagging. Since integers in Java must be 32 bits, the tagging cannot be encoded within the same word as the integer. CLDC Hotspot solves this by associating an extra type tag word with every stack temporary and local variable. This is shown in figure 3.26 on the following page. The memory used for keeping track of pointers is thus dependent on the number of activation records; not on the number of stack maps. When combined with techniques such as *lazy type tagging* [BL02], the performance impact of maintaining the type tags is negligible.

Figure 3.26 Type tagging of stack temporaries and local variables



3.1.5 Garbage Collection

In this section we will discuss low-level design issues for garbage collectors. This is *not* intended as an in-depth coverage of garbage collection in general. See [Wil92] and [GS93] for more thorough discussions of the topic.

There are two fundamentally different ways of detecting garbage: *Reference counting* and *tracing*. Reference counting collectors keep track of the number of references to an object. When the number reaches zero, the memory used by the object is reclaimed. Tracing collectors traverse the object pointer graph to determine liveness. If an object is pointed to by a live object, it is also live. Reference counting collectors have been successful in distributed and real-time systems; see [Rit99]. Unfortunately, they are generally not as efficient as tracing collectors, and they cannot easily reclaim cyclic data structures. For these reasons, we will focus entirely on tracing collectors.

At first glance, indirect pointers seem to make garbage collection less complex. After all, the direct references to an object only exist in the object table. However, when using indirect pointers with tracing collectors, the pointer graph must still be traversed to determine liveness. The only advantage of using indirect pointers is that moving an object takes time proportional to the size of the object. With direct pointers, the worst-case scenario is that the object is very popular, meaning that there are many references to it. It is unlikely, but not impossible, that every object field in the heap refers to the object being moved. Thus, the time complexity of moving an object in this case is $O(n, m) = n + m$, where n is the size of the object and m is the used size of the heap.

Tracing collectors traverse the object pointer graph to determine which objects are alive. The garbage collector uses pointer maps or tagging to identify pointers in objects and activation records. This is covered in section 3.1.4. The remaining challenge is finding any pointers that are neither in objects, nor in activation records.

The virtual machine holds references to a number of global objects. This includes references to the symbol table and globals such as `nil`, `true`, and `false`. The set of references to global objects is largely static, and therefore the virtual machine can have tailored code for traversing it. Pointers may also exist in local variables of stack activations described by the virtual machine implementation language. These pointers are more difficult to handle, since they change during execution. Figure 3.27 shows an example of such pointers. If the execution of `new_object()` triggers a garbage collection, the `name` and the `value` pointers must be found.

Figure 3.27 Allocating association objects in the virtual machine

```
Association* Universe::new_association(Symbol* name, Object* value) {
  Association* result = (Association*) new_object(3);
  result->set_class(association_class());
  result->set_name(name);
  result->set_value(value);
  return result;
}
```

For the purposes of this discussion, it does not matter whether the virtual machine uses separate stacks for running virtual machine instructions and native code written in the implementation language, or if it uses an approach where both are intermixed on the same stack [BGH02]. The problem of finding pointers in the parts of the stack described by the implementation language remains the same.

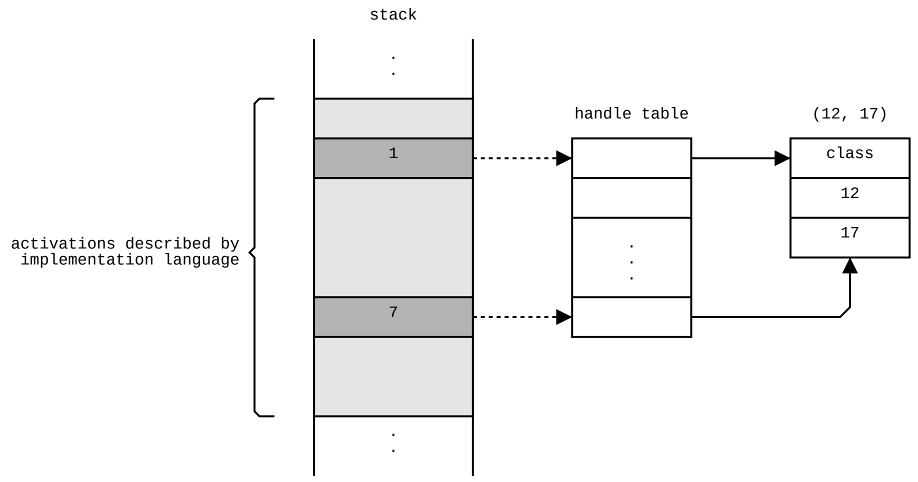
3.1.5.1 Handles

Unless the virtual machine implementation language has strict policies for placing pointers, it is impossible to find them just by looking at the stack. Direct pointers *are* the addresses of the objects they point to, and it is not possible to distinguish addresses from integers by looking at bit patterns.

In some cases, it is possible to have the compiler generate stack maps for activations described by the implementation language. Unfortunately, the type system of C++ allows using integers as pointers, and for that reason implementations based on that language cannot rely on stack maps. The most common solution is to use *handles* instead of direct pointers.

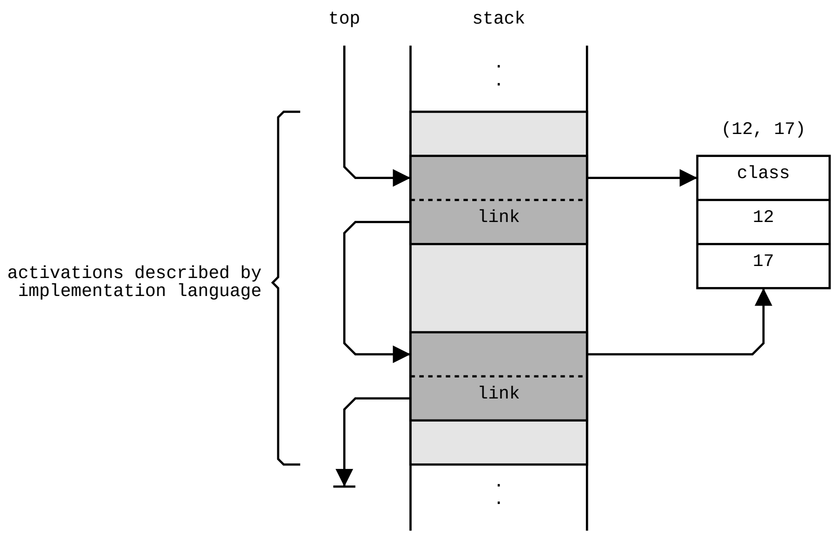
A handle is a data structure that has an associated object pointer, which the virtual machine knows how to find. Indirect pointers are one variant of handles. Figure 3.28 shows how integers can be used to index handle table entries. In return, the entries refer to objects in the heap. This way, the garbage collector only has to traverse the handle table. All objects referred to from the handle table are assumed to be live. It is up to the virtual machine implementation to free entries no longer needed, thereby enabling reclamation. Table-based handles are used in the Hotspot virtual machine.

Figure 3.28 Handles using handle table



Another variant chains individual handles together through the stack. This is shown in figure 3.29 on the next page. The handles contain the object pointer and a link to the previous handle on the stack. The virtual machine maintains a pointer to the topmost handle on each stack. The garbage collector can traverse the handles from top to bottom by following the handle links. CLDC Hotspot relies on such chained handles.

Figure 3.29 Handles chained through the stack



3.1.5.2 Ignorance

Rather than devising clever ways of locating pointers in activation records described by native code, it might be better to avoid having them around during garbage collections. In most systems, garbage collections happen as a result of allocations. Typically, there are many parts of the implementation that allocate objects. For that reason, it seems virtually impossible to guarantee that no pointers from within the implementation exist when collecting garbage.

We have implemented a straight-forward solution to the problem. The only situation where we switch from executing virtual machine instructions to executing native code written in C++ is for handling complex operations. We have arranged for all our native code to be restartable. If the garbage collector is invoked, the native code responsible for the allocation is restarted. This means that all activation records for the native code are removed. Any pointers in the activation records are thus ignored. Since we do not allow native code to invoke Smalltalk methods, all activation records described by native code are guaranteed to be removed. In effect, all troublesome pointers are ignored, and we have no handles in our implementation.

3.1.6 Design and Implementation

In this section we will conclude our discussion of object models by describing the design and implementation of the object model for the virtual machine we have built. Since our virtual machine is simple by design, the description also serves as an introduction to the inner structures of a modern virtual machine implementation. The reader is assumed to be familiar with the C++ language, which is our implementation language of choice.

Our implementation is based solely on direct object pointers. As described in section 3.1.4, it is convenient to tag these pointers to avoid allocating memory for small integers. For reasons of integer arithmetic efficiency, we have chosen the tag assignment shown in figure 3.30.

Figure 3.30 Pointer tag assignments

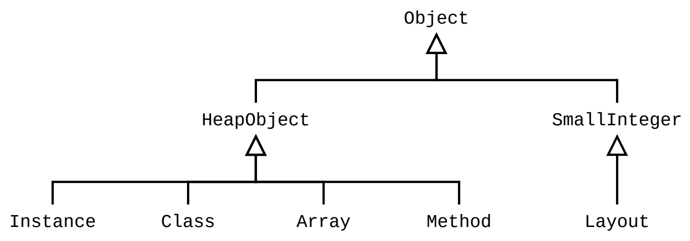
```
enum Tag {
  small_integer_tag = 0,
  heap_object_tag   = 1,
};
```

The cornerstone of our object model is the class `Object`, as shown in figure 3.31. The direct object pointers in our system are of `Object*` type, or any pointer type derived from that; see the implementation class hierarchy in figure 3.32 on the facing page. Given an object pointer, we can compute the tag by extracting the two least significant bits of it. This is the purpose of the `tag()` member function on objects. Using the tag assignments, it is now possible to answer if a given object is a small integer, or if it is allocated in the heap. Obviously, the implementation supports such queries by means of the member functions `is_small_integer()` and `is_heap_object()`.

Figure 3.31 Object implementation

```
class Object {
 public:
  bool is_small_integer() { return tag() == small_integer_tag; }
  bool is_heap_object()   { return tag() == heap_object_tag;   }
  ...

 protected:
  Tag tag() { return (Tag) ((int) this & 3); }
};
```

Figure 3.32 Class hierarchy for virtual machine implementation



Small integers are represented as the most significant 30 bits of object pointers with small integer tags. Given such an object pointer, the value can be extracted by shifting the pointer two positions to the right arithmetically. Figure 3.33 shows how small integers are derived from objects, and how `value()` can be implemented. Note that the arithmetic primitives do *not* extract the value of their arguments. Instead they work directly on the tagged pointers, thereby eliminating most arithmetic shifting.

Figure 3.33 Small integer implementation

```
class SmallInteger: public Object {
 public:
  int value() { return ((int) this >> 2); }
};
```

One of the fundamental properties of our object model is uniformity. As explained in section 3.1.1, it is convenient for all objects allocated in the heap to have a class reference. Since classes themselves are allocated in the heap, the references to them can be stored in an ordinary heap object field, without complicating garbage collection. Figure 3.34 on the following page shows the implementation class for heap objects, and illustrates how the address of a heap object can be computed by subtracting the heap object tag from the pointer. In return, the address can be used to access fields.

Figure 3.34 Heap object implementation

```
class HeapObject: public Object {
 public:
  Class*  class()   { return (Class*) field(class_index);      }
  Address address() { return (Address) this - heap_object_tag; }
  ...

 protected:
  Object* field(int index);

 protected:
  static const int class_index = 0;
};
```

Implementing the `field()` member function is straight-forward. As illustrated in figure 3.3 on page 17, heap objects are just collections of fields, which contain object pointers. The code is shown in figure 3.35. The function for setting the contents of a field is similar.

Figure 3.35 Accessing the fields of an object in the heap

```
Object* HeapObject::field(int index) {
  Object** field = (Object**) (address() + index * sizeof(Object*));
  return *field;
}
```

The simplest heap objects are instances. Instances are state-holding objects, such as points. They are allocated and used during program execution. Even though classes, methods, and arrays can be considered instances of their respective classes or metaclasses, the instances represented by the class `Instance` are only those that are neither classes, methods, nor arrays. The code is shown in figure 3.36 on the next page. Since fields in such instances correspond to Smalltalk object variables, we have implemented the `variable()` accessor. Notice how variables are indexed starting from one, due to the presence of the class field.

Classes are used to describe the state and behavior of instances. To achieve this, classes contain three fields: `layout`, `super`, and `methods`. The implementation, shown in figure 3.37 on the facing page, defines accessors for these fields. The implementation of the `Layout` and `Array` classes will be described later in this section.

The layout field in the class is essential for determining sizes and iterating over pointers in instances of a given class. The layout object itself

Figure 3.36 Instance implementation

```
class Instance: public HeapObject {
 public:
  Object* variable(int index) { return field(index); }

 private:
  static const int number_of_header_fields = 1 + class_index;
};
```

Figure 3.37 Class implementation

```
class Class: public HeapObject {
 public:
  Layout* layout()  { return (Layout*) field(layout_index);  }
  Class*  super()   { return (Class*)  field(super_index);   }
  Array*  methods() { return (Array*)  field(methods_index); }
  ...

 protected:
  static const int layout_index  = 1 + class_index;
  static const int super_index   = 1 + layout_index;
  static const int methods_index = 1 + super_index;

 private:
  static const int number_of_header_fields = 1 + methods_index;
};
```

is encoded as a small integer, where the two least significant bits of the small integer value define the type of the instances. If the layout is for statically sized instances, the remaining 28 bits define the instance length; see figure 3.38 on the next page.

Now that the layout description in the class is complete, we can start querying the heap objects for their implementation type. This can be done by adding the type test member functions shown in 3.39 on the following page to the heap object class. For instance, if we want to know if a given heap object is an array, we consult the layout in the class of the heap object.

Recall that the methods defined by a class are stored in an array. Arrays represent integer indexable state with a length that is specified at allocation time. The length is stored as a small integer in the array object itself. Figure 3.40 on the next page shows the implementation of arrays in our system.

Methods contain both instructions and constants. Like arrays, methods are dynamically sized. We have encoded both the number of instructions

Figure 3.38 Layout implementation

```
class Layout: public SmallInteger {
 public:
  enum Type {
    class_layout_type,  array_layout_type,
    method_layout_type, instance_layout_type
  };

  Type type()   { return (Type) (value()  & 3); }
  int  length() { return        (value() >> 2); }

  bool is_class_layout()    { return type() == class_layout_type;    }
  bool is_array_layout()    { return type() == array_layout_type;    }
  bool is_method_layout()   { return type() == method_layout_type;   }
  bool is_instance_layout() { return type() == instance_layout_type; }
};
```

Figure 3.39 Querying the implementation type of heap objects

```
class HeapObject: public Object {
  ...
  bool is_class()    { return class()->layout()->is_class_layout();    }
  bool is_array()    { return class()->layout()->is_array_layout();    }
  bool is_method()   { return class()->layout()->is_method_layout();   }
  bool is_instance() { return class()->layout()->is_instance_layout(); }
};
```

Figure 3.40 Array implementation

```
class Array: public HeapObject {
 public:
  SmallInteger* length() { return (SmallInteger*) field(length_index); }

 protected:
  static const int length_index = 1 + class_index;

 private:
  static const int number_of_header_fields = 1 + length_index;
};
```

and the number of constants in a single `length` field. Figure 3.41 on the facing page shows our implementation.

As noted in section 3.1.5, it is vital for the garbage collector to be able to compute the size of any object in the heap. Since the size of such an object depends on the layout in its class, we can define a pseudo-virtual

Figure 3.41 Method implementation

```
class Method: public HeapObject {
 public:
  SmallInteger* length() { return (SmallInteger*) field(length_index); }

  int number_of_instructions() { return length()->value() & 32767; }
  int number_of_constants()    { return length()->value() >>   15; }

 protected:
  static const int length_index = 1 + class_index;

 private:
  static const int number_of_header_fields = 1 + length_index;
};
```

`size()` member function as shown in figure 3.42. The `switch` is typically compiled to an indirect jump through a jump table, and as such it is equivalent, performance-wise, to a dispatch through a virtual dispatch table.

Figure 3.42 Pseudo-virtual heap object size implementation

```
int HeapObject::size() {
  switch (class()->layout()->type()) {
    case Layout::class_layout_type    : return (Class*)    this->size();
    case Layout::array_layout_type    : return (Array*)    this->size();
    case Layout::method_layout_type   : return (Method*)   this->size();
    case Layout::instance_layout_type : return (Instance*) this->size();
  }
}
```

The actual size computations are done in the implementation classes. Some of the code is shown in figure 3.43 on the following page. For the statically sized heap objects, the size is determined by their classes. Instances have their length encoded in the layout in their class. For dynamically sized objects, the size is determined by the objects themselves.

The object model we have described is both simple and compact. We use 50% less memory for classes, methods, and strings than a commercial Java implementation developed with low-end mobile devices in mind. Compared to a standard Smalltalk system, our compact object representation yields a 35–45% footprint reduction. Detailed measurements and the associated results are available in section 6.1.1.

Figure 3.43 Size computation functions for instances and arrays

```
int Instance::size() {
  return number_of_header_fields + class()->layout()->length();
}

int Array::size() {
  return number_of_header_fields + length()->value();
}
```

3.2 Execution Model

The execution model extends the object model with behavior. At the heart of most execution models is an *instruction set*. The instructions define the processing capabilities of the virtual machine, and they must therefore be sufficient for implementing the programming language supported by the virtual machine. This section describes how we have designed and implemented an execution model for our virtual machine. Our model is simple by design, but it is by no means incomplete. This way, this section helps illustrate how dynamic object-oriented systems can be equipped with an efficient, production-quality execution engine.

Before diving into the details of our instruction set, we will discuss the execution strategy and solve key design issues. The section concludes by covering and evaluating some of the optimizations we have implemented.

3.2.1 Strategy

The execution of virtual machine instructions must be carried out by hardware processors. Most instructions cannot be executed directly by the hardware, and therefore they must be either interpreted or compiled into equivalent native code. Usually there is an overhead involved in interpreting instructions, and therefore native code is often several times faster than interpreted code.

In the context of embedded systems, the fundamental problem with compilation is that the native code representation produced by the compiler takes up too much memory. In the context of Java, it is our experience that native code is four to five times larger than bytecode. In an attempt to minimize the size of the generated code, some virtual machines employ an *adaptive* runtime compiler. The idea is that only frequently executed code is compiled to native code. Many programs spend most of the time in a relatively small subset of the total code, and by only compiling this *working* *set* of code, the virtual machine can optimize performance without sacrificing too much memory. The virtual machine avoids having to compile all executed code by interpreting infrequently executed code. Choosing how much and what code to compile is a balancing act, where the virtual machine trades memory for performance. It is complicated by the fact that the working set of a program often changes over time. Like many larger Java virtual machines, CLDC Hotspot contains an adaptive compiler. It also has the ability to remove compiled code if available memory runs low or the working set changes [BAL02]. Unfortunately, an adaptive compiler often takes up tens of kilobytes of code just for the compiler itself, thus increasing the total size of the virtual machine as well as the overall memory footprint considerably.

To minimize memory footprint, we have chosen to implement a virtual machine based solely on interpretation. The virtual machine is stack-based, like virtual machines for Java, Smalltalk, and SELF. By implicitly using top-of-stack elements as operands, the size of programs for stack-based machines can be up to eight times smaller than the equivalent code for register-based machines [PJK89]. The performance of the interpreter is also improved, since implicit operands require no decoding. We want to push interpreter performance as far as possible by designing an instruction set optimized for speed. To further narrow the performance gap between interpreted and compiled code, our optimized interpreter is coupled with an efficient runtime system. The result is a 30 KB fast, interpreted, object-oriented virtual machine implementation for embedded devices. On average, it outperforms even the fastest interpreted Java virtual machines by 5–29%. See section 6.2.1 for detailed benchmarks.

3.2.2 Design Issues

Given an object model and an overall execution strategy, there are still several issues that must be resolved before an instruction set can be designed, let alone optimized. In this section, we will focus on three fundamental design issues, the solutions to which are the cornerstones of our virtual machine implementation.

3.2.2.1 Evaluation Order

Most high-level languages evaluate expressions from left to right. The first Smalltalk system to use strict left-to-right evaluation order was Smalltalk-80. Consider a Smalltalk expression, `Console show: 5 + 7`, that prints the result of adding 7 to 5 on the console. In a Smalltalk system with left-to-right evaluation order, the instructions for this expression would be as shown in figure 3.44.

Figure 3.44 Instructions with left-to-right evaluation

```
push Console
push 5
push 7
send +
send show:
```

The benefit of this approach is that the instructions are easy to understand and easy to generate. However, just before sending `show:`, the integer `12` is on top of `Console` on the execution stack. Smalltalk is an object-oriented language with single dispatch, and therefore the argument, `12`, does not affect the method lookup. To find the right method to invoke in the `show:` send, the virtual machine only has to consider the runtime type of `Console`. The C++ method in figure 3.45 shows the steps necessary to interpret a send instruction in a standard Smalltalk system.

Figure 3.45 Interpretation of a Smalltalk-80 send

```
void interpret_send(Symbol* selector) {
  Object* receiver = stack->at(selector->number_of_arguments());
  Method* method   = receiver->class()->lookup(selector);
  method->invoke();
}
```

The computation of the number of arguments for a given selector is a trivial, but time-consuming task. It consists of counting the number of colons in the selector, with special handling of binary selectors. The result of the computation can be cached in the selector, but this takes up extra memory for all selectors in the system. Another commonly used approach is to cache the number of arguments in the send instructions. By introducing a fixed number of new send instructions, customized by the number of arguments in the send selector, the selector-local number-of-arguments cache can be eliminated without sacrificing performance. The C++ method in figure 3.46 on the next page shows how to interpret the new, customized instructions.

In most systems, new send instructions are introduced only for the most common number of arguments, e.g. zero, one, two, and three, and

Figure 3.46 Interpretation of customized Smalltalk send

```
void interpret_send_<n>(Symbol* selector) {
  Object* receiver = stack->at(<n>);
  Method* method   = receiver->class()->lookup(selector);
  method->invoke();
}
```

the generic send instruction is kept in the system and used for the rest of the sends. Figure 3.47 shows the instruction sequence of the console printing example using the new set of instructions.

Figure 3.47 Customized instructions with left-to-right evaluation

```
push   Console
push   5
push   7
send_1 +
send_1 show:
```

In the system described so far, the only disadvantage of this approach is that four new send instructions are introduced. However, in our Smalltalk implementation, and in many other available systems, a send can be in different states. Consider the send states introduced by inline caching: monomorphic send and megamorphic send. As will be described in section 3.2.4.7 the most efficient way of storing the state of a send is directly in the instruction, but since the send state is orthogonal to the number of arguments in the send selector, we have to introduce instructions for all combinations of state and cached number of arguments. This includes instructions such as `monomorphic_send_1` and `megamorphic_send_2`.

Inline caching is not the only optimization that depends on send state. In our Smalltalk system, we currently have seven different send states and are considering adding more to support further optimizations. To eliminate the need for more than 35 different send instructions, we have chosen to generate instructions that ensure that the receiver is always at the top of the stack when doing a send. The net effect is that customizing the send instructions based on the number of arguments in the send selector is no longer necessary.

In Smalltalk-76 [Ing84], the arguments to a send are evaluated from left to right, but the receiver is always evaluated last. This ensures that the receiver is at the top of the stack, and consequently the send instructions do not need to know the number of arguments. In Smalltalk-80, the evaluation order was changed to a strict left-to-right evaluation order. The change was done since post-evaluation of receivers made the order of evaluation different from the order of appearance in the code. In our Smalltalk system, we have chosen strict right-to-left evaluation order. Figure 3.48 shows the instructions for the console printing example in a Smalltalk system with right-to-left evaluation order. The advantage of a right-to-left evaluation order is that send instructions do not have to know the number of arguments, and the evaluation order remains easy to understand.

Figure 3.48 Instructions with right-to-left evaluation

```
push 7
push 5
send +
push Console
send show:
```

The C++ method for interpreting sends is reduced to the one in figure 3.49 on the facing page. The only problem with our approach is that it changes the semantics of the Smalltalk language. When evaluating an expression in the presence of side-effects, the evaluation order becomes significant. In practice, however, this has not been a problem. In our experience, most Smalltalk programmers are unaware of the fact that the evaluation order is left-to-right and most Smalltalk code does not rely on any particular evaluation order. In our system, we managed to change the evaluation order from left-to-right to right-to-left without changing a single line of Smalltalk code. The only problem that might arise from reversing the evaluation order, is when evaluating the argument has side effects. In Smalltalk, much of the code that has side effects is enclosed in blocks. Because blocks are not evaluated until they are explicitly invoked, the order in which the blocks are passed to the method does not matter.

3.2.2.2 Efficient Blocks

One of the most convenient properties of Smalltalk is the language support for defining control structures. This support comes in the form of blocks. Blocks are expressions that can be evaluated on-demand with access to their scope. Section 2.1.2 describes the syntax and semantics of

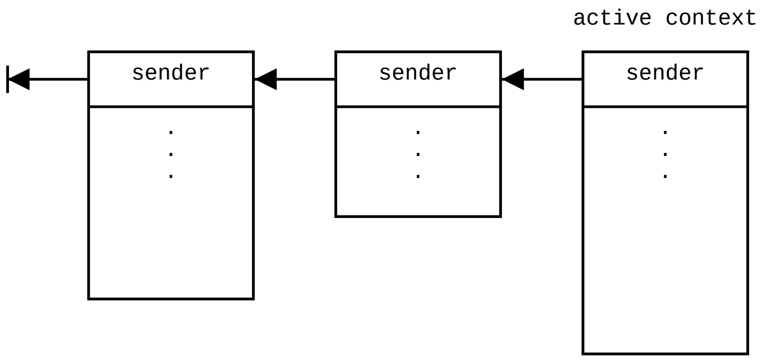
Figure 3.49 Interpretation of right-to-left sends

```
void interpret_send(Symbol* selector) {
  Object* receiver = stack->at(0);
  Method* method   = receiver->class()->lookup(selector);
  method->invoke();
}
```

blocks in our system. In this section, we will focus on how blocks can be implemented efficiently.

The Smalltalk-80 implementation described in [GR84] gives useful insights into how blocks can be implemented. During interpretation, it is necessary to keep track of the values of temporaries, such as the receiver and the arguments. This interpreter state is saved in objects known as *contexts*, which are similar to activation frames in procedural programming languages. Each time a message is sent to an object, a new context is allocated. The sender of the message is registered in the newly allocated context, and in this way the contexts are chained together. Each context belongs to a method, and the context that belongs to the currently executing method is called the *active context*. Figure 3.50 shows such a chain of contexts.

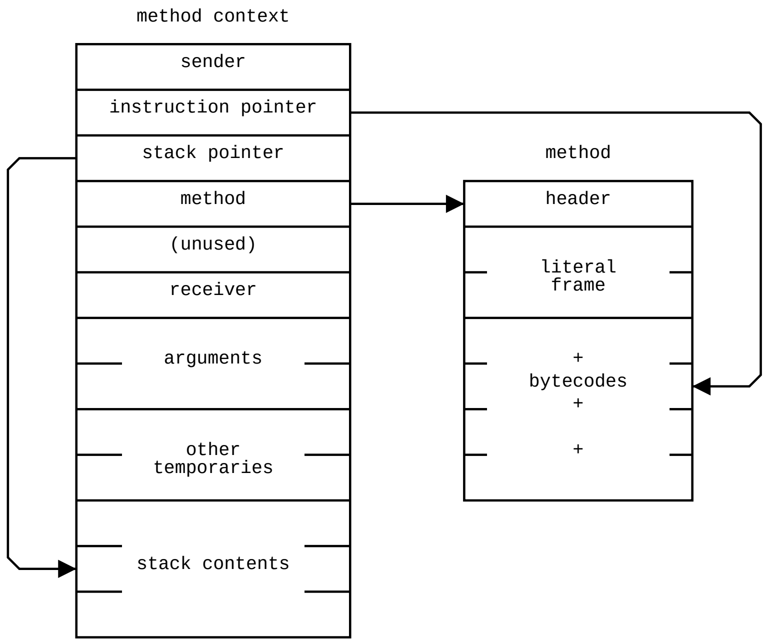
Figure 3.50 Contexts chained through sender field



The active context is used extensively by the interpreter. Figure 3.51 on the following page shows the contents of the context in details. The figure itself is a replica of figure 27.5 in [GR84], included here for reasons of clarity. The reason why figure 3.51 refers to the contexts as *method contexts* is that Smalltalk-80 supports another type of context known as *block contexts*.

As the name indicates, block contexts are used when evaluating blocks. They are allocated whenever a block expression is evaluated.

Figure 3.51 Method contexts in Smalltalk-80



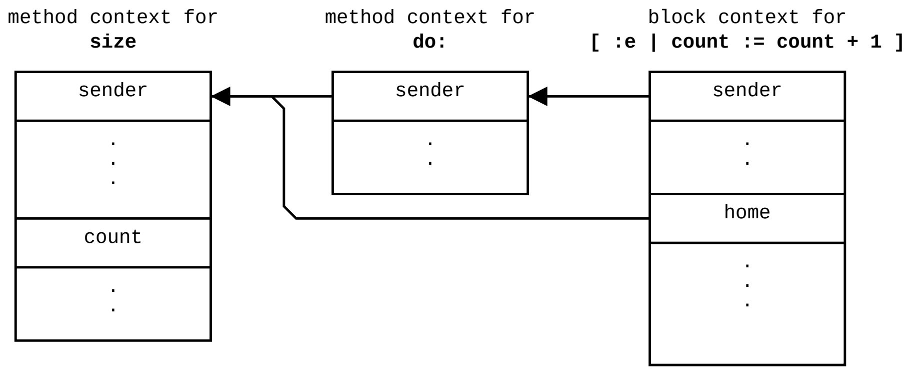
Block contexts are similar to method contexts, except that the method, receiver, arguments, and temporaries have been replaced with a reference to the method context in which the block context was allocated. This reference is known as the *home* of the block context. The purpose of the home reference is illustrated in figure 3.52 on the next page, which shows a method from the collection hierarchy. When executing `size`, the temporary variable `count` is contained in the active method context. When the block context for `[ :e | count := ... ]` is evaluated, the interpreter is able to increment `count`, because `home` refers to the method context where the `count` variable is stored. Figure 3.53 on the facing page shows the contexts involved.

A major problem with the implementation described in [GR84], is that the allocation, initialization, and deallocation of contexts is expensive. In [Mos87], an attempt is made to rectify this situation. The idea is to recycle contexts by chaining unused contexts into a doubly linked free-list. The free-list is initialized with a number of preallocated contexts. When invoking methods due to message sends, unused contexts are grabbed from the

Figure 3.52 Counting the number of elements in a collection

```
size = (
   | count |
   count := 0.
   self do: [ :e | count := count + 1 ].
   ^count
)
```

Figure 3.53 Contexts involved in elements counting



free-list. When the methods later return, their contexts are re-inserted into the free-list. This scheme allows for very fast allocation of contexts, and since contexts are reused, it reduces the pressure on the garbage collector.

Unlike activation frames in procedural programming languages, contexts are not necessarily used in last-in first-out (LIFO) way. Block contexts are objects, and references to them can be stored in the fields of other objects. When that happens, the block context is said to *escape*, and it must be kept in memory until no references to it exist. Escaped block contexts still hold references to their home contexts, which too must be retained. This is trivial in the straight-forward implementation, since the garbage collector only deallocates objects that are not referenced. When reusing contexts through a free-list, care must be taken not to insert and thereby reuse escaped contexts.

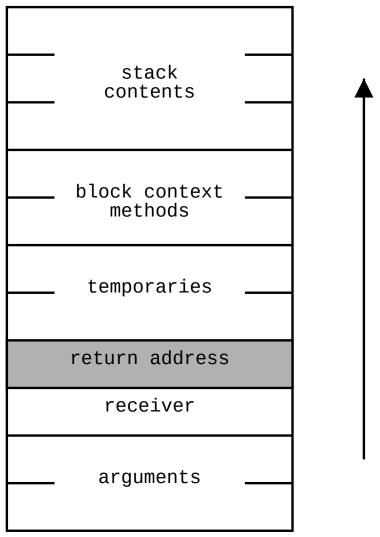
Most Smalltalk contexts do not escape, but without escape analysis, as described in [Bla99], it is impossible to tell which ones do. According to [DS84], more than 85% of all contexts do not escape. In the same paper, Deutsch and Schiffman present a model where contexts can be in one of three states. Normal contexts are by default in the *volatile* state. Volatile contexts are allocated on the stack and can safely be removed when the method returns. Contexts that exist in the heap as normal objects are called *stable*. Block contexts are created in the stable state. If a pointer is generated to a volatile context, it is turned into a *hybrid* context by filling out some fields so the context looks like an object, and preallocating an object to be used in case the context has to enter the stable state. Hybrid contexts still exist on the stack, but may not be removed. A hybrid context is turned into a stable context if its method returns, or if a message is sent to it. This approach eliminates many method context allocations, but requires tracking when a context *might* escape, and is thus a conservative approach. Stack-allocated contexts also have to include space for the fields that must be filled out in case the context must be made hybrid. This space is wasted if the context remains volatile throughout its lifetime.

In the presence of an optimizer, it is possible to avoid the expensive context allocation by means of *inlining*. Inlining eliminates calls and returns by expanding the implementation of called methods in their caller. Consider the code in figure 3.52 on the page before. If the `do:` send is recursively inlined down to where `[ :e | count := ... ]` is evaluated, there is no need to allocate the block context. In effect, inlining helps the optimizer realize that the block never escapes. This kind of optimization is used in the dynamic compiler of the Strongtalk virtual machine [BBG$^+$b].

As explained in section 2.1.2, we have extended the Smalltalk syntax with static type declarations for blocks. Combined with *type-based selectors*, this allows us to enforce LIFO behavior for blocks. Type-based selectors is a way of internally rewriting send selectors to reflect the static type of the arguments. Consider the `do:` send in the `size` method in figure 3.52 on the preceding page. The argument to the send is a block context. Therefore, the source code compiler rewrites the selector for the send to `do:[]`. This way the method lookup will always find a target method that expects a block context argument. Type-based selectors and type tracking makes it possible for the compiler to disallow block contexts to be returned or stored in object fields. Consequenctly, block contexts cannot escape, and there is thus no need to store method and block contexts in the object heap. Instead, they can be allocated on a stack, just like activation frames. Figure 3.54 on the next page shows the layout of stack-allocated method contexts in our system. Notice that due to the right-to-left evaluation order, the receiver is above the arguments.

To clarify how method and block contexts interact, consider figure 3.55 on page 58. It shows stack-allocated contexts for the Smalltalk code in figure 3.52 on the page before. It consists of three separate contexts, two

Figure 3.54 Stack-allocated method contexts



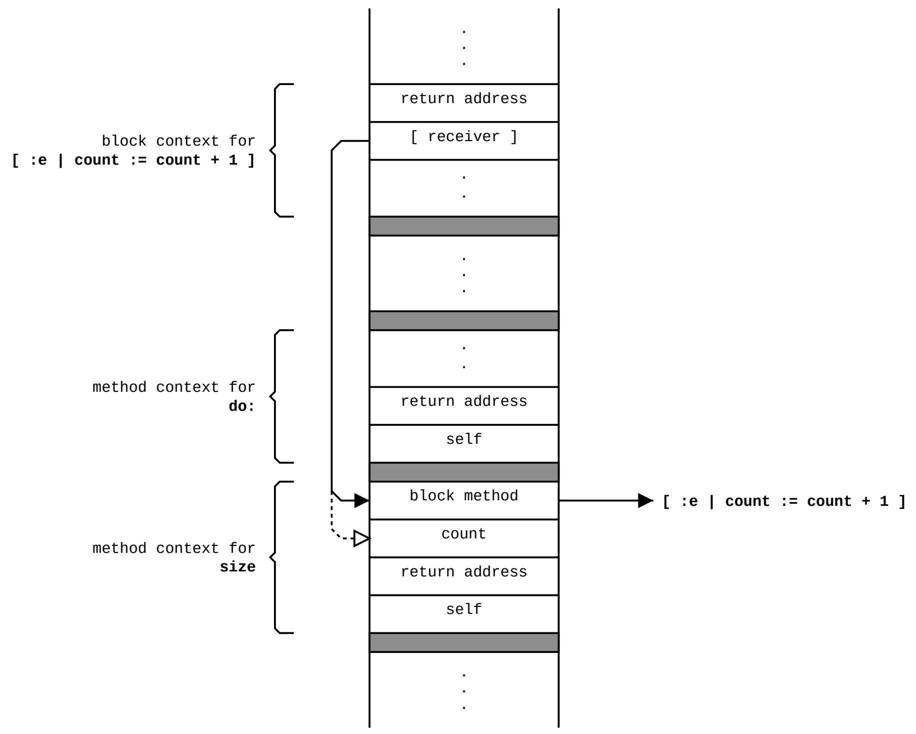
method contexts for `size` and `do:`, and one block context. At the bottom of the stack is the method context for the `size` method. Just above the temporary variable, `count`, there is a reference to an anonymous method. This method contains the code for the block argument to `do:`. The block context at the top of the stack is allocated in response to a `value:` send to the block. The block itself is just a pointer to the block method reference in the method context for `size`. This way, a single pointer encodes both the home context and code associated with the block. In the given example, the `count` temporary in the home context can be incremented from within the block by going through the receiver block pointer. The offset from the receiver block pointer to the temporary is known at compile-time.

Our novel design and implementation of LIFO blocks allow efficient interpretation of block-intensive Smalltalk code. For such code, our virtual machine is more than four times as fast as other interpreted virtual machines for Smalltalk. See section 6.2.2 for details on the block efficiency in our system.

3.2.2.3 Threads and Execution Stacks

Many modern virtual machines rely on the underlying operating system for providing threads and their associated execution stacks. The benefit is that the virtual machine does not have to implement any scheduling;

Figure 3.55 Stack-allocated contexts



the threads are automatically scheduled by the operating system. In most cases this is the only way of effectively using the processing power of *symmetric multiprocessing* (SMP) machines, and share physical resources with threads not running on the virtual machine. Even state-of-the-art embedded devices are not likely to incorporate SMP techniques anytime soon, so we will focus on single processor machines throughout this section.

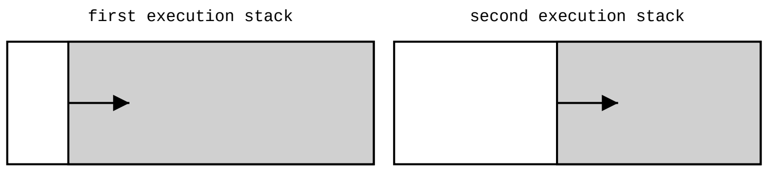
When running on operating system threads, the operating system provides execution stacks for the threads. Some popular languages allow the programmer to manipulate pointers to elements on the execution stack. Consider the C code in figure 3.56 on the facing page. During execution of the `fscanf` function, pointers to `rows` and `columns` exist; either in registers or on the execution stack. An operating system that supports this code is faced with the problem that it does not know where these pointers are; it depends on the C compiler. Without this knowledge, an execution stack cannot be moved, because a move requires such pointers to be identified and updated.

Figure 3.56 Scanning formatted input on `stdin` in C

```
void main(int argc, char** argv) {
   int rows, columns;
   fscanf(stdin, "%d x %d", &rows, &columns);
}
```

When execution stacks cannot be moved, they cannot be extended at runtime; they must have the right size when allocated. The problem is illustrated by figure 3.57. The first stack does not use all the memory reserved for it. Depending on the behavior of the thread using it, it may need more memory in the future. Therefore, the second execution stack cannot be allocated in the unused part of the first execution stack. Operating systems often reserve several kilobytes for each execution stack. Unfortunately, physical memory is a scarce resource on many systems. A widely used technique is to use *virtual memory* for execution stacks. By committing physical memory to execution stack pages on-demand, the operating system can avoid wasting physical memory on unused execution stack pages.

Figure 3.57 Non-extendable execution stacks



The smallest amount of physical memory that can be committed to an execution stack is one memory page. Even on small systems, the typical size of a memory page is around four kilobytes. This means that execution stacks provided by an operating system tend to take up more memory than needed. In our system, several of the execution stacks used for handling interrupts use less than 512 bytes of memory.

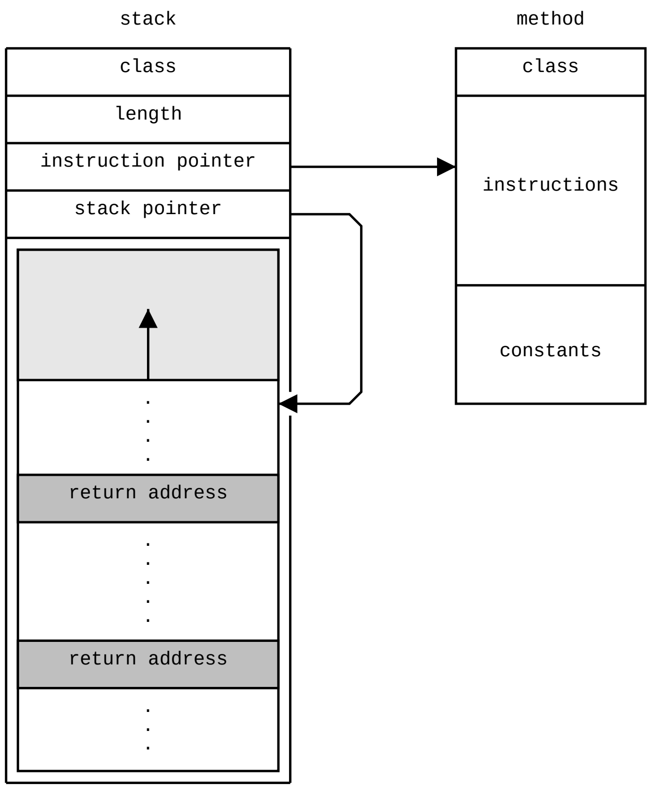
Another problem is that embedded processors rarely have sophisticated memory management units (MMU). The Motorola M·CORE M210 processor is an embedded 32-bit processor for industrial control and measurement, health care equipment, and scientific instrumentation. Unfortunately, it does not have an MMU, and as a consequence it does not support virtual memory. Operating systems for such processors have a difficult time managing execution stacks without wasting memory.

It is not only small systems that suffer from execution stacks provided by the operating system. On large 32-bit server systems, the four gigabyte address space provided by the processors can easily get too crowded. On many platforms, including Microsoft® Windows®, the default execution stack size is one megabyte, which makes it theoretically impossible to allocate more than 4,096 threads. In practice, the limit is less than half of that. This can be a serious problem for database systems, telephone switches, web servers, and application servers, where it is desirable to have one thread per connection and still be able to handle tens of thousands of simultaneous connections.

To avoid resource sharing problems and artifical limitations on the number of threads, we want our virtual machine to be able to run directly on the hardware, without the support of an operating system. Therefore, we manage execution stacks entirely within our system, even when our virtual machine is hosted on top of an operating system such as Linux. We have chosen to allocate execution stacks as objects in the heap. This enables uniform resource management, which means that unused stacks will be reclaimed automatically by our garbage collector. Figure 3.58 on the facing page shows the layout of execution stacks in our system. Execution stacks have varying lengths, encoded in an explicit length field. In this sense, execution stacks are very similar to arrays (see section 3.1.2). The stack pointer in an execution stack points to the top element of the used stack contents. The elements above this constitute the unused part of the stack contents.

In addition to holding stack contents, our execution stacks also act as *coroutines*. This means that they can be suspended and resumed later on. To support this, execution stacks are equipped with an instruction pointer field. When an execution stack is suspended, the current instruction pointer is saved in the field. When it is resumed, the instruction pointer is restored. In our scheduler implementation, we rely on an operation that transfers control from one execution stack to another. The operation performs a suspension of the active execution stack and a resumption of the other execution stack atomically. See section 5.3 for more details on scheduling.

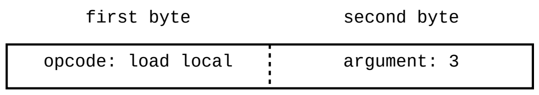
Figure 3.58 Layout of an execution stack



3.2.3 Instruction Set

This section gives a description of the instruction set used in our virtual machine. Instructions constitute the executable parts of methods, and as such they are equivalent to *bytecodes* in Java. Individual instructions consist of an *opcode* and an argument. As shown in figure 3.59, the opcode is a one byte numeric encoding of the parameterized instruction operation, and the argument is its one byte parameter.

Figure 3.59 Instruction encoding for `load local 3`



It is worth noticing that the instruction encoding is completely uniform. This has several advantages. First of all, the decoding of opcode and argument is the same for all instructions. This simplifies going from one instruction to the next, and it enables optimizations such as *argument prefetching* (see section 3.2.4.2). Secondly, when all instructions have the same size, it is possible to go from one instruction to the instruction preceeding it.

In a stack-based virtual machine architecture such as ours, almost all of the instructions access elements on the stack. Some virtual machines address elements relative to the frame pointer which marks the beginning of the current context. This is the case for Java virtual machines. Instead, we have chosen to address elements relative to the top of the stack. This means that we do not have to have a frame pointer, and we can use the same instruction to access both arguments and local variables.

3.2.3.1 Load Instructions

Several instructions load objects onto the stack. Three of the instructions are used to access variables and locals. The two remaining instructions load constants from the constant section and new blocks respectively.

`load local`

The instruction argument is used as an index into the stack. Indexing starts at zero from the top of the stack. The stack element at the index is loaded and pushed on the stack.

`load outer local`

The block at the top of the stack is popped. The instruction argument is used as an index into the stack. Indexing starts at zero from the block context method in the home context of the block. The stack element at the index is loaded and pushed on the stack. This instruction is used in the example shown in figure 3.55 on page 58 to read from `count`.

`load variable`

The instruction argument is used as a variable index into an object popped from the stack. The variables are indexed from one. The variable is loaded from the object and pushed on the stack.

`load constant`

The instruction argument is added to the instruction pointer to form a direct pointer to a constant in the constant section of the current method. The constant is loaded and pushed on the stack.

`load block`

A new block, with home context in the current context, is pushed on the stack. The instruction argument is the number of elements between the top of the stack and the block context method.

3.2.3.2 Store Instructions

Store instructions are used to modify the state of variables and locals. The instructions correspond to the first three load instructions. They leave the element they store on the stack for later use.

`store local`

The instruction argument is used as an index into the stack. Indexing starts at zero from the top of the stack. The stack element at the index is overwritten with an element loaded from the top of the stack.

`store outer local`

The block at the top of the stack is popped. The instruction argument is used as an index into the stack. Indexing starts at zero from the block context method in the home context of the block. The stack element at the index is overwritten with an element loaded from the top of the stack. This instruction is used in the example shown in figure 3.55 on page 58 to write to `count`.

`store variable`

The instruction argument is used as a variable index into an object popped from the stack. The variables are indexed from one. The variable is overwritten with an element loaded from the top of the stack.

3.2.3.3 Send Instructions

Sends are used to dispatch to methods based on the message selector and the dynamic type of the receiver. As such, they are the foundation of any Smalltalk execution model.

`send`

The receiver of the message is loaded from the top of the stack. The instruction argument is added to the instruction pointer to form a direct pointer to the selector in the constant section of the current method. The selector is loaded and used to find the target method in the class of the receiver. A pointer to the next instruction is pushed on the stack, and control is transferred to the first instruction in the target method.

`super send`

The instruction argument is added to the instruction pointer to form a direct pointer to a (class, selector) pair in the constant section. The class and the selector are loaded from the pair. The selector is used to find the target method in the loaded class. The target method is invoked by pushing a pointer to the next instruction on the stack, and by transferring control to the first instruction in the target method. The receiver of the message is at the top of the stack, but it does not affect the method lookup. This instruction is used to send messages to `super`. Therefore, the class of the receiver is always a subclass of the class loaded from the constant section. The interpreter must know which class to look up the method in because it may have been overridden in subclasses. If the interpreter always starts the lookup in the receiver class, it risks invoking the same method repeatedly.

`block send`

This instruction is identical to `send`, except that the receiver must be a block. Thus, the target method lookup is always performed in the block class.

3.2.3.4 Return Instructions

In Smalltalk, there are two ways of returning. A *local return* is used to return to the previous context on the stack, ie. the context that created the current context. For method contexts, this transfers control to the message sender, or in procedural terms, to the method that called the currently executing method. For block contexts, control is transferred to the context that initiated the evaluation of the block. When evaluating blocks, it is also possible to return from the method context in which the block was created. This is known as a *non-local return*, and it is described in section 2.1.2.

`return`

The result of the message send is popped from the stack. The instruction argument is the number of elements above the instruction pointer which was saved on stack by the `send`. Those elements are popped from the stack. The instruction pointer is popped from the stack, and the element at the top of the stack is overwritten with the result. Execution continues at the restored instruction pointer.

`non-local return`

The result of the message send is popped from the stack. The instruction argument is the number of elements above the instruction pointer which was saved on stack by the `send`. As shown in figure 3.55 on page 58, the receiver block is just beneath the instruction pointer on the stack. The contexts above the home context of the receiver are popped from the stack, and the result is pushed to the stack. If the new top context is a method context, the result is returned as if `return` had been executed. Otherwise, the non-local return is restarted from the top block context.

3.2.3.5 Miscellaneous Instructions

Two instructions do not fit in the previous categories. The instructions are used for removing elements from the stack and calling native code in the virtual machine respectively.

`pop`

The number of elements specified by the instruction argument are popped from the stack.

`primitive`

The virtual machine primitive code associated with the instruction argument is executed. The result of the execution is pushed on the stack. If the primitive code succeeds, the method containing the instruction returns to the sender as if a `return` instruction had been executed. Otherwise, execution continues at the next instruction, allowing the virtual machine to handle primitive failures in Smalltalk.

3.2.4 Optimizations

This section gives an overview of the interpreter optimizations we have implemented. Some of the optimizations presented here are difficult, if not impossible, to implement in off-the-shelf high-level languages. For that reason, we have implemented our interpreter directly in native code. We have optimized both the Intel IA-32 and ARM native interpreter implementations. Since the ARM architecture is unfamiliar to many, we will use the IA-32 version in code examples.

3.2.4.1 Register Caching

One of the advantages of implementing the interpreter in native code is that it is possible to cache the information that is accessed most frequently in registers. Most optimizing, high-level language compilers attempt to do the same, but without profiling the runtime characteristics of the interpreter, it is impossible to achieve a near-optimal result.

The following table shows the register assignment for our interpreter implementation. Since there are more registers available on ARM processors than on Intel IA-32 processors, we have chosen to cache more information on ARM-based platforms. The *stack limit* register is used for fast stack overflow checks. Section 3.2.4.2 explains the use of the *prefetched argument* register, and section 3.2.4.4 accounts for the use of the *top-of-stack cache* register. The purpose of the *interpreter dispatch table* register will become apparent after reading section 3.2.4.3.

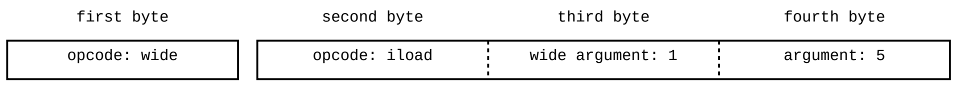
	Intel IA-32	**ARM**
Stack pointer	`esp`	`sp`
Stack limit	`ebp`	`fp`
Instruction pointer	`esi`	`r1`
Prefetched argument	`edi`	`r2`
Top-of-stack cache	`eax`	`r0`
Interpreter dispatch table		`r4`
Small integer class		`r9`

We have found that register caching in the interpreter is key to achieving high performance. This way, our experience is similar to what is reported in [MB99]. Register caching of the stack pointer, stack limit, instruction pointer, and prefetched argument yields a speedup of 26–34%. See section 6.2.3 for details on the performance impact of register caching.

3.2.4.2 Argument Extension and Prefetching

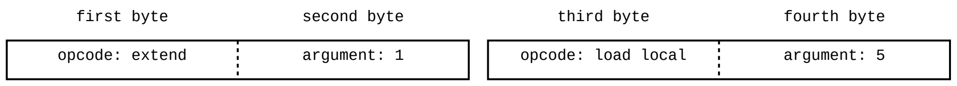
As shown in figure 3.59 on page 61, we have only reserved one byte for the instruction argument. This accounts for most cases, but there are situations where arguments larger than 255 are needed. In Java, this is handled by introducing a prefix bytecode, `wide`, that widens the argument to the next bytecode. Figure 3.60 shows how an `iload` bytecode with a wide argument is encoded in Java. The effective argument to the `iload` has the wide argument (1) in bits 8–15 and its argument (5) in bits 0–7. The interpreter must know if the executing bytecode is prefixed with `wide`, because the argument encoding for wide bytecodes is different. In practice, this requires prefix-dependent interpretative routines for at least some of the bytecodes.

Figure 3.60 Java bytecode encoding for `iload 261`



We have chosen to handle large arguments by mimicking the index extension mechanism found in SELF. If the argument of an instruction does not fit in one byte, the instruction is prefixed with an *extend* instruction. This is similar to the index-extension bytecode used in SELF [CUL89]. The argument to the extend instruction specifies the most significant bits of the argument to the following instruction. Figure 3.61 shows how a `load local` instruction with an argument of `261` is encoded. The effective argument to the `load local` instruction has the extend argument (1) in bits 8–15 and its own argument (5) in bits 0–7. This way the uniformity of our instruction set is preserved.

Figure 3.61 Instruction encoding for `load local 261`



To handle the extend instruction efficiently, we have designed and implemented a new scheme for prefetching instruction arguments. Each interpretative routine assumes that its argument has been prefetched from memory and put in a register for fast access. On the Intel IA-32 architecture, the register is `edi`. In the common case where the argument fits in one byte, it is loaded and zero-extended at the end of the interpretation of the previous instruction. In case of extension prefixes, the interpretation of the extend instructions handles the widening of the argument, so the argument is extended and prefetched when the next instruction is interpreted.

Figure 3.62 shows our extension implementation. In order to prefetch the argument, we rely on the fact that the epilog code of the interpretative routine knows the encoding of the next instruction. Thus, instruction set uniformity is not only aesthetically pleasing, but also convenient. The first native instruction shifts the current argument eight positions to the left. Due to the shift, the eight least significant bits of the argument are zeroes before they are overwritten with the argument byte of the next native instruction, using bitwise or.

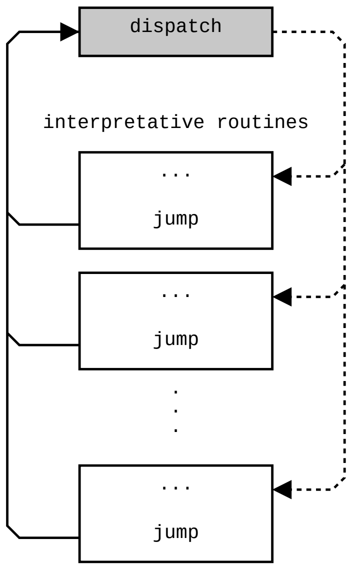
Figure 3.62 Intel IA-32 native argument extension implementation

```
shl    edi, 8                  ; extend the argument
or     edi, byte ptr [esi + 3] ; overwrite low bits with next argument

...                            ; go to next instruction
```

3.2.4.3 Interpreter Threading

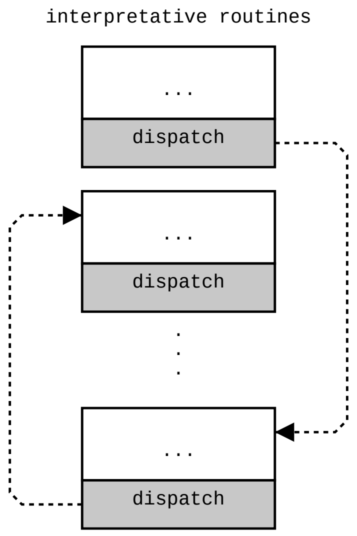
Interpreters spend a lot of time transferring control from one instruction to the next. This process is called *dispatching*, and it consists of jumping to the interpretative routine for the next instruction. The classic way of doing it, is to have a loop with a single code location that does the dispatch. This situation is depicted in figure 3.63. The solid arrows represent direct jumps, whereas the dotted arrows represent indirect jumps. Unfortunately, the performance of this simple solution is suboptimal. Modern pipelined processors do speculative work across indirect jumps by predicting the jump target. In case of misprediction, the speculative work must be thrown away, which causes processor stalls and performance loss. One common prediction strategy is to associate the last jump target with the jump site, through a processor-internal cache known as the *branch target buffer*. With only one point of dispatch, the prediction will only succeed if a single instruction is executed several times in a row.

Figure 3.63 Classic interpretation with one dispatch point



An alternative to classic interpretation is described in [Bel73]. Instead of having one shared dispatch code location, the interpretative routine for each instruction is equipped with its own dispatch code. This means that the interpreter jumps directly from the interpretation of one instruction to the interpretation of the next without going through a shared dispatch point. Thus, a thread of execution is sewn through the interpreter. Figure 3.64 shows how this can be implemented. By *threading* the interpreter, it is possible to save a direct jump for each instruction interpretation, and also increase the indirect jump prediction accuracy rate by up to 45% [EG01].

Figure 3.64 Threaded interpretation with multiple dispatch points



An interesting side effect of the multiple dispatch points in a threaded interpreter is that having more instructions, or several slightly different versions of the same instruction, can increase performance. Due to *stack caching*, the Hotspot interpreter has multiple versions of some instructions; see section 3.2.4.4. This is the reason why the Hotspot interpreter is faster than the CLDC Hotspot interpreter on some micro-benchmarks.

In the PDP-11 implementation described in [Bel73], the opcode of the next instruction is used as an index into a table of references to interpretative routines. This is known as indirect threading. It is also possible to let the opcode be the address of the interpretative routine. The benefit of this approach is that the table indirection can be avoided, thereby improving performance. The direct threading approach has two significant disadvantages. First, using addresses as opcodes take up more bits than other opcode encodings. In most modern systems, addresses are at least 32 bits wide, whereas the opcodes of most virtual machine instruction sets fit in 8 bits. Second, due to code changes or relocation, the addresses of the interpretative routines tend to change frequently during the development of the virtual machine. When the addresses of the interpretative routines change, the opcode encoding in the interpreter and language compiler must also be changed. Although the changes to the interpreter and compiler can be made automatically, the volatile instruction encoding means that Java-like class files cannot store the instruction opcodes directly since they can change between virtual machine versions.

We have optimized our interpreter by means of indirect threading. To illustrate the implementation, consider the code in figure 3.65. Recall that register `esp` is the stack pointer, `esi` holds the current instruction pointer, and that `edi` is the argument of the current instruction. The first instruction pushes the local indicated by the argument to the top of the stack. The four instructions following that are all part of the dispatch to the next instruction. Notice how the dispatch increments the instruction pointer and prefetches the argument for the next instruction.

Figure 3.65 Intel IA-32 native implementation of `load local`

```
push   [esp + edi * 4]    ; push local to the top of the stack

add    esi, 2             ; increment instruction pointer
movzx  ecx, [esi + 0]     ; load opcode from instruction pointer
movzx  edi, [esi + 1]     ; load argument from instruction pointer
jmp    [table + ecx * 4]  ; dispatch through dispatch table
```

We have evaluated the performance impact of threading the interpreter. We have found that interpreter threading yields a speedup of 21–32%, thereby reducing the time spent dispatching from 65–85% of the total execution time to 56–78%. Section 6.2.4 gives more details on this evaluation.

3.2.4.4 Stack Caching

When interpreting instructions for stack-based virtual machines, fast access to stack cells is paramount. Most modern processors treat stacks as any other part of memory, and consequently stack access is as slow as memory access. As explained in section 3.2.4.1, frequently used memory cells can be cached in machine registers. The result is reduced memory traffic and increased performance. If registers are used for caching stack cells, it is referred to as *stack caching*. For example, a `load local` instruction followed by a `store local` instruction will typically require four memory accesses to load the local, push it on the stack, load it from the stack, and store it into the new local. In a system where the top element of the stack is cached, the same instruction sequence will only require two memory accesses, because the local is never stored on the physical memory stack but kept in a register.

The most straight-forward stack caching strategy is to cache a constant number of the top stack cells $(s_1, ..., s_n)$ in dedicated registers $(r_1, ..., r_n)$. Unfortunately, this solution is not particularly attractive, since pushing an element to the top of the stack is rather involved. Before assigning the element to r_1, the interpreter must spill r_n to s_n and assign r_i to r_{i+1} for all $1 \leq i < n$. Unless register assignment is inexpensive and n is not too large, this approach may actually hurt performance.

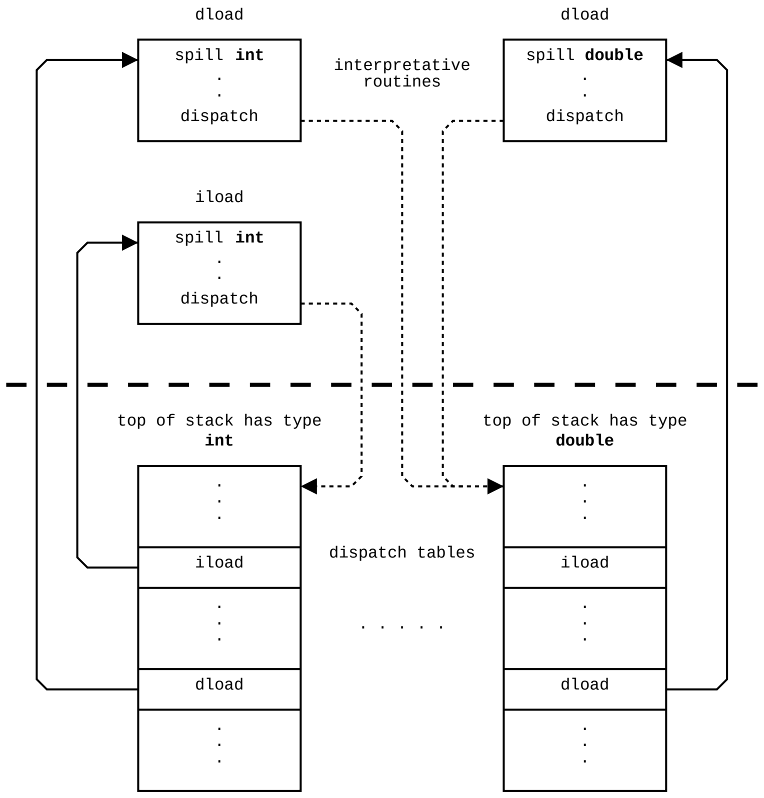
An improvement to the straight-forward strategy is to have several stack cache states. The cache state is a mapping from stack cells to registers, such as $(s_1 \rightarrow r_1, ..., s_n \rightarrow r_n)$. Duplicating the stack top can be implemented by changing the cache state to $(s_1 \rightarrow r_1, s_2 \rightarrow r_1, ..., s_n \rightarrow r_{n-1})$. Thus, stack operations that would normally require spilling and register assignments can be implemented simply by changing the cache state. Notice, however, that it may take an infinite number of cache states to handle arbitrary instruction sequences by state change only.

Due to the many cache states, the interpreter has to be extended to use stack caching. It must be capable of interpreting instructions in all possible cache states, and changing to another cache state must be supported. An efficient way of achieving this is to introduce cache state specific interpretative routines for all instructions. To ensure that control is transferred to the right routine, the cache state must be taken into account when dispatching to the next instruction. In an indirect threaded interpreter, this can be done by introducing a new dispatch table for each state. When dispatching, the indirect jump is directed through the dispatch table associated with the current cache state. In [Ert95], this kind of stack caching is called *dynamic stack caching*, since the interpreter switches between cache states at runtime. The disadvantage of this approach is that the code size of the interpreter is compromised, since several dispatch tables and interpretative routines have to be added.

As mentioned in [Ert95], it is never disadvantageous to cache the top element of the stack in a dedicated register, provided that enough registers are available. Therefore, the optimization known as *top-of-stack caching*, is implemented in interpreters for many languages. However, some languages, such as Java, allow the top-of-stack element to be of many different types. On a 32-bit platform, such as Intel IA-32, elements of type `int`, `float`, and `reference` fit in one machine word, whereas `long` and `double` elements require two. This means that it may be impossible to use the same register for caching elements of any Java type.

The Hotspot virtual machine solves this problem by introducing cache states and top-of-stack registers corresponding to the different types. A small subset of the interpretative routines and the dispatch tables used are shown in figure 3.66. As an example, observe the `dload` routine in the `int` cache state. Before loading the `double` top-of-stack register, the routine spills the current `int` top-of-stack register. Since the top-of-stack is now a `double`, the routine changes to the `double` state by dispatching through the appropriate dispatch table.

Figure 3.66 Stack caching states in the Hotspot virtual machine



There is no need to have more than one cache state in our system. All the elements pushed on the stack, including return addresses, small integers, and blocks, fit in one machine word. This way, we can dedicate a register to caching the top of the stack. It should be emphasized that the register *always* contains the top of the stack. As a consequence, the register contains the return address when executing the first instruction in a method. In the Intel IA-32 implementation, we use `eax` as the top-of-stack cache register. To illustrate this, compare the stack-cached implementation shown in figure 3.67 with the implementation described in figure 3.65 on page 71. On average, this optimization gives a speedup of 0.3–3.0%; see section 6.2.5.

Figure 3.67 Stack-cached implementation of `load local`

```
push  eax                   ; spill top-of-stack cache register
mov   eax, [esp + edi * 4]  ; load local from the stack

...                         ; go to next instruction
```

3.2.4.5 Lookup Caching

When interpreting dynamically typed object-oriented languages, such as Smalltalk, much of the execution time is spent looking up methods for execution. Method lookup means finding a method matching a given selector in the receiver class or its superclasses. In its simplest form, the method lookup mechanism is slow because it involves a traversal of the superclass chain of the receiver class. Most systems incorporate a lookup cache to improve performance. In fact, implementing lookup caching in the Berkeley Smalltalk system increased execution speed by 37% [CPL84].

A lookup cache is a mapping from receiver classes and method selectors to methods. The mapping is typically implemented as a hash table, where the class and selector are used as input to a function that yields a table entry. Since different class and selector pairs can hash to the same entry, the entries are (class, selector, method) triples. This makes it possible to verify that the class and selector correspond to the entry found. If so, the method in the entry triple is the result of the lookup. Otherwise, the slow method lookup mechanism must be invoked. The result of the slow lookup can then be stored in the cache for later use.

The hash function has to minimize the number of cache conflicts and be efficient to compute. In [CPL84], three simple functions are studied. The functions are all of the form *hash*(class, selector) $\rightarrow$ class $\oplus$ selector, where the $\oplus$ operator is either `xor`, `and`, or `add`. For a particular interactive Smalltalk session, the function that resulted in the highest cache hit ratio was found to be `add`, at 94.8%. We have conducted our own hash function evaluation. According to our numbers, `add` performs slightly worse than `xor` for almost all cache sizes. With a cache size of 1024 elements and inline caching enabled, we get a hit ratio of 96.5% using `add` and 96.8% using `xor`. See section 6.2.7 for further details.

Figure 3.68 shows a part of our native lookup cache implementation. First the hash table index is computed in register `edx`. The `selector` and `class` are two registers used in this computation. The entry triple at the computed index is verified by comparing receiver classes and selectors. If both checks succeed, the method in the triple is invoked. The actual implementation used in our system is more complex, since it uses a two-level cache to minimize the cost of most of the first-level cache misses that are due to conflicts. Lookup caching improves performance by 20–40% depending on the benchmarks used. Section 6.2.8 gives more details on the evaluation of lookup caching.

Figure 3.68 Intel IA-32 native implementation of lookup caching

```
mov  edx, selector             ; index: (selector ^ class) & 0xfff
xor  edx, class
and  edx, 0xfff

cmp  class, classes[edx]       ; verify class
jne  cache_miss

cmp  selector, selectors[edx]    ; verify selector
jne  cache_miss

...                            ; invoke methods[edx]
```

In a system that uses direct pointers, elements in the lookup table may have to be rehashed during garbage collection. Recall that a direct pointer is the address of the object it refers to. Thus, the output of the simple hash functions depend on the addresses of the class and the selector. If the garbage collector decides to move any of these objects, the lookup table index of some cache elements may change.

Many virtual machines based on direct pointers simply clear the cache whenever the garbage collector is invoked. The benefit is that rehashing can be avoided, and that the pointers in the cache do not have to be traversed during garbage collection. However, there is a performance penalty involved in refilling the cache.

If the garbage collector is generational, most classes and selectors do not move during garbage collections. This means that most cache elements do not need rehashing. This can be exploited by leaving the cache unchanged when collecting garbage. If an element in the cache ends up at an outdated table index, the worst thing that can happen is that the next lookup will cause a cache miss.

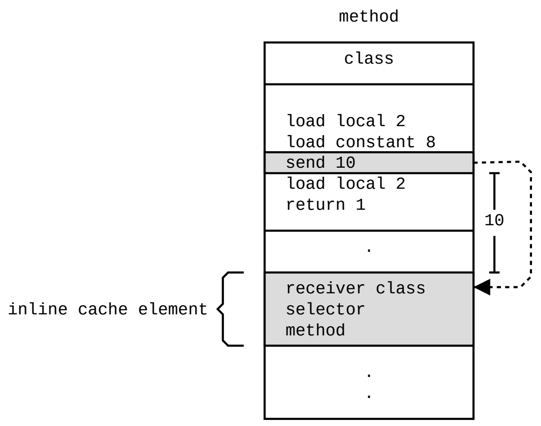
3.2.4.6 Inline Caching

The class of the message receiver remains constant for most Smalltalk send sites. Measurements reported in [DS84] have shown that about 95% of all dynamically executed sends invoke the same method repeatedly. By gathering send statistics in our system, we have found that the percentage is lower. In our system, 82% of sends invoke the same method repeatedly; see section 6.2.6. Such send sites are known as *monomorphic sends*, as opposed to *megamorphic sends* which have changing receiver classes. Monomorphic sends have the property that it is only necessary to perform the method lookup once; the result never changes and it is easily cached. Unfortunately, the lookup caching mechanism described in section 3.2.4.5 cannot guarantee that the method lookup will only be performed once. It is possible that the cache element that holds the result is overwritten by another element, if it happens to hash to the same lookup table index.

The Smalltalk implementation described in [DS84] pioneered a solution to the problem. Instead of relying solely on the shared lookup cache, the monomorphic send instructions have non-shared cache elements associated directly with them. Such a send-local cache element is never overwritten due to sharing, and as a consequence there is no need to lookup the method more than once. The technique is known as *inline caching*, since it essentially inlines the cache elements into the instructions.

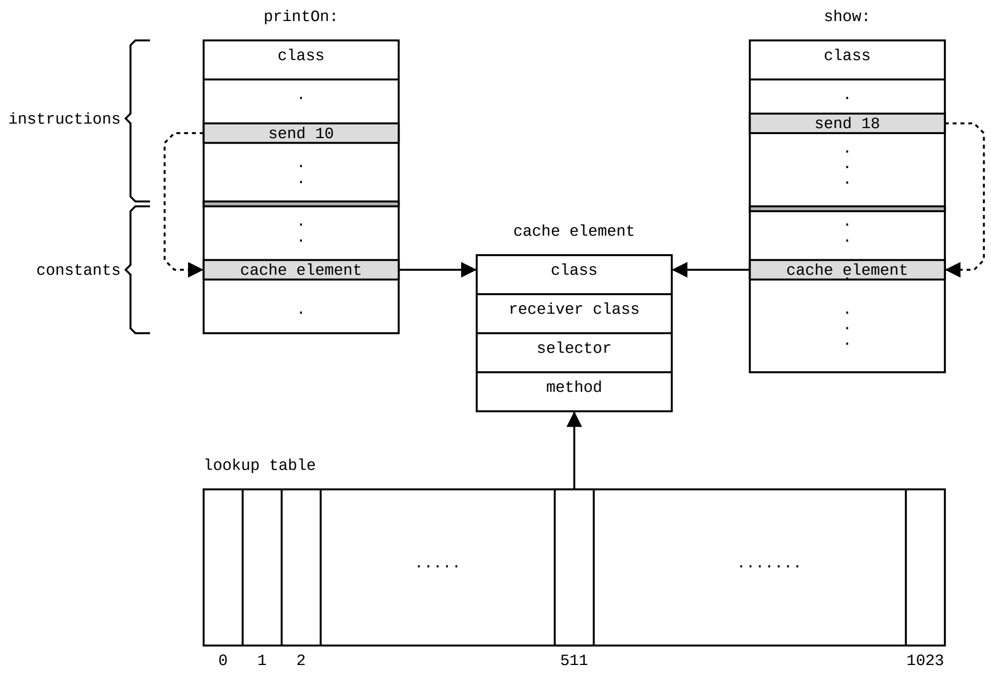
Recall that cache elements are (class, selector, method) triples. There is always a selector associated with sends, but when implementing inline caching in interpreters, one has to reserve space at the monomorphic send instructions for storing the class and the method. Unfortunately, all sends are potentially monomorphic, and therefore space must be allocated for all sends. Figure 3.69 on the facing page shows how methods can be extended to support inline caching. The first time a send instruction is executed, a method lookup is performed and the resulting method is stored in the inline cache element. The class of the receiver is also stored in the cache element. As long as future receiver classes are identical to the cached receiver class, subsequent executions of the instruction can invoke the cached method without any lookup. In this sense, send sites are assumed to be monomorphic until proven megamorphic. According to our measurements, inline caching improves performance by 14–23% over lookup caching. The combined effect of lookup caching and inline caching is a speedup of 38–48%. See section 6.2.8 for more details on the measurements.

Figure 3.69 Inline caches in constant section



To avoid reserving three entries in the method constant section for all sends, we have materialized our cache elements as objects. When a send site becomes monomorphic, we replace the selector constant with a pointer to a cache element object. If the send site later is proven to be megamorphic, the selector can be restored from the cache element. This way, no extra constants are required for non-monomorphic send sites. Unfortunately, separate cache elements for each monomorphic send site require four words each. To overcome this problem, we have designed and implemented a new technique, which we call *inline cache sharing*. The key observation is that cache elements themselves are immutable, and thus shareable between all monomorphic send sites that target the same method via identical receiver classes and selectors. To canonicalize the cache elements, we use the lookup cache table already in the system. By letting the table entries refer to cache elements, the elements can be found by means of hashing (see section 3.2.4.5). Figure 3.70 on the next page shows the relationship between methods, cache elements, and the lookup table. Inline cache sharing through the lookup cache table achieves a sharing of 88.9% of the optimal sharing. Compared to our system without such sharing, this results in an 18.8% reduction of the memory needed for classes, strings, and methods for a suite of benchmarks. See section 6.2.9 for more details on the evaluation of inline cache sharing.

Figure 3.70 Inline cache sharing through lookup cache table



It is also possible to have more than one cache element per send site. This generalization, described in [HCU91], is known as *polymorphic inline caching*. We have found that polymorphic inline caching does not improve performance for interpreters; see section 6.2.10. However, the runtime receiver class information in send caches can be valuable type information in dynamically typed systems. In the SELF system, and in Hotspot, it is used as *type feedback* from the runtime system to the optimizing compiler [HU94].

3.2.4.7 Dynamic Customization of Sends

In some cases, having concrete type information can enable a variety of performance optimizations. In dynamically typed systems, this information is only available at runtime. In this section, we will discuss how `send` instructions can be customized dynamically to adapt to the runtime type information available.

We have already covered inline caching; an optimization that is easily improved by customizing the `send` instructions based on the number of receiver classes they have encountered. We introduce two new instructions, `monomorphic send` and `megamorphic send`, and switch between them by dynamically rewriting the instructions. The interpreter rewrites from `send` → `monomorphic send` → `megamorphic send`. The following descriptions give the semantics of the new instructions in the presence of inline cache sharing.

`send`

The receiver of the message is loaded from the top of the stack. The instruction argument is added to the instruction pointer to form a direct pointer to the selector in the constant section of the current method. The selector is loaded and used with the receiver class to find a cache element. If no cache element is found, the `send` behaves as if it was a `megamorphic send`. Otherwise, the instruction is rewritten to `monomorphic send`, and the selector in the constant section is overwritten with the cache element. The rewritten instruction is restarted.

`monomorphic send`

The receiver of the message is loaded from the top of the stack. The instruction argument is added to the instruction pointer to form a direct pointer to a cache element in the constant section of the current method. The cache element is loaded and it is checked that the receiver class is identitical to the cached receiver class. If the check succeeds, a pointer to the next instruction is pushed on the stack, and control is transferred to the first instruction in the cached method. Otherwise, the instruction is rewritten to `megamorphic send`, and the cache element in the constant section is overwritten with the cached selector. The rewritten instruction is restarted.

`megamorphic send`

This instruction is identical to the `send` instruction described in section 3.2.3.

The virtual machine must dynamically rewrite send instructions in methods to support this kind of inline caching. In memory-constrained embedded systems, it is often desirable to burn program code such as classes and methods into *read-only memory*. If methods are put in read-only memory, it is impossible to rewrite instructions in them. To solve this issue, it is possible to introduce a new monomorphic send instruction that remains monomorphic even in case of inline cache misses. This instruction should only be used for sends that are likely to remain monomorphic throughout the execution; all other sends should be megamorphic. It is possible to compute the set of sends that should use this new monomorphic send instruction by doing a training run of the system, before burning the methods into read-only memory.

In object-oriented languages, and Smalltalk in particular, it is common to introduce methods for accessing object variables. Such methods are called *accessor methods*, and an example is given in figure 3.71. The execution of an accessor method involves sending a message, loading the variable, and returning.

Figure 3.71 Accessing the size of lists

```
List = Collection (
   | size |

   size = ( ^size )

   ...
)
```

In a system with inline caching, it is possible to improve performance of accessor methods. The observation is that the message send and the return are superfluous; only the actual loading of the variable is important. The `send` instruction can be extended to check if the target method is an accessor. If it is, the variable can be loaded by the `send` instruction, without executing the instructions in the method. Unfortunately, checking if a target method is an accessor requires at least a few native instructions. Thus, the performance of non-accessor sends degrades. To avoid the active check and the implied performance degradation, dynamic send customization can be used. When rewriting the `send` instruction to a `monomorphic send`, it can be encoded in the instruction whether or not the cached method is an accessor. This only works if the system uses inline caching, and it requires an additional instruction:

monomorphic accessor send

This instruction is identical to `monomorphic send`, except that cache hits do not transfer control to the cached method. Instead the method is known to be a variable accessor method, and the variable index is decoded from the instructions in the method. The variable is accessed as if a `load variable` instruction with the index as argument had been executed.

We have found that using inline accessors gives a speedup of 5–8% in an otherwise optimized interpreter. Section 6.2.11 gives details on our evaluation of this optimization.

Another important category of methods that can be handled efficiently by customizing sends is *primitive methods*. A primitive method is a method that begins with a `primitive` instruction. Figure 3.72 shows an example of such a method. Note that the `throw` message is not sent if the primitive succeeds, because a successful execution of a primitive will return from the method, as described in section 3.2.3.5.

Figure 3.72 Adding small integers by invoking primitive code

```
SmallInteger = Integer (

   + other = ( {Primitive:25} throw )

   ...
)
```

When the execution of the virtual machine primitive code succeeds, the message send and the return implicit in the `primitive` instruction are unnecessary. Most primitive code executions succeed in our system. Therefore, it is beneficial to let the `send` instruction execute the primitive code. If it succeeds, there is no need to execute any instructions in the primitive method. This optimization yields a speedup of 2–9%; see section 6.2.11. The `monomorphic send` instruction can be customized to encode if the cached method is a primitive method. For this, we have added an additional instruction:

monomorphic primitive send

This instruction is similar to `monomorphic accessor send`, except that the cached method is known to begin with a `primitive` instruction. The primitive index is decoded from that instruction, and the corresponding virtual machine primitive code is executed. If the primitive code succeeds, the receiver at the top of the stack is overwritten with the result. Otherwise, a pointer to the next instruction and the result are pushed on the stack, and control is transferred to the *second* instruction in the cached method.

3.2.4.8 Inlining of Control Structures

Most high-level languages have built-in support for control structures, such as conditional processing and looping. Smalltalk uses evaluation of blocks to implement these control structures. Figure 3.73 shows how conditional processing is used in Smalltalk. The result of evaluating the `self > other` condition is either `true` or `false`. The `true` object responds to `ifTrue:ifFalse:` by sending `value` to the first argument, whereas the `false` object sends `value` to the second argument.

Figure 3.73 Implementation of `max:` on `Integer`

```
max: other = (
   ^self > other ifTrue: [ self ] ifFalse: [ other ]
)
```

Figure 3.74 on the next page shows the instructions generated by the compiler for the `max:` method implemented in the integer class. First, the two methods that hold the code for the blocks are pushed on the stack. Due to right-to-left evaluation order, the block arguments are created and pushed onto the stack, before the condition is evaluated. Finally, the result of the condition evaluation is sent the `ifTrue:ifFalse:` message.

Given the simplicity of the source code in figure 3.73, the code in figure 3.74 seems rather involved. In fact, most of the instructions are only necessary because conditional processing relies on blocks. The first four instructions, two `load method` instructions and two `load block` instructions, are required to setup the block arguments. The `load method` instructions are `load constant` instructions, where the constant is a method. The `load outer local` instructions are used in the block methods to access `self` and `other` in the `max:` context. Finally, the `ifTrue:ifFalse:` send and the two return instructions in the block methods are necessary to create and return from contexts associated with the blocks.

Figure 3.74 Instructions for `max:`

```
load method
   load local 1
   load outer local 2
   return 0
load method
   load local 1
   load outer local 4
   return 0
load block 0
load block 2
load local 6
load local 6
send >
store local 1
pop 1
send ifTrue:ifFalse:
return 4
```

In the virtual machine specification in [GR84], it is described how the original Smalltalk-80 implementation used inlining to eliminate blocks and sends associated with conditional processing. Recall that inlining is a way to avoid message sends by integrating the target method in the sending method. By integrating the boolean `ifTrue:ifFalse:` methods into the sender, it is possible to remove several instructions. Figure 3.75 shows how inlining reduces the number of instructions used in the `max:` implementation by more than 35%.

Figure 3.75 Instructions for `max:` with inlining of `ifTrue:ifFalse:`

```
  load local 2
  load local 2
  send >
  load constant true
  branch forward if equal -> higher
  load local 3
  return 2

higher:
  load local 2
  return 2
```

The instruction set described in section 3.2.3 does not support inlining of `ifTrue:ifFalse:`. As shown in figure 3.75, we have introduced a conditional branch instruction to rectify this situation. To support inlining of additional control-structures, we have extended our instruction set with the following three instructions.

`branch backward`

The instruction argument is subtracted from the instruction pointer. The effect is that the dispatch to the next instruction will cause control to be transferred to an instruction preceding the current instruction.

`branch forward`

The instruction argument is added to the instruction pointer. The effect is that the dispatch to the next instruction will cause control to be transferred to an instruction following the current instruction.

`branch forward if equal`

The match element is popped from the stack. The element at the top of the stack is compared against the match element. If they refer to the same object, the branch is taken as if a `branch forward` instruction had been executed. Otherwise, execution continues at the next instruction.

The `branch forward if equal` instruction is a general-purpose conditional branch. When it is used with the unconditional branches, it is possible to optimize Smalltalk looping control-structures, including iterating and repeating. Our source code compiler takes advantage of this and inlines the following sends:

- `ifTrue:` and `ifFalse:`
- `ifTrue:ifFalse:` and `ifFalse:ifTrue:`
- `whileTrue:` and `whileFalse:`
- `repeatUntil:` and `repeatWhile:`
- `and:` and `or:`

There are a few problems associated with inlining sends for control-structures. Because Smalltalk is dynamically typed, it is impossible to guarantee that the runtime receivers of `ifTrue:` sends actually are `true` or `false`. The standard Smalltalk semantics of sends to receivers of unexpected types includes lookup failures. This semantics should be preserved even when inlining the sends.

Another problem is that the sends visible in the source code are not necessarily represented directly in the generated instructions. This may have implications for debugging if the source code is not available, since correct decompilation of instructions to source code may be difficult. It is possible to solve this by adding inlining information to the instructions. In the Strongtalk system, such decompilation information was encoded in the opcodes of the branch instructions [BBG$^+$a].

We have implemented inlining of control structures in our system. On average, it yields a speedup of 30–57%. The performance impact is even higher for some micro-benchmarks. The loop benchmark described in appendix B runs 84% faster with inlining of control structures. See section 6.2.12 for more details on the evaluation of this optimization.

3.2.4.9 Superinstructions

Even though we have used threading to optimize the dispatch between instructions, the overhead due to indirect jumps remains a performance problem; see section 3.2.4.3. To eliminate some of the overhead, we have introduced *superinstructions*. Superinstructions are combinations of two or more instructions, and as such they are similar to the *superoperators* described in [Pro95]. Some Java virtual machines have similar constructs.

To keep the instruction set uniform, superinstructions have the same encoding as the instructions described in section 3.2.3. This implies that superinstructions have only one opcode and one argument. Combining two instructions requires that one of the instruction arguments is a constant, or that it can be expressed in terms of the other argument. Figure 3.76 on the next page shows how two instructions, with one constant argument, can be combined. The superinstruction shown is widely used in our system; it loads the value from a (name, value) pair stored in the constant section of the current method.

As stated, it is also possible to express one argument in terms of the other. Figure 3.77 on the following page shows another example of instruction combination. Since the receiver is just beneath the instruction pointer saved on the stack by the send, the superinstruction can be used to return `self` from a method. It would also have been possible to combine `load local` with `return 2` instructions, but the superinstruction shown is more general and therefore more useful.

Figure 3.76 Combining `load constant` and `load variable 2`

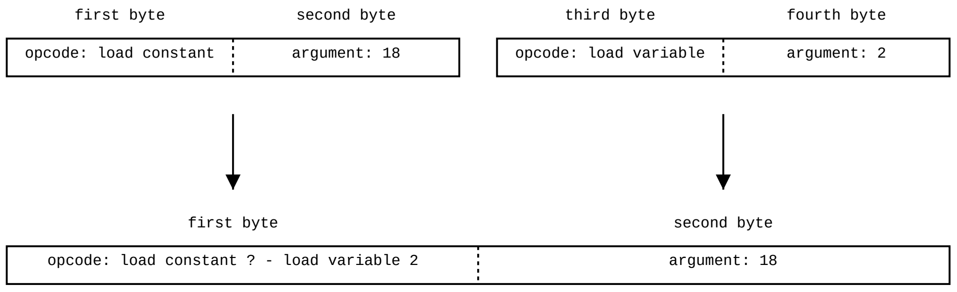
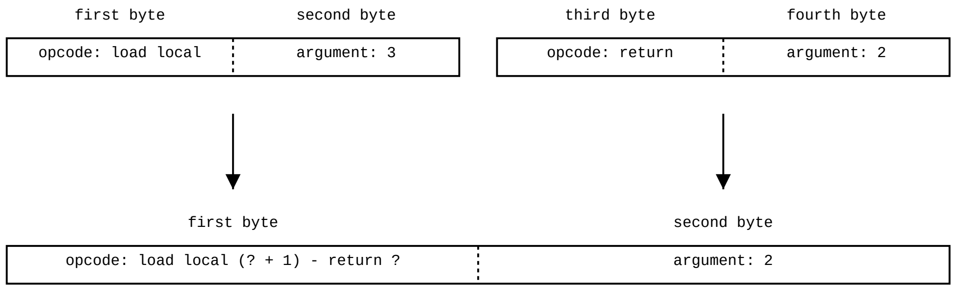


Figure 3.77 Combining `load local` and `return`



Computing the optimal superinstruction set is a very time consuming task. In fact, it is NP-complete [Pro95]. We have implemented an algorithm that given a set of methods and an existing instruction set, computes a single optimal pair superinstruction. Running this algorithm iteratively yields a set of superinstructions that is a reasonable approximation of an optimal set.

We have used this algorithm to generate candidates for new superinstructions, but have found it difficult to automate the process completely and still get high-quality superinstruction sets as output. As with many other things, superinstructions can be optimized for speed or space. So far we have optimized the instruction set to minimize the size of methods, since this could be measured without changing the virtual machine. Optimizing for speed requires instrumenting the virtual machine to record traces of frequently executed instruction pairs.

We have implemented support for superinstructions in our source code compiler. Section 6.1.4 shows the results of our implementation evaluation. Superinstructions reduces the size of methods by 12–15% on the benchmarks we have run. We expect that we can improve our superinstruction set to yield a size reduction of 20–25%. We have not yet implemented the interpretative routines for superinstructions. For that reason, we have been unable to measure the performance impact of superinstructions. We expect superinstructions to yield a considerable speedup, because superinstructions eliminate instruction dispatch between the instructions they cover.

Chapter 4

Software Development

This section gives an overview of our software development platform. It is not intended as an in-depth coverage of the topic. Current development platforms for embedded devices are all very static and do not cope well with dynamic software updates. To solve these problems, we propose a development platform design, which enables full runtime serviceability. The serviceability of our system makes it possible for developers to debug, profile, and update embedded software in operation. This allows for true *incremental* software development on embedded devices; something that has only been possible on a few desktop and server systems until now.

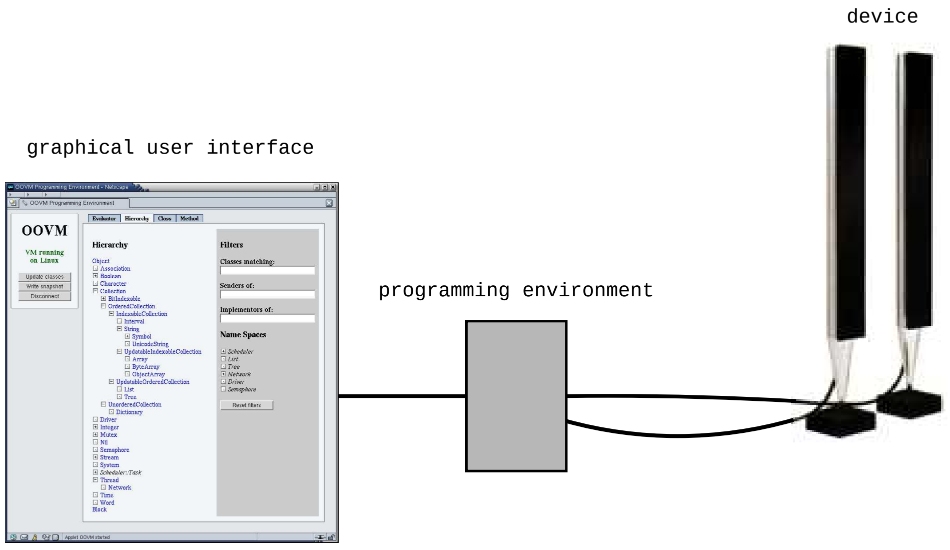
Traditionally, embedded software has been developed in C. The source code is compiled to native code and linked on the development platform, and the resulting binary image is then transferred to the embedded device. Debugging and profiling are normally done by instrumenting the native code or by using *in-circuit emulation* (ICE) hardware. If any part of the source code is updated, everything must be recompiled and relinked to produce a new binary image. Before the source code change is effectuated, the new binary image must be transferred to the device. Software productivity is thus compromised by the cumbersome update process.

In the last few years, the industry has tried to solve some of the traditional problems by introducing Java on embedded devices. Java is a safe object-oriented language, and as such it is a step in the right direction. The industry standard Java virtual machine for embedded devices, KVM, supports remote debugging through the KVM Debug Wire Protocol (KDWP). This means that Java applications running on embedded devices can be inspected and debugged from a development platform. Unfortunately, KDWP support is only included in debug builds of KVM. The debug version of KVM is much slower and several times larger than the optimized version. This makes it impractical to fit a Java implementation with remote debugging support on memory-constrained systems. Consequently, the software productivity benefit of introducing Java on embedded devices is very limited.

4.1 Overview

We have designed and implemented a novel way of developing software for embedded devices. By allowing programmers to connect to running embedded devices and directly manipulate every object, class, or method, we have extended the applicability of truly interactive programming environments to embedded systems. Figure 4.1 shows the components of our system. The speakers are the device running our virtual machine. They are connected to the programming environment via a communication channel. The channel can be based on an ethernet link, an IEEE 1394 (FireWire) link, or an RS232 serial link.

Figure 4.1 Developing software for digital speakers



The programming environment is written in Smalltalk. It includes routines for browsing and updating classes and methods. One of the principal tasks of the programming environment is to compile source code to virtual machine instructions. The instructions are transferred to the virtual machine on the device via a *reflective interface*. The reflective interface is accessed through the communication channel. It defines operations for updating the code that is running on the device, and for inspecting and profiling. It is described in more details in section 4.3.

The graphical user interface of the programming environment is based on HTML, which means that any web browser can be used to interact with the environment. The programming environment acts as an HTTP server, publishing HTML content to the user interface. Section 4.2 shows close-up screenshots of the interface and gives more details on its design and implementation.

Instead of connecting the programming environment to an embedded device, it is also possible to connect it to a virtual machine hosted on an operating system. This is particularly useful during development. We have implemented hosted versions for both embedded Linux and desktop Linux. When hosted on Linux, the communication channel is based on local pipes or network connections. The reflective interface is unchanged between the different versions.

4.2 User Interface

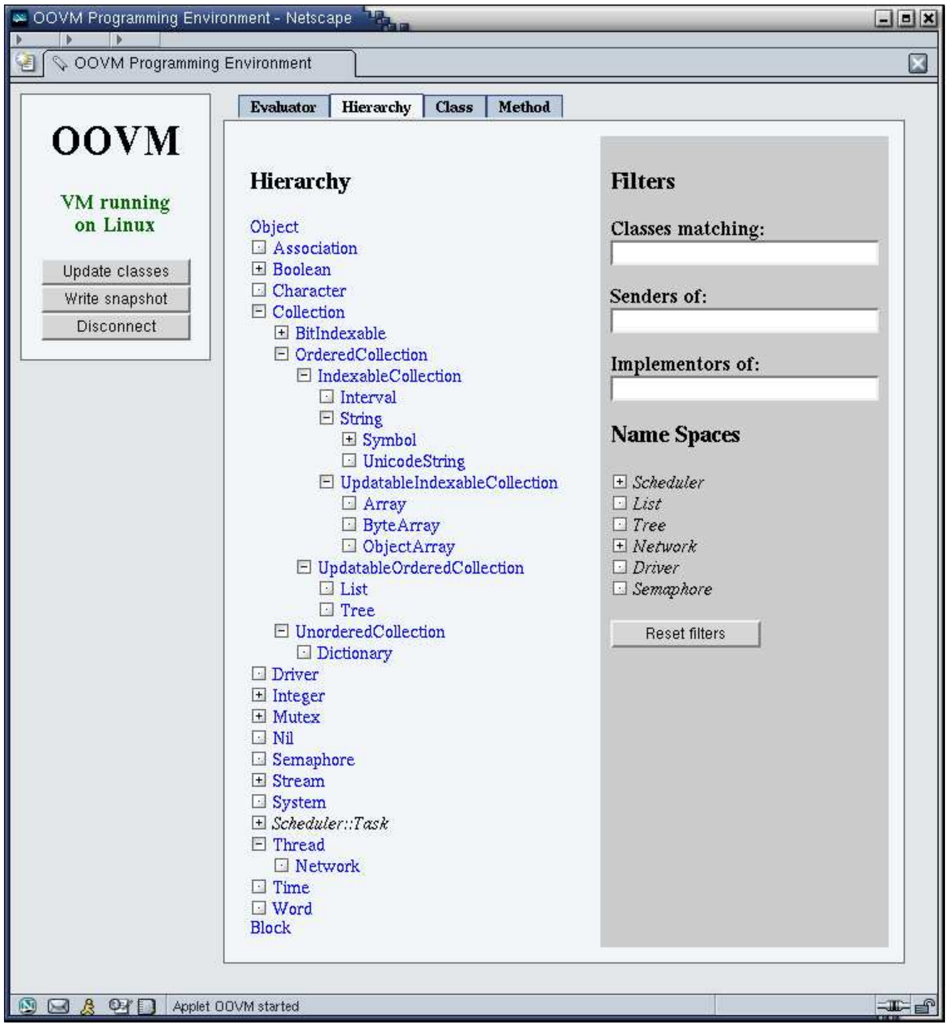
Our graphical user interface runs inside a web browser. Figure 4.2 on the following page shows how the class hierarchy can be browsed using Netscape 7.0. The status bar to the left of the page shows that the programming environment is connected to a virtual machine hosted on Linux. The filters to the right are used to select the classes that appear in the hierarchy tree view in the middle of the page. The filtering shown deselects all classes but the ones in the root namespace.

The individual classes can also be browsed. If the **Interval** link in figure 4.2 is clicked, the user will instantly be redirected to a page for browsing the interval class. Figure 4.3 on page 93 shows the layout of this page. It shows that the interval class defines three instance variables, three instance methods, and two class methods.

Browsing individual methods is shown in figure 4.4 on page 93. If the source code is changed, the programming environment automatically compiles it and transfers the updated method to the device. The update is fully incremental; the device does not have to be stopped or restarted.

The user interface supports evaluating arbitrary Smalltalk expressions. This is useful for experimentation purposes. In figure 4.5 on page 94, the interval implementation is being tested by evaluating an expression that iterates over an interval. The expression is compiled and transferred to the

Figure 4.2 Browsing the class hierarchy



Hierarchy

```
Object
⊡ Association
⊞ Boolean
⊡ Character
⊟ Collection
   ⊞ BitIndexable
   ⊟ OrderedCollection
      ⊟ IndexableCollection
         ⊡ Interval
         ⊟ String
            ⊞ Symbol
            ⊡ UnicodeString
         ⊟ UpdatableIndexableCollection
            ⊡ Array
            ⊡ ByteArray
            ⊡ ObjectArray
      ⊟ UpdatableOrderedCollection
         ⊡ List
         ⊡ Tree
   ⊟ UnorderedCollection
      ⊡ Dictionary
⊡ Driver
⊞ Integer
⊞ Mutex
⊡ Nil
⊡ Semaphore
⊞ Stream
⊡ System
⊞ Scheduler::Task
⊟ Thread
   ⊡ Network
⊡ Time
⊡ Word
Block
```

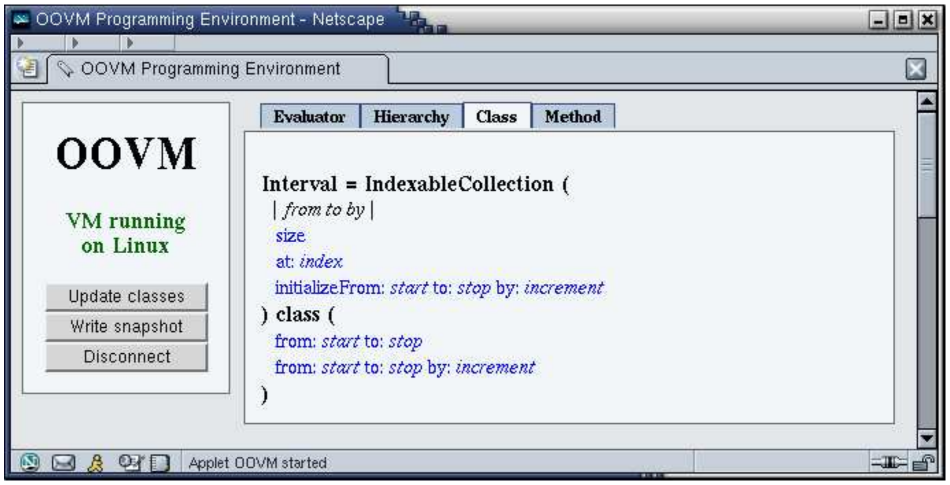
Name Spaces

```
⊞ Scheduler
⊡ List
⊡ Tree
⊞ Network
⊡ Driver
⊡ Semaphore
```

device, where it is executed. The output of the evaluation is redirected to the console shown at the bottom of the web page.

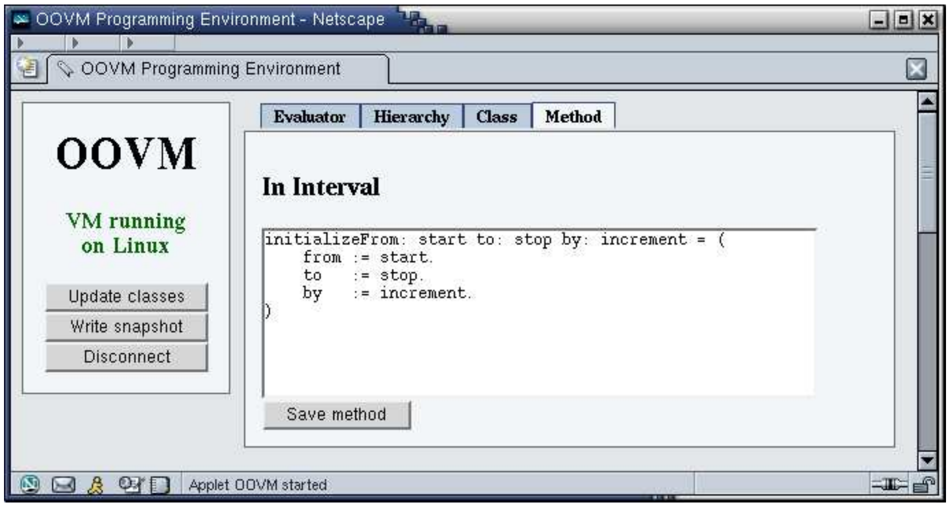
Our programming environment is connected to the web browser via the network and it is capable of updating the web contents dynamically. This makes it possible to *push* information from the programming environment to the user interface. To accomplish this, our implementation relies on dynamic HTML and JavaScript.

Figure 4.3 Browsing the interval class



```
Interval = IndexableCollection (
  | from to by |
  size
  at: index
  initializeFrom: start to: stop by: increment
) class (
  from: start to: stop
  from: start to: stop by: increment
)
```

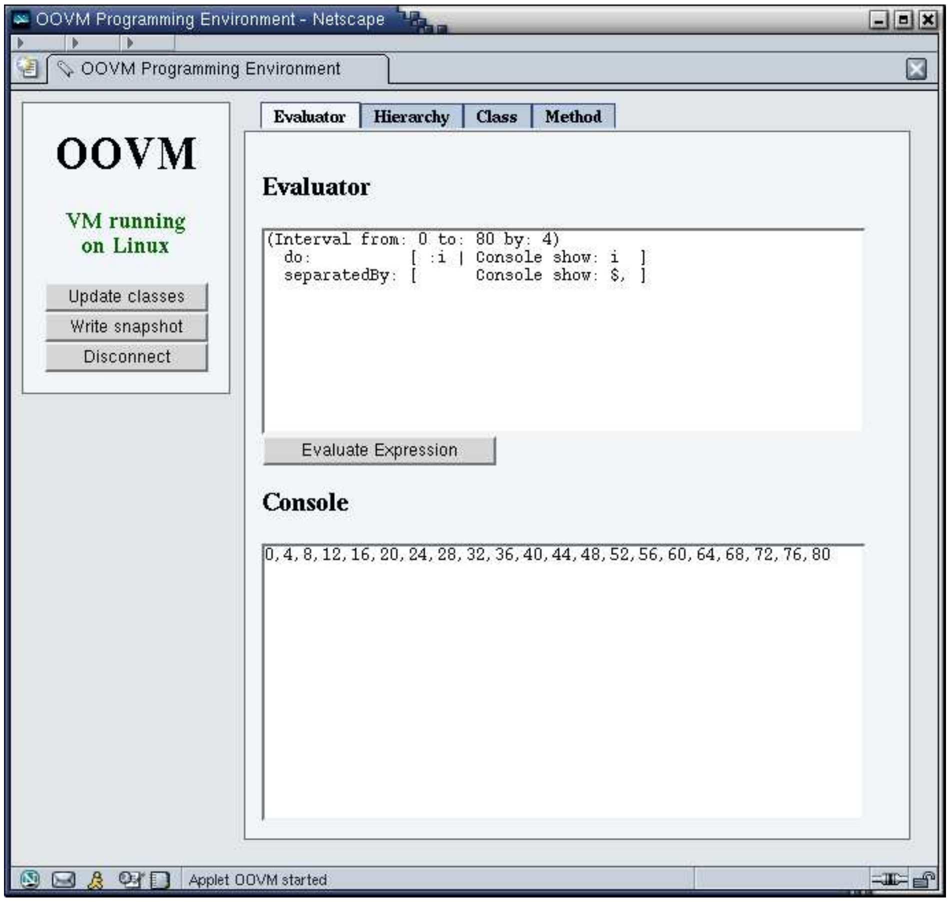
Figure 4.4 Browsing an interval initialization method



In Interval

```
initializeFrom: start to: stop by: increment = (
    from := start.
    to   := stop.
    by   := increment.
)
```

Figure 4.5 Testing the interval implementation



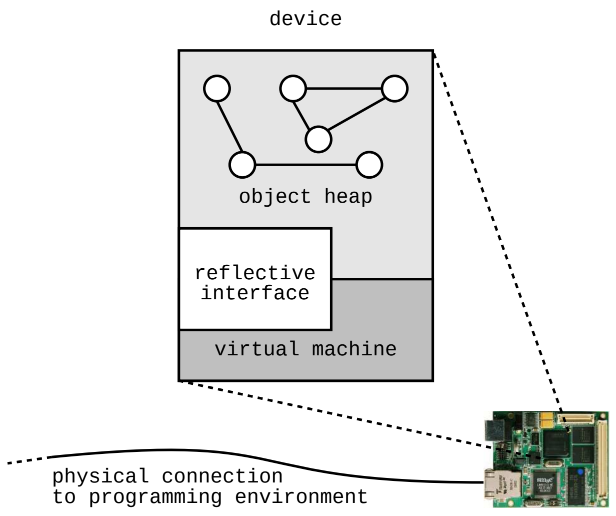
```
(Interval from: 0 to: 80 by: 4)
  do:          [ :i | Console show: i  ]
  separatedBy: [      Console show: $, ]
```

```
0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80
```

4.3 Reflective Interface

Our system provides a reflective interface on the embedded device software platform. The programming environment can connect to it through a physical connection. Using the reflective interface, the programming environment can inspect, update, debug, and profile the running system. The reflective interface consists of a number of primitives in the virtual machine and some Smalltalk code running on top of it, as shown in figure 4.6.

Figure 4.6 Reflective interface on device



Standard Smalltalk allows the running system to reflect on itself. This requires support code for source code compilation on the device. We have minimized the amount of reflective code needed on the device by decoupling the programming environment from the running system.

It is possible to have the entire reflective interface inside the virtual machine. However, this requires implementing communication drivers inside the virtual machine, invariably dragging the whole network stack with it. It also means that the virtual machine itself has to be customized for new communication media and protocols.

We want the reflective interface to use the same communication drivers and network stacks as normal programs, so at least part of the reflective interface has to run on top of the virtual machine. At the same time, part of the reflective interface has to be inside the virtual machine because our programming language has no mechanisms for creating or changing classes or methods.

To some degree, we decide where to draw the red line between code in the virtual machine and code running on top of it. However, complexity travels in packs, and implementing one module inside the virtual machine tends to require implementing other modules inside the virtual machine also. In general, it is desirable to push complexity as far up the software stack as possible, since the higher levels have better abstractions and more control.

In our system, the reflective interface runs as a normal application thread. It receives incoming packets from the communication drivers, and calls virtual machine primitives when needed. The connection between the programming environment and the reflective interface on the device is made using a connection-oriented protocol. Our current implementation uses TCP/IP over an ethernet connection or a serial line, but other physical media and transport protocols can be used, provided they offer a service level similar to that of TCP.

Access to the reflective interface should be restricted, so that the programming environment has to authenticate itself before being allowed access. This is especially true if the reflective interface is exposed to an untrusted domain such as the internet. We have not yet implemented any authentication or encryption on the reflective channel, but both can be implemented using standard techniques and protocols.

4.3.1 Updating

Software updates are often complex and require multiple changes to classes and methods. In order to update software on the device, the programming environment connects to a running system through the reflective interface. Because of the serial nature of the communication link between the programming environment and the reflective interface, changes have to be sent over one at the time. If the changes are applied as soon as they are received by the reflective interface, there is a great risk that the integrity of the running system will be compromised.

There is no problem if the update only adds new classes to the system, since the software already running on the device does not use those classes. However, an update that changes already existing classes will impact running software. Changes made to one class often depends on changes made to other classes, and if they are applied one by one, the system will be in an inconsistent state until all changes are applied. In short, to protect the integrity of the system when updating running software, the updates have to be applied atomically.

In our system, the reflective interface pushes changes onto a stack. Our reflective stack machine has operations to push constants such as integers, symbols, and instructions, operations to create methods or classes using the topmost elements of the stack, and operations that push already existing classes and methods to the stack. The reflective stack is an array that exists in the object heap.

The reflective operations allow the programming environment to build a set of changes by uploading new methods, creating classes, and setting up modifications to existing classes. When the change set has been constructed, it can be applied atomically using a single reflective operation. The nature of the reflective system allows software to run while the change set is being uploaded. The virtual machine only has to suspend execution for a few milliseconds while the changes are being applied.

4.3.2 Debugging

Inspecting the state of objects at runtime is an essential technique for locating problems. In our design, inspecting an object is achieved by pushing a reference to it on the reflector stack. The virtual machine contains reflective primitives for accessing the state of the object through the reference. The primitives are identical to the `load variable` and `store variable` instructions, except that the primitives fetch their arguments from the reflector stack instead of the execution stack. The result is also pushed to the reflector stack. The programming environment is thus able to use the reflective interface to poll the virtual machine for the state of an object.

Another important debugging technique is stepping through running code. Many systems achieve this by patching the running code with *breakpoints*. Whenever a breakpoint is encountered, the running thread is stopped and control is transferred to the programming environment. Breakpoints are typically global; every thread that encounters them is forced to stop. This causes threads of no interest to the debugger to be slowed down, since they have to ignore the breakpoints. In our system, this includes threads providing operating system functionality, and thus a single breakpoint in a frequently used method may cause serious performance degradation for the whole system.

To remedy this, our threads can be in one of two modes: *Normal* or *debug*. The mode can be changed dynamically by the programming environment. Only threads in debug mode are affected by breakpoints. This is achieved by using a sligthly modified interpreter for threads in debug mode. At every send, the modified interpreter checks if the target method contains breakpoints. If so, an unoptimized version of the target method is invoked. The unoptimized version does not contain any superinstructions or inlined control structures. This makes it possible to place explicit breakpoint instructions anywhere in the method, without worrying about superinstructions or branches. Notice that adding breakpoints may require deoptimizing methods on the execution stacks of debug threads.

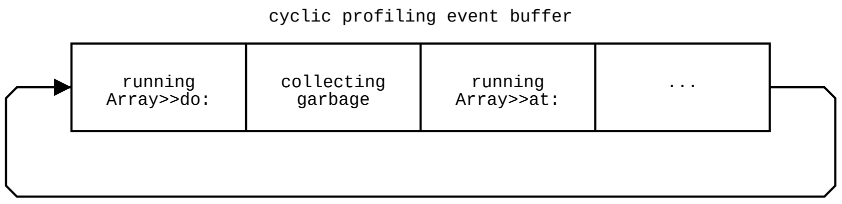
4.3.3 Profiling

For optimization purposes, it is important to know where time is spent. To measure this, a tool known as an *execution time profiler* is used. Many profilers work by instrumenting the code before running it. Every method is instrumented to gather usage statistics. The statistics may include invocation counts and time spent. The GNU Profiler (`gprof`) is an example of a profiler that works by instrumenting code; see [GKM82]. Unfortunately, instrumented code tends to behave differently than the original code. An alternative solution is to use *statistical profiling*. Statistical profiling is based on periodic activity sampling, and it incurs a much lower overhead than instrumentation. Because the profiled code is unchanged, it is also possible to turn profiling on and off at runtime. For these reasons, we have implemented a statistical profiler. Others have used statistical profiling for transparently profiling software in operation; see [ABD$^+$97].

Our statistical profiler gathers runtime execution information by periodic sampling. Every ten milliseconds, the profiler records an event in its event buffer. The event holds information about the current activity of the virtual machine. If the virtual machine is interpreting instructions, a reference to the active method is included in the event. Other virtual machine activities, such as garbage collections, are also included as events. Figure 4.7 on the next page shows the cyclic event buffer we use for profiling.

The contents of the event buffer can be sent to the programming environment using the reflective interface. The programming environment can use the events to provide an ongoing look at virtual machine activity in real-time. As it was the case for debugging, this functionality is *always* available. Even embedded devices running in operation can be profiled remotely through a communication channel.

Figure 4.7 Statistical profiling using cyclic event buffer



4.4 Libraries

Like most programming platforms, we also include basic libraries of data structures and algorithms with our system. Although the high-level language executed by the virtual machine and the primitives provided by it are enough to develop software, the inclusion of basic libraries means developers can start developing software instantly. Although most developers are capable of implementing their own version of the data structures provided by the basic libraries, it is also easy to make small mistakes in the implementation. By providing an official, debugged version of commonly used code, we take responsibility for debugging the library code and free the developers to work on their specific programs.

The basic libraries provide a stable code base consisting of commonly used data structures and algorithms. The basic libraries are not just provided as a convenience to developers, we also the very same libraries to implement the system software. Even though we have chosen to base our language on Smalltalk, we do not include the standard Smalltalk class libraries. The main reason for this is that the Smalltalk class libraries are too complex and contain reflective code. We have no need for most of the reflective code since we have moved reflection to the programming environment.

Besides classes for integers, characters, booleans, and blocks, our set of basic libraries also contains a collection library and support classes for system software. We have also extended blocks with an exception mechanism that allows us to *throw* and *catch* abitrary objects. The example in figure 4.8 on the following page demonstrates throwing, catching, and rethrowing exceptions.

The exception handling code in the example only handles I/O exceptions, all other exceptions are rethrown by sending the `throw` message to the exception object. We use symbols for exceptions because they are con-

Figure 4.8 Throwing, catching, and rethrowing exceptions

```
[ ...
  #IOException throw.
  ...
] catch: [ :e |
  ...
  e ~= #IOException ifTrue: [ e throw ].
  ...
]
```

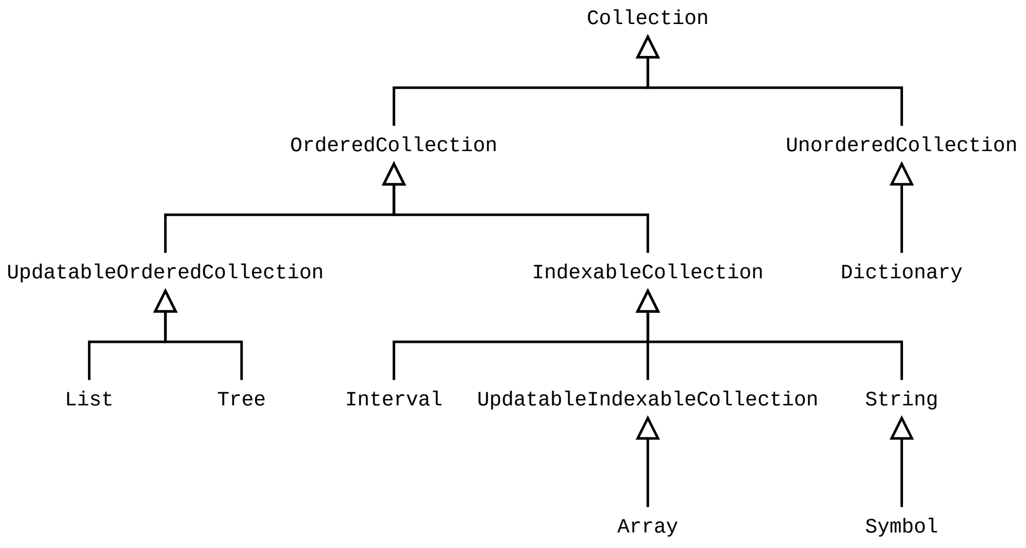
venient, but any object can be thrown. The exception mechanism is implemented using the *unwind-protect* mechanism on blocks. Unwind-protect is a way of protecting the evaluation of a block against unwinds. If the block unwinds due to the execution of a non-local return or the throwing of an exception, the virtual machine notifies the running thread by evaluating a user-defined block.

The operating system support classes include threads, a library of synchronization abstractions, time measurement, and a device driver framework that provides low-level access to hardware. It is described in detail in chapter 5.

Our collection hierarchy consists of basic classes such as `Interval`, `String`, `Array`, `List`, `Tree`, and `Dictionary`. They are organized as shown in figure 4.9 on the next page. The classes are divided into ordered and unordered collections, and further subdivided into indexable and updatable classes.

This collection hierarchy is smaller than the standard Smalltalk collection hierarchy, but we have found it to be sufficient. Compared to standard Smalltalk collection classes, we have made a few changes. The most important and noticeable change is that our arrays are growable; elements can be added to the end of the array using an `add:` method. This way, our `Array` implementation is similar to the `OrderedCollection` implementation in Smalltalk-80.

Figure 4.9 Collection hierarchy



Chapter 5

System Software

It is our claim, that a platform based on the virtual machine we have described can replace traditional operating systems. To substantiate this claim, this chapter describes our implementation of system software in Smalltalk. The purpose of this chapter is not to provide a complete review of our implementation, but rather to demonstrate the ability to implement operating system software. It is also worth noting that our implementation by no means is the only possible implementation. The supervisor mechanism described in section 5.1 closely mimics the interrupt mechanism of computer hardware, so any operating system mechanisms that can be implemented on raw hardware can also be implemented in Smalltalk using our supervisor event mechanism.

5.1 Supervisor

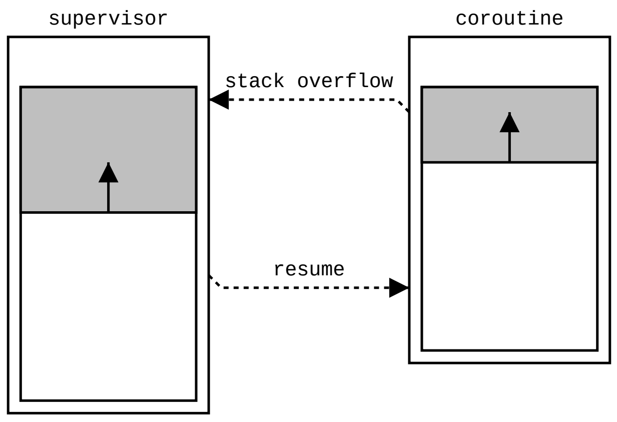
Part of the job of traditional operating systems is to handle events such as hardware interrupts, hardware exceptions, and system calls. Hardware exceptions such as page faults and illegal instructions cannot occur in our system because we do not rely on virtual memory or hardware protection. However, the use of a stack-based virtual machine that executes dynamically typed code adds new types of events, such as lookup errors and stack overflows. To fully benefit from the serviceability of our system, the events must be handled in Smalltalk code running on top of the virtual machine.

We mentioned in section 3.2.2.3 that our execution stacks act as coroutines and can be suspended and resumed. In our system, we have a special *supervisor* coroutine. Whenever an event occurs, the currently active coroutine is suspended and the supervisor coroutine is resumed. When the supervisor is resumed, the event object that caused the resumption is placed on the supervisor's stack so the supervisor knows why it was resumed.

Events can be initiated by the active coroutine, purposely or inadvertently, or by the virtual machine itself. Events that the coroutine initiates on purpose are called *synchronous*, and events that are unexpected by the active coroutine are called *asynchronous*. Stack overflows and lookup errors are examples of asynchronous events inadvertently caused by user code. Synchronous events include exception throwing and explicit yields.

Transferring control to the supervisor when an error occurs allows us to handle exceptions in Smalltalk. This means that we can develop and inspect exception handlers using the same development tools we use to develop application code. Using a supervisor written in Smalltalk also means that the virtual machine itself is smaller, since it does not have to include code for handling exceptions but can pass them onto the supervisor instead. Because coroutines automatically suspend when causing an event, the supervisor has full access to the state of the coroutine at the exact moment the event occurred. The supervisor can thus manipulate the stack of the coroutine, unwinding activations if necessary, and resume the coroutine at will once the event is handled. This is illustrated by figure 5.1.

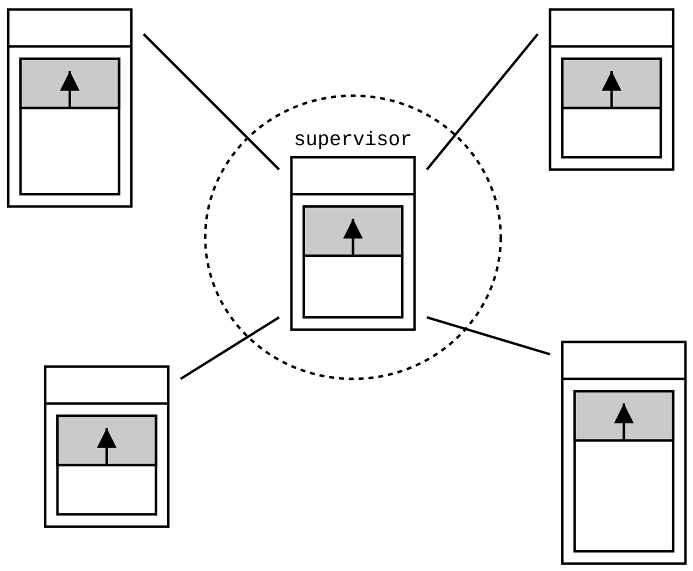
Figure 5.1 Using a supervisor coroutine to handle events



It should be noted that events cannot be initiated while the supervisor coroutine is active. This is not a serious limitation, since initiating an event involves suspending the active coroutine, in this case the supervisor, and resuming the supervisor. In fact, it would be harmful if events could be initiated while the supervisor is active since in addition to suspending and immediately resuming the supervisor, the event object would be pushed on the supervisor stack. To avoid this, we disable events while the supervisor is active. This places some strict demands on the supervisor code. It is easy to avoid synchronous events, and asynchronous hardware events can be disabled when the supervisor is activated. However, asynchronous events triggered by supervisor code because of exceptions, such as lookup errors and stack overflows, must also be avoided. Apart from these restrictions, the only difference between the supervisor and normal coroutines is the fact that the virtual machine knows the supervisor and automatically resumes it when an event occurs.

In essence, the supervisor forms the kernel of our system. It runs on top of the abstracted hardware platform provided by the virtual machine. The supervisor is slightly more privileged than other coroutines because it cannot be interrupted by events. As a result of this some heavier demands are placed upon it. The supervisor forms the trusted code base of our system, but unlike traditional operating systems, we can still debug it using the same tools we use to debug application software. Figure 5.2 illustrates the central role of the supervisor in our system.

Figure 5.2 The role of the supervisor



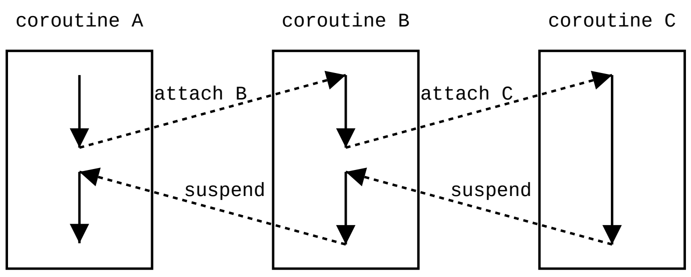
5.2 Coroutines

Although we claim that our execution stacks act as coroutines, our virtual machine does not support the usual coroutine operations as found in for example Simula and BETA. A description of the coroutine support in BETA, as well as several usage examples, can be found in [MMPN93].

In Simula, coroutines use the `resume` operation to suspend the active coroutine and resume a named coroutine. When a coroutine is resumed, execution continues at the point where it was last suspended. The `resume` operation is *symmetrical*, because a coroutine that wishes to suspend itself has to name the next coroutine to run explicitly.

Simula and BETA also provide the *asymmetrical* operation `attach` and `suspend`. The `attach` operation does the same as `resume`, but also remembers which coroutine caused the attached coroutine to be resumed. The `suspend` operation uses this information to resume the coroutine that attached the currently active coroutine, as shown in figure 5.3. These operations are asymmetrical because the coroutine to be resumed is named explicitly in `attach`, but not in `suspend`.

Figure 5.3 Coroutine `attach` and `suspend` operations



The asymmetry of the `attach` and `suspend` operations means that the attached coroutine can return control to its caller without knowing in advance who called it. This allows for a more flexible software design, and as shown in [MMPN93], it is possible to implement `resume` using `attach` and `suspend`.

Our system does not directly support any of the coroutine operations, but they can easily be implemented. The `resume` operation can be implemented by creating a synchronous event requesting the supervisor to resume a named coroutine. An implementation of `attach` and `suspend` based on the `resume` event mechanism is shown in figure 5.4 on the next page.

Figure 5.4 Implementation of `attach` and `suspend` based on `resume`

```
Coroutine = (
  | link |

  attach: coroutine = (
    coroutine link: self.
    Supervisor resume: coroutine.
  )

  suspend = (
    | caller |

    caller := link.
    link := nil.
    Supervisor resume: caller.
  )

  link: coroutine = ( link := coroutine )
)
```

We have chosen to implement a thread-based system, rather than base our system on coroutines. In our system, a thread contains an execution stack as well as various other state, including whether the thread is ready to run, and if not what it is waiting for. However, the flexibility of our system allows others to implement and use a complete coroutine library instead of, or along with, the thread library.

5.3 Scheduling

The first issue that must be addressed in a system supporting multiple threads of execution is deciding which thread to run. The scheduling problem is largely a policy problem, since the necessary mechanisms can be as simple as being able to atomically suspend one thread and resume another. Deciding on a scheduling policy is not an easy task. Significant research has gone into finding optimal scheduling policies for various systems, and the only conclusion so far seems to be that different application domains have different scheduling needs. In other words, if a system has a fixed scheduling policy, it must be very generic to support executing as many application types as possible.

There are several popular scheduling strategies, each with different properties. The simplest strategy is to schedule threads according to a fixed cyclic order. This is known as *round robin* scheduling. Round robin scheduling ensures that all threads get an equal opportunity to run. Often, however, the fixed fairness of round robin scheduling is not desirable. For interactive systems, it is desirable to give more processor time to the threads currently interacting with the user to make the system seem more responsive. This leads to a scheduling policy based on *priorities*. In a priority-based system, the thread with the highest priority level is allowed to run. If there are several threads with equal priority, these can be scheduled using round robin scheduling. To ensure that all ready threads eventually are allowed to run, the scheduling algorithm must sometimes run threads with lower priority to avoid starvation. One way of doing this is to temporarily boost the priority of threads that have not been allowed to run for some time. If the priority boost is proportional to the time since the thread last ran, then the thread will eventually have the highest priority, and be allowed to run. Once the thread has been allowed to run, its priority level returns to normal.

A priority-based real-time scheduler also has to guard against priority inversion. Consider the following scenario: A low priority data analysis thread is holding the mutex on a shared bus, which a high priority data distribution thread is waiting for. In this scenario, a medium priority communications thread can preempt the low priority thread and thus prevent the high priority thread from running, possibly causing the high priority thread to miss a real-time deadline. This happened on the Mars Pathfinder while it was performing its duties on Mars. The JPL engineers at NASA were able to fix it by uploading a code patch to the Pathfinder using mechanisms they had developed themselves [Ree97].

An unusual, but quite efficient, variant of priority scheduling is *lottery scheduling* [WW94]. In a lottery-based system, each thread is given a number of lottery tickets. The scheduler selects the thread to run by drawing a ticket at random. The thread holding the ticket is allowed to run. Lottery scheduling supports priorities by allowing a thread to have more than one ticket, and thus an increased chance of being allowed to run. Lottery scheduling turns to probability theory to ensure that all ready threads are allowed to run at some point in time.

In addition to deciding on a scheduling algorithm, a systems designer must also decide when to exchange the currently running thread for another. There are two ways of doing this: *Cooperative scheduling* and *preemptive scheduling*. Cooperative scheduling allows the threads themselves to decide when to run the next thread, by having the currently running thread actively *yield* control of the processor at sufficient intervals. Many operating system operations, such as synchronization and I/O, will yield control if necessary. Thus, as long as the applications invoke operating system operations regularly, the developer does not have to insert explicit yields in the program code. For interactive systems, this may not be enough as it allows one thread to assert control of the processor indefinitely. Preemptive scheduling remedies this by allowing a thread to be preempted by the scheduler. In a preemptive system, the scheduler is typically invoked periodically, preempting the currently active thread.

5.3.1 Cooperative Scheduling

Cooperative scheduling has been used in many operating systems. Microsoft Windows used cooperative scheduling until Windows 95. Apple® only recently switched to preemptive scheduling with Mac OS® X.

Because our virtual machine supports coroutines as a basic primitive, and uses the supervisor coroutine for handling events, it is easy to build a system based on cooperative scheduling. When the active thread wants to yield control of the processor, it can transfer a `yield` event to the supervisor. The supervisor then calls the scheduling algorithm to select the next thread to run, and resumes its coroutine. The code for a simple supervisor supporting cooperative scheduling is shown in figure 5.5.

Figure 5.5 Simple scheduler implementation

```
[ | active event |

  "The list of waiting threads is sorted according to wake-up time"

  (Scheduler::waiting first isReady) whileTrue: [
    Scheduler::running add: (Scheduler::waiting removeFirst)
  ].

  active := Scheduler::running nextIfEmpty: [ Scheduler::idle ].

  event := active resume.
  event handle.

] repeat.
```

This simple supervisor always selects a new thread to run after handling an event. As the code in figure 5.5 illustrates, events in our system are normal objects and can therefore contain code. This allows for a more modular supervisor design, since the supervisor does not have to check the event to find out how to handle it, but can simply ask the event to handle itself.

5.3.2 Preemptive Scheduling

Cooperative scheduling is adequate for many purposes, but for true multi-threading and real-time guarantees, we have to support preemptive scheduling. By introducing a new asynchronous event that is triggered by the virtual machine whenever the system timer ticks, the active thread will be suspended and the supervisor resumed on every timer tick.

At the hardware level, interrupts are handled between execution of any two native instructions. A similar property holds for events in our system. Threads can only be suspended between instructions, not in the middle of interpreting an instruction or while inside a native function. We use the stack overflow checks in our system to handle events. The hardware interrupt handler in our system sets the current stack limit artifically low, forcing the next stack overflow check to fail. The stack overflow checking code in the virtual machine recognizes the artificial limit, concludes that a hardware interrupt has occured, and initiates the interrupt event. Stack overflow checks are made at every send, so a thread executing a tight loop without any sends can prevent the system from being interrupted. To address this, we make a stack overflow check on backward branches as well, even though this is not necessary for the execution of the code. Handling interrupts this way means that individual instructions are executed atomically by the virtual machine, and that threads have a well-defined state when suspended.

When the supervisor is resumed because of a timer tick event, it can select the next thread and transfer control to it as if the active thread had yielded control manually. The timer tick event is an example of an asynchronous event initiated by the virtual machine. Unlike the synchronous yield event used in cooperative scheduling, the currently active thread is not aware that it has been preempted by the timer tick event since this happens asynchronously.

5.4 Synchronization

We discussed synchronization as a language requirement in section 3.1.1.4 on page 25, using Java as an example. In Java, all objects can be synchronization targets and synchronization is implemented using the `monitorenter` and `monitorexit` bytecodes. This approach means that synchronization has to be implemented in the virtual machine, adding to the complexity of the virtual machine code.

We have minimized the amount of virtual machine code, by implementing synchronization on top of the virtual machine rather than in it. Because our system is based on preemptive scheduling, we have to introduce a critical section mechanism so we can manipulate critical data structures without risking preemption. We do not, however, want to enable user threads to create critical sections at will, since this would defeat the purpose of preemptive scheduling.

Our system already contains a coroutine that is never preempted: The supervisor. Using this, synchronization can be implemented by triggering an event representing a request to lock a specified mutex object. The supervisor then examines and manipulates the fields of the mutex object and transfers control back to the calling thread if the lock is uncontended. In case of contention, it inserts the thread into a queue and transfers control to another thread.

This scheme bears some resemblance to the traditional operating system model where control is transferred to the kernel for critical operations. Although our control transfers are not as expensive as the hardware-enforced context switches used in traditional operating systems, there is still a cost associated with them. In case of contention, the small control transfer overhead is not important since the thread will be blocked anyway, but for uncontended locking it is desirable to eliminate the control transfer.

A simple way to eliminate it, is to introduce an atomic `test-and-store variable` virtual machine instruction and use it to lock objects optimistically, only falling back to the supervisor in case of contention. The instruction is similar to `cmpxchg` (compare-and-exchange), `bts` (bit-test-and-set), and equivalent instructions found in modern microprocessors.

`test-and-store variable`

The instruction argument is used as a variable index into an object popped from the stack. The variables are indexed from one. Another element is popped from the stack. If the variable and the popped element refer to the same object, then the variable is overwritten with a third element popped from the stack, and `true` is pushed onto the stack. If the variable refers to another object than the popped element, then the third element is still popped from the stack, but the variable is not overwritten, and `false` is pushed onto the stack. Like all instructions, all of this is done atomically.

The syntax chosen for the test-and-store statement is shown in figure 5.6. When evaluating this statement, `assign-expression`, `test-expression`, and `self` are pushed onto the stack in this order, and the `test-and-store variable` instruction is executed with the index of `variable` as its argument.

Figure 5.6 Syntax for atomic test-and-store statement

```
<variable> ? <test-expression> := <assign-expression>
```

We can use this statement to implement fast synchronization because instructions are executed atomically. An example of a mutex implementation using the test-and-store statement is shown in figure 5.7. A mutex object has an `owner` field which points to the thread currently owning the mutex, or `nil` if it is unlocked. We can use the test-and-store statement to simultaneously check if the mutex is unlocked and lock it if it is. As long as the test-and-store expression evaluates to `false`, ie. the owner field was not `nil`, the current thread yields control of the processor. When the thread is resumed, it will retry the locking. In the uncontended case, the use of the test-and-store instruction allows us to lock the mutex without transferring control to the supervisor.

Figure 5.7 Mutex implementation using atomic test-and-store

```
Mutex = Object (
  | owner |

  do: [block] = (
    [ owner ? nil := Thread current ] whileFalse: [ Thread yield ].
    block value.
    owner := nil.
  )
)
```

This is just an example to demonstrate the test-and-store statement. The described busy-wait loop is impractical, not only because of the time spent repeatedly checking the mutex, but also because there can be no guarantees for fairness and no control over the time elapsed between the owning thread unlocking a mutex and a contending thread locking it.

While it is possible to let the supervisor handle the contended case to ensure fairness, a better solution is to implement mutexes using a more general concurrency primitive such as a *semaphore*. In fact, most concurrency data structures can be implemented using semaphores [Mad93]. By building higher-level abstractions using semaphores, we minimize the amount of supervisor code. Like the code in the virtual machine, it is also beneficial to minimize the amount of supervisor code, because of the extra rules imposed on it. Because all other abstractions are based on semaphores, we only have to implement fairness and time-constraints on the semaphore code.

Figure 5.8 shows a semaphore implementation that uses the atomic test-and-store statement. Again, the idea is to minimize the number of control transfers by updating the count optimistically. The atomic test-and-store statement ensures that no other threads have changed the count while the current thread is calculating the new value. If there is any chance that the thread would be blocked in case of locking, or if there are waiting threads in case of unlocking, control is transferred to the supervisor which will handle the entire semaphore operation, and block or unblock threads as needed.

Figure 5.8 Acquiring and releasing semaphores using test-and-store

```
Semaphore = Object (
  | count |

  acquire = (
    [ | c |
      c := count - 1.
      c < 0 ifTrue: [ ^Supervisor acquire: self ].
      count ? c + 1 := c
    ] whileFalse.
  )

  release = (
    [ | c |
      c := count + 1.
      c < 1 ifTrue: [ ^Supervisor release: self ].
      count ? c - 1 := c
    ] whileFalse.
  )
)
```

The loop is needed to ensure that another thread does not update the count variable between the point where we have read the old value and the point where we store the new value. There is only a tiny window of opportunity for this to happen, so the loop will only be re-executed in a few cases. To avoid the theoretical possibility of two threads fighting over who gets to update the semaphore count, a retry counter can be introduced. When then retry limit for a thread is exceeded, the thread will call upon the supervisor to arbitrate the update.

We have measured the effect of the test-and-store instruction. In the uncontended case, the semaphore implementation that uses the test-and-store instruction for optimistic locking is 13 times faster than a version that always transfers control to the supervisor.

5.5 Device Drivers

System software must provide access to the hardware. If only one application has to have access to a particular hardware device, it can be allowed to interface directly with the hardware. If several applications need to use a single device, the system software must provide some mechanism for ensuring that the applications do not interfere with each other. Part of the illusion when running several applications at the same time is that every application thinks it has the undivided attention of all the hardware in the system.

Rather than letting every program access the actual hardware directly, an abstraction layer called a *device driver* is introduced. A device driver handles all the gory details of interfacing with a specific hardware device, ensures that several applications can access the hardware safely, and provides a cleaner interface to higher-level software.

The interface offered by a driver is often unified with other devices of similar type, whereas the actual implementation of the interface is specific to the actual hardware. This abstraction level makes it possible to implement, for example, a network stack that is capable of using several different network devices without modification, as long as the network device drivers provide a common interface.

Device drivers run on top of the virtual machine along with all the other software, so we can debug, profile, and generally inspect it using the same mechanisms used for normal programs. To facilitate this, the virtual machine must provide mechanisms for accessing hardware from software running on top of it. By restricting access to those mechanisms, we can prevent programs from accessing hardware directly, and ensure that drivers only can access their own hardware. It should be noted that the safety properties of the virtual machine and the controlled hardware access only prevent drivers and programs from interfering with each other. A faulty driver controlling a vital hardware device can still cause the system to malfunction, since the virtual machine has no control over what the driver actually does with its hardware device.

5.5.1 Input/Output

Hardware devices can be accessed by reading from or writing to addresses within the address space of the device. Some processors provide an alternate address space for devices which must be accessed using special I/O instructions, but the most common method is to map device address spaces as part of the system memory space. This allows devices to be accessed using normal memory access instructions.

In our system, we provide access to memory-mapped I/O through external memory proxy objects. A simplified driver for the general-purpose I/O module (GPIO) of the Intel® StrongARM processor is shown in figure 5.9.

Figure 5.9 Sample driver for Intel StrongARM GPIO module

```
GPIO = Object (
  | io |

  initialize = (
    io := ExternalMemory at: 16r90040000 length: 32.
   )

  setOutput: pins   = (
    io at: 16r08 put: pins.
  )

  clearOutput: pins = (
    io at: 16r0C put: pins.
  )
)
```

As figure 5.9 shows, the driver requests a proxy object that represents the memory address space of the device. Proxy objects are instances of `ExternalMemory`. To the driver, the proxy object looks like an array of bytes that can be accessed like any other byte array. When the driver sends an `at:` or an `at:put:` message to the proxy, the corresponding virtual machine primitive checks that the requested address is within the bounds of the proxy, and then reads from or writes to the requested address. Obviously, the virtual machine has to make sure that a driver cannot allocate a proxy that refers to the object heap, since that would allow the driver to corrupt the heap and thus circumvent the pointer safety of our system.

5.5.2 Interrupts

Input and output operations are always initiated by software. If a hardware device needs attention, it can issue an *interrupt request* to gain the attention of the processor. When the processor sees the interrupt request, it calls the appropriate interrupt handler based on the interrupt request ID of the device.

In traditional operating systems, the interrupt handler is part of the device driver and exists inside the operating system. Since it is invoked directly by the processor, it has to be written in unsafe native code. In our system, device drivers are written entirely in safe Smalltalk code, so we need a mechanism for converting hardware interrupt requests into something we can use from Smalltalk. Since we already have a mechanism to handle asynchronous events, the obvious solution is to create a hardware interrupt event and let the supervisor decide what to do with it.

We have chosen to reify interrupt requests as signal objects allocated by the supervisor. When the supervisor receives a hardware interrupt event, it finds the corresponding interrupt request signal object and signals it, as shown in figure 5.10. A typical device driver contains a loop, which waits for the interrupt request signal associated with the device it is handling, and attends to the device when the signal is raised, as shown in figure 5.11 on the facing page.

Figure 5.10 Supervisor code for handling hardware interrupts

```
handleHardwareInterrupt: interruptID = (
  | signal |

  signal := self findSignalForIRQ: interruptID.
  signal notifyAll.
)
```

Using signal objects is a very flexible way to handle interrupts, but it adds latency to the interrupt handling. An alternative is to let the interrupt request events themselves contain driver code to handle the interrupt, but this is only necessary for critical devices with strict real-time demands.

Figure 5.11 Interrupt-driven driver for Intel StrongARM real-time clock

```
RealTimeClock = Thread (
  ...

  run = (
    [
      (Supervisor findSignalForIRQ: 30) wait.
      self handleTimerInterrupt.
    ] repeat.
  )
)
```

5.6 Networking

This section examines an implementation of a TCP/IP network stack written in Smalltalk. We do not intend to describe the complete implementation, but rather provide an example of object-oriented operating system code. For a complete description of the protocols and design issues of a modern TCP/IP network stack, see [Ste94].

We have implemented a network stack supporting TCP on top of IP. The implementation is based on information found in the relevant RFCs [Pos81a, Pos81b], and it is inspired by the 4.4BSD-Lite networking code. The BSD network stack is the reference implementation of TCP/IP networking. It is used in many operating systems due to its clean and highly optimized design. The BSD network stack implementation is described in detail in [WS95].

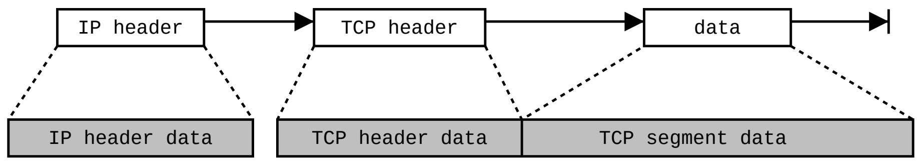
5.6.1 Memory Buffers

When sending data, most layers in the network protocol stack add layer-specific header information to the data packet. When receiving data, such header information is interpreted by the relevant protocol layer, and stripped from the data packet before passing it on to the next layer. A naive implementation would simply copy data between layers, but since data copying is expensive, effort is well spent on minimizing the amount of copying.

To minimize data copying, the BSD stack introduces *memory buffers*. BSD memory buffers contain 128 bytes, but 20 of these are reserved for internal buffer data, leaving 108 bytes for packet data. Buffers can be linked to form larger network packets. Protocol layers encapsulate data packets from higher layers by adding a buffer with the header data to the front of the buffer chain. If a protocol layer needs more than 108 bytes, there is a special kind of memory buffer that stores the data in a 2048 byte external buffer, instead of inside the memory buffer. To relieve the pressure on the memory system, buffers are placed on a free-list when deallocated, to be recycled. There is also a limit on the number of memory buffers that can be allocated, to ensure that the network stack does not use too much memory.

We have implemented a mechanism similar to the BSD memory buffers, with the exceptions that our buffers always store the data in an variably sized byte array. We allow several buffers to refer to different areas of the same byte array, as shown in figure 5.12. We do not currently impose a limit on the number of buffers the network stack is allowed to allocate, since we expect this to be handled by a future per-application resource management system as mentioned in section 5.7.

Figure 5.12 Chained packet buffers



Our buffer design is more flexible than the BSD memory buffers because our buffers have variable sizes. The reason for the fixed size of the BSD memory buffers is that they have to be reused to avoid a severe allocation and deallocation overhead due to the use of `malloc` and `free`. We have fast object allocation and efficient, automatic resource management, so we can afford to allocate objects when they are needed.

Our design means that the network device driver can store an entire incoming packet in one buffer of exactly the right size. When the packet is passed up the protocol stack, the individual protocol layers can split the packet into a header part and a data part, interpret the header, and pass the encapsulated data to the next protocol layer without having to copy any data.

Another advantage due to the object-oriented nature of our system, is that the individual protocol layers can specialize the buffers to provide accessors for protocol-specific header data. Figure 5.13 on the facing page shows some of the accessors for TCP packet headers.

When sending data, we can encapsulate data just as easily as BSD by allocating a new buffer for the header and prepending it to the data buffer

Figure 5.13 TCP packet header accessors

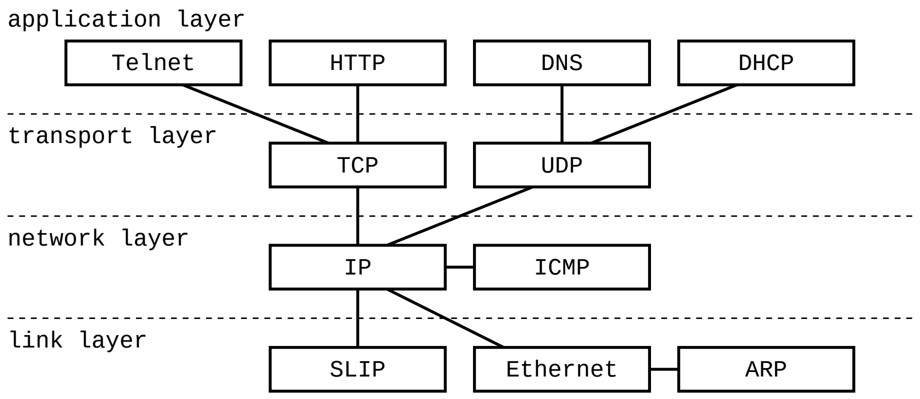
```
Packet = Network::Packet (
  sourcePort      = ( ^self shortAt: sourcePortOffset      )
  destinationPort = ( ^self shortAt: destinationPortOffset )
  sequenceNumber  = ( ^self longAt:  sequenceNumberOffset  )

  ...
)
```

chain. Unlike BSD, there is no internal fragmentation because the individual protocol layers allocate exactly the right size buffer for their header data.

5.6.2 Protocol Layers

In TCP/IP, the session and presentation layers of the Open Systems Interconnection (OSI) network model [ISO84] are included in the application layer. At the bottom of the OSI stack is the physical layer which we will ignore in this section, since it is implemented entirely in hardware. Figure 5.14 shows the remaining four layers along with some of the protocols in them.

Figure 5.14 Layers in a TCP/IP protocol stack



At the bottom of our stack is the link layer which is responsible for sending and receiving packets on the physical network media. When an incoming packet arrives, the link layer dispatches it to the relevant protocol in the datagram layer, as shown in figure 5.15. The link layer is a point-to-point layer, and can thus only send packets to devices connected to the physical media.

Figure 5.15 Demultiplexing incoming network packets

```
handle: incoming = ( | protocol |
  protocol := protocols at: incoming type ifAbsent: [ ^nil ].
  protocol handle: incoming.
)
```

The datagram layer handles incoming packets in a way similar to the link layer, by dispatching them to the relevant protocol in the transport layer. The handling of outgoing packets is more complex, though, because the datagram layer is responsible for routing packets. The datagram layer is where the IP protocol resides, and given an IP address, it is possible to send packets to any computer connected to the network. The datagram layer is responsible for determining how to get the datagram to the remote computer. Many devices have only one gateway to the internet, so for them routing is a simple task. If a device has more than one way to send data to the internet, the IP protocol has to decide which gateway to use. Like the link layer, the datagram layer is connection-less, and decides routing on a datagram to datagram basis.

The transport layer is the most complex layer. It provides an interface to the application layer, and in the case of connection-oriented protocols like the *Transport Control Protocol* (TCP), it also guarantees that sent data is received by the remote device, retransmitting datagrams if necessary. TCP is by far the most advanced component of the TCP/IP network stack, and our implementation of it is described in some detail in the next section.

The final layer of the network stack is the application layer. This is where the software that actually uses the network resides. We will not focus on this layer.

5.6.3 TCP Protocol

The TCP protocol handles incoming datagrams and user requests using a finite state machine. In the BSD implementation, state is recorded using an integer field which the TCP event handler routines check to decide which action to take. We have chosen to represent state using objects. Each TCP state object knows how to handle events occurring in its state.

The function responsible for handling incoming TCP segments in the BSD implementation is over 1000 lines of C code. The segment processing is divided into several steps, and many of the steps explicitly examine the state to decide how to process the segment. In our implementation, the state is implicit since the event handler is defined in the state object and thus knows what state the connection is in. Representing state as objects also allows us to share code between states through inheritance. We have developed the state object hierarchy shown in figure 5.16 to share as much code as possible both for incoming segments and user requests. The shaded classes correspond to the TCP states defined in [Pos81b].

Figure 5.16 TCP state hierarchy

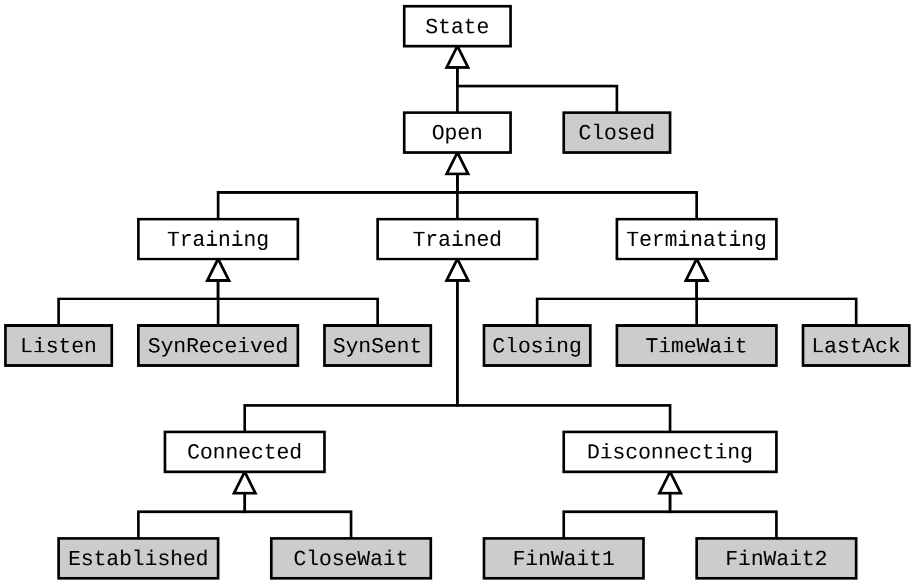


Figure 5.17 on the following page shows the generic state handler for all `Open` states, except for `Listen` and `SynSent`. The code fragments presented here do not include enhancements such as header prediction and round-trip time estimation. While such enhancements are important for TCP performance, they do not impact the overall modelling of the TCP state machine, and have been left out for clarity. The sequence numbers in the comments refer to the segment processing steps listed in [Pos81b].

Figure 5.17 Generic state handler for incoming TCP segments

```
handle: packet = (
  "First, check the sequence number"
  self checkSequenceNumberOn: packet ifFail: [ ^false ].

  "Second, check the RST bit"
  packet isReset ifTrue: [ self processResetOn: packet. ^false ].

  ...

  "Seventh, process the segment text"
  packet hasText ifTrue: [ self processTextOn: packet ].

  ...
)
```

The actual processing of the incoming segment is split into several methods to allow states to specialize individual steps. For example, before a connection has completed the three-way handshake phase, we choose to ignore incoming segment text, so the segment text handler in the generic `Open` state, shown in figure 5.18, does nothing.

Figure 5.18 Generic processing of incoming segment text

```
processTextOn: packet = (
  ^true
)
```

The segment text handler is specialized in the `Trained` state, which we enter when the connection is acknowledged by both computers. In the `Trained` state, we handle incoming text by adding it to the read buffer of the connection. The specialized text handler is shown in figure 5.19.

Figure 5.19 Processing incoming segment text in `Trained` states

```
processTextOn: packet = (
  self receiveNext: (packet sequenceNumber + packet segmentLength).
  self receiveWindow: self readBufferCapacity.

  self addToReadBuffer: packet data.
  self sendAcknowledgement.
)
```

Splitting the steps into individual methods means that the state variable is checked for each of those steps, like it was in the BSD implementation. However, in our implementation, the check is implicit and relies on dynamic dispatching, which is optimized by the virtual machine.

5.7 Summary

In general, it is desirable to push policies and complexity as far up the software stack as possible. Each layer in the software stack adds abstractions. Structures that are complex to implement at the lower levels, often become easier to implement using the abstractions provided by higher levels.

Our design keeps the virtual machine simple, and gives developers freedom to implement their own solutions. The abstraction level provided by the virtual machine resembles that provided by the physical hardware, thus making it possible to implement virtually all the policies and mechanisms that can be implemented on raw hardware. Our system also allows different abstractions, such as coroutines and threads, to co-exist on the same machine.

We have not yet implemented any limitations on the use of resources. Without resource limitation, a single thread can allocate all the available memory in the system, leaving nothing for the other threads or the system software itself. For that reason, it is desirable to be able to limit the amount of memory and other resources that a thread or group of threads can allocate. Limiting resources on a per-application basis also ensures that the network stack does not use all available system resources, for example during a denial-of-service attack.

In this chapter, we have shown parts of an operating system implementation based on our virtual machine design. The use of an object-oriented language makes it possible to write clean operating system code that is easy to understand. Furthermore, our design allows us to debug and profile operating system code as if it were normal application code, which is a major advantage over current system platforms.

Chapter 6

Experiments and Evaluations

We have evaluated our virtual machine by measuring the memory footprint of the object model and the performance of the execution model using a set of micro-benchmarks and two widely used benchmarks: Richards and DeltaBlue. The benchmarks are described in appendix B. We use the benchmarks to compare our system to other, similar virtual machines. Both memory footprint and execution performance are important for embedded systems, and we will show that our virtual machine outperforms similar virtual machines by using less memory and executing code faster.

6.1 Object Model

In this section, we evaluate the characteristics of our object model in terms of memory usage. We show that our object model requires less memory than that of similar virtual machines for classes, methods, and strings, collectively referred to as *reflective data*. We have chosen to compare our virtual machine to Squeak and KVM, by measuring the amount of memory used for reflective data and execution stacks.

Squeak (3.2-4) is an open-source Smalltalk-80 implementation. The virtual machine is written in Slang, a functional subset of Smalltalk, that can be translated into C. Squeak has not been developed with embedded systems in mind, so we expect it to use more memory than the other systems. In order to compare our virtual machine to another Smalltalk virtual machine, we have chosen to include Squeak anyway.

KVM (1.0.4) is a Java implementation for embedded systems from Sun. It is designed from the ground up with the constraints of inexpensive mobile devices in mind. Since KVM is designed for mobile devices, we expect it to perform well memory-wise. KVM is an industry standard, currently used on a variety of embedded devices, including mobile phones.

6.1.1 Overall

We have measured the memory usage for each of the benchmarks by instrumenting the source code of the three virtual machines to gather allocation statistics. The results are shown in figure 6.1. Our virtual machine uses the smallest amount of memory of the three. We use roughly half the space KVM uses, and 35–45% less memory than Squeak. Contrary to our expectations, KVM uses more memory than Squeak. Section 6.1.4 shows that this is due to the representation of methods.

Figure 6.1 Total memory usage for reflective data

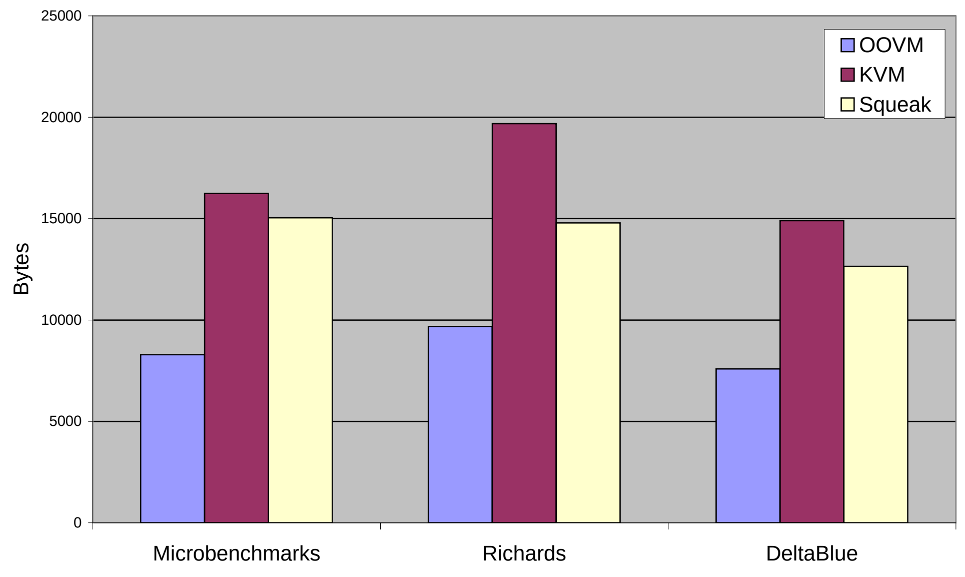
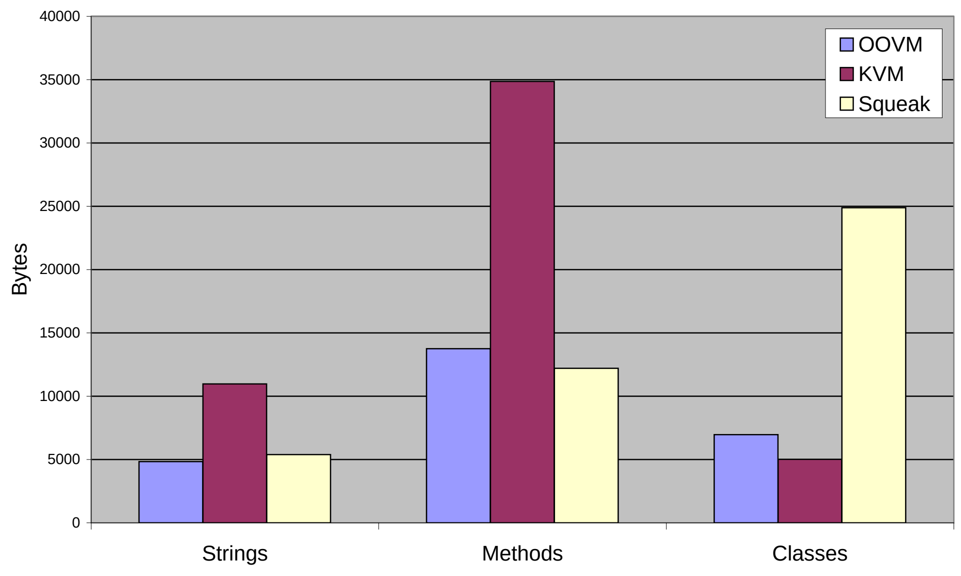


Figure 6.2 on the facing page shows the memory used by the reflective data for the combined benchmarks, divided into strings, methods, and classes. We use roughly the same space as Squeak for strings and methods. For classes, we use a little more memory than KVM, but a lot less than Squeak: 72%. The chart also shows that 69% of the memory used by KVM is used for methods. We will explore the memory usage for each of these categories in the following sections.

Figure 6.2 Memory usage for reflective data by category

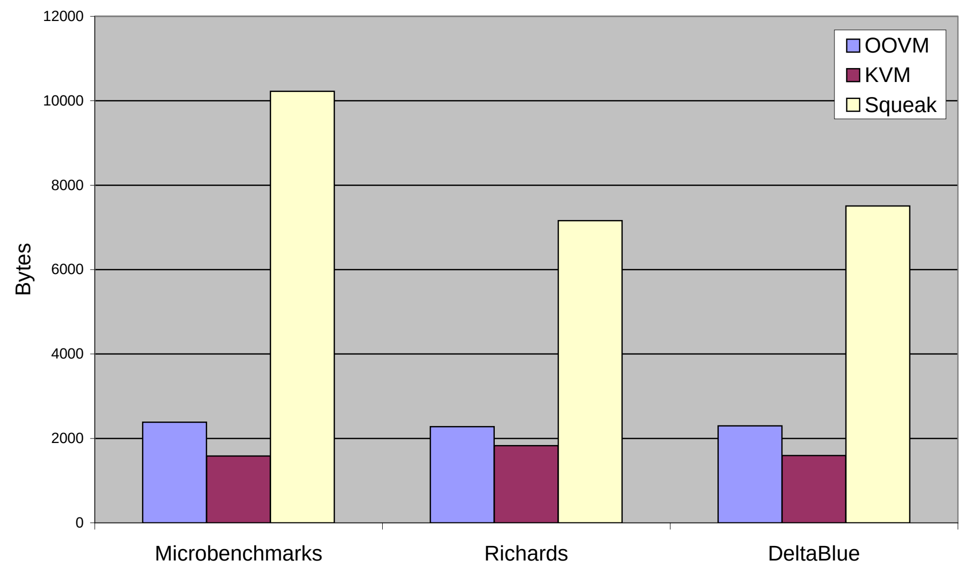


6.1.2 Classes

The amount of memory used for classes is shown in figure 6.3 on the next page. Squeak spends comparatively large amounts of memory on classes, because they contain superfluous information such as method categories and lists of subclasses. This information is used by the Smalltalk programming environment which is an integral part of the runtime environment in Squeak.

Classes in our system take up slightly more space than the classes in KVM due to two things. First, each class in our system contains a list of the classes contained in its namespace. KVM has no notion of namespaces. Instead, it has a central system dictionary listing all the classes in the system. We have not included the system dictionary in the memory usage for KVM. Second, a method in KVM contains a pointer to its selector. Our methods do not, because this would limit our ability to share methods. This means that the classes in our system have to contain a pointer to the method selector in addition to the pointer to the method. In KVM, only the pointer to the method is needed in the class, because the pointer to the selector is in the method instead.

Figure 6.3 Memory used for classes



6.1.3 Strings and Symbols

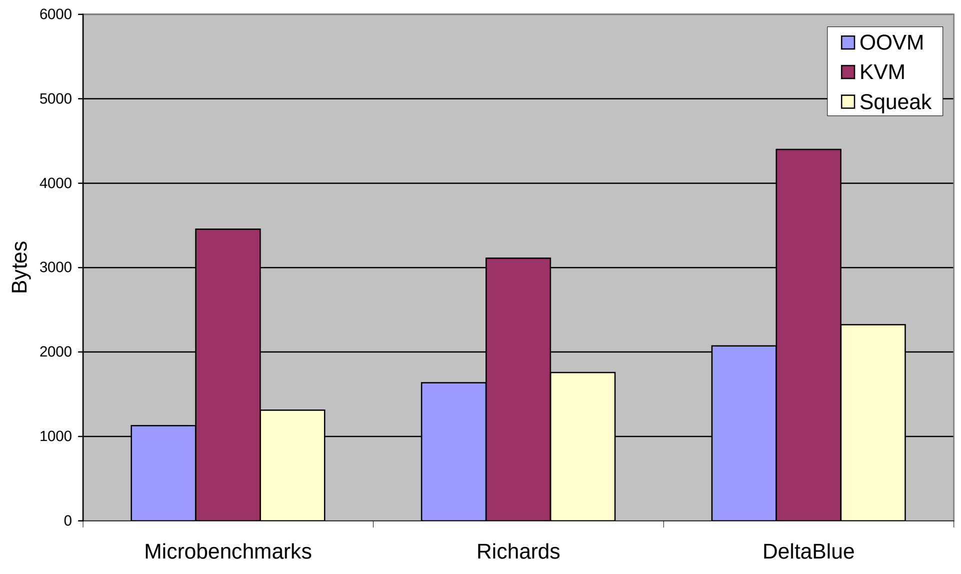
Figure 6.4 on the facing page shows the memory used for strings and symbols for the three benchmarks. We use slightly less memory for strings than Squeak, and a lot less than KVM. In Java, methods have both a name and a signature, which contains type information. A method that takes a `java.lang.Object` as argument and returns a `java.lang.String` has the signature `(Ljava/lang/Object;)Ljava/lang/String;`. The use of symbols for type information is the main reason why KVM uses up to three times as much memory for strings and symbols as our virtual machine does.

Even if the symbols were equal, KVM would still use more memory for strings, because all strings in Java are wrapped in a `java.lang.String` object which contains a pointer to a character array. In our virtual machine, and in Squeak, the character array is embedded inside the `String` instance. This way, we save an object and a reference per string.

6.1.4 Methods

Figure 6.5 on page 130 shows the amount of memory used for methods. We use about the same amount of memory as Squeak on Richards and

Figure 6.4 Memory used for strings and symbols



DeltaBlue, but 1.2 KB more on the micro-benchmarks. This is probably due to our use of a uniform instruction set encoding, where each instruction uses two bytes. We expect that the size of our methods will be reduced, once we optimize the superinstruction set.

KVM uses more than twice the amount of memory for methods compared to our virtual machine. In KVM, the method objects contain reflective information such as the name and signature of the method and which class it belongs to. The methods also contain a number of stack maps that describe the types of stack elements at certain bytecodes. We will examine the memory performance impact of stack maps further in section 6.1.5.

We have implemented a number of memory usage optimizations for methods. Figure 6.6 on the following page shows the impact of different optimizations compared to the fully optimized methods.

Inlining control structures makes our methods 20–30% smaller. This may be somewhat surprising, since inlining integrates more code in the methods. However, the inlined code does not have to be reified as a method object, so we save at least one method object per inlining. The inlined code can also access locals directly without using `load outer local` or `store outer local`, so the inlined code is also smaller. Section 3.2.4.8 gives an example of a method with and without inlining.

Figure 6.5 Memory used for methods

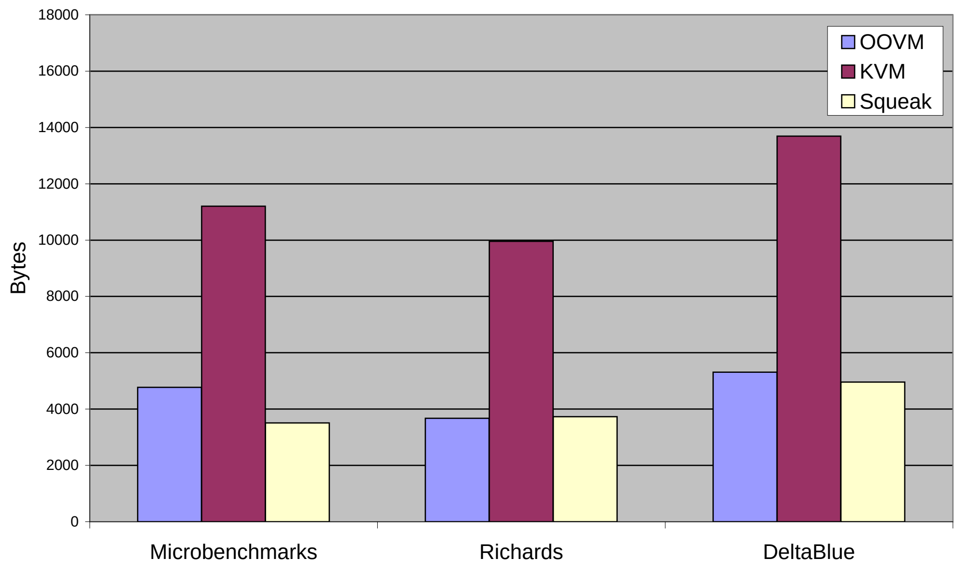
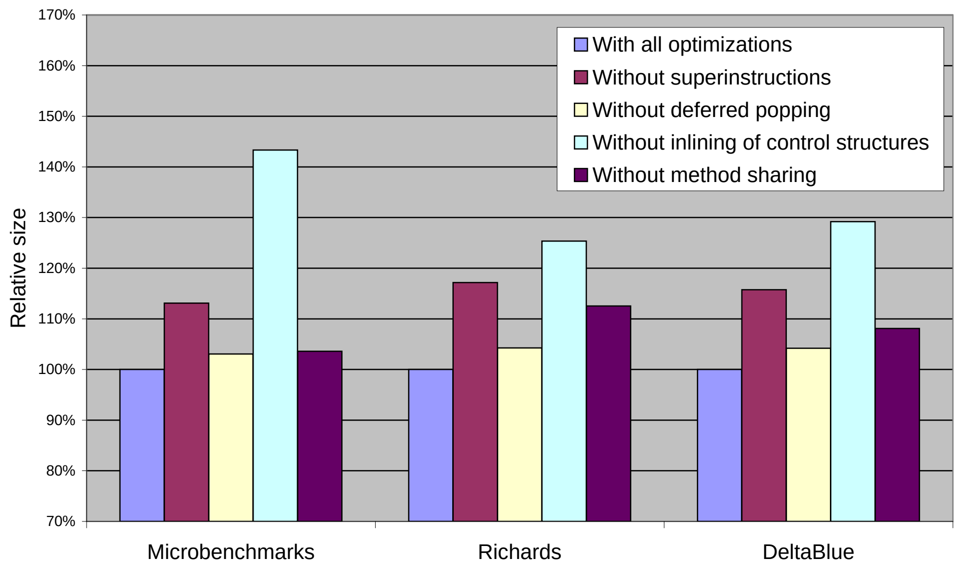


Figure 6.6 Relative effect of method size optimizations

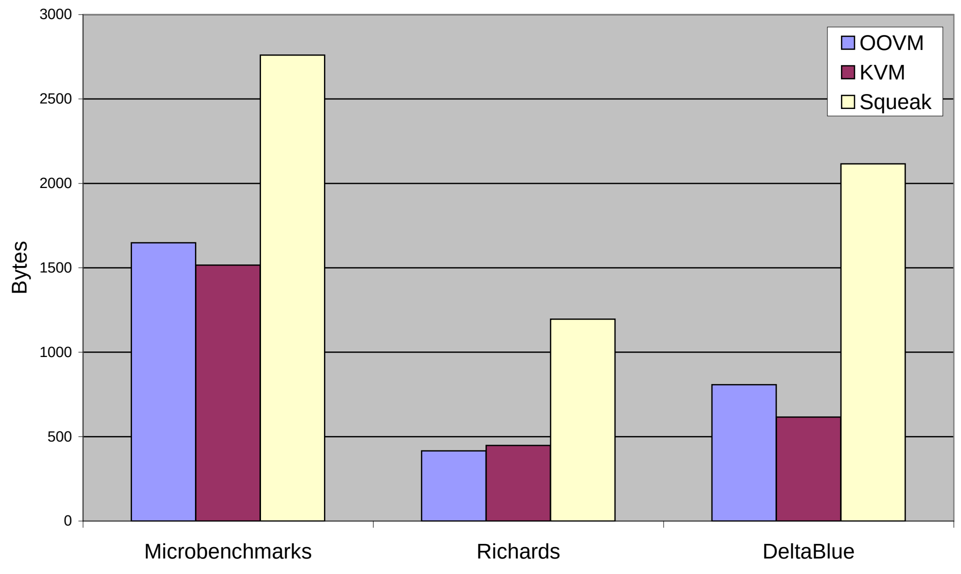


Our superinstructions save 12–15% on the benchmarks. We anticipate an even greater reduction with a better superinstruction set. Method sharing saves 3–11%. The most commonly shared methods are accessor methods which are frequent in object-oriented code. The frequency of accessors is highly code-dependent, which explains why the size reduction due to method sharing varies. Deferred popping saves 3–4%, but has a negative impact on the amount of stack space used. We will explore this effect further in section 6.1.5.

6.1.5 Stack Sizes

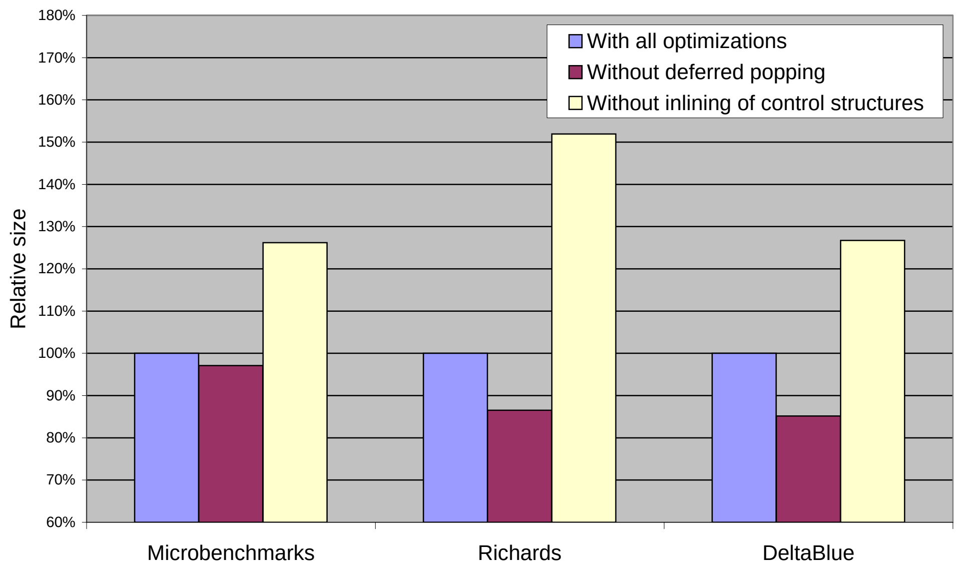
Figure 6.7 shows the maximum amount of stack space used for running the benchmarks. On the micro-benchmarks, we use 40% less stack space than Squeak. On the larger, more realistic benchmarks, we perform even better, using 62–65% less stack space than Squeak.

Figure 6.7 Maximum stack space used



We use 100–200 bytes more stack space than KVM on the micro-benchmarks and DeltaBlue, and 30 bytes less on Richards. The extra stack space used on the micro-benchmarks and DeltaBlue may be due to a couple of things. First of all, we pop elements from the stack lazily. Figure 6.8 on the following page shows that without deferred popping, the maximum stack space used is reduced by 3–15%. Deferred popping makes the methods smaller and potentially improves execution times, but the price is larger stacks. We will have to examine this effect further to decide if we want to keep this optimization.

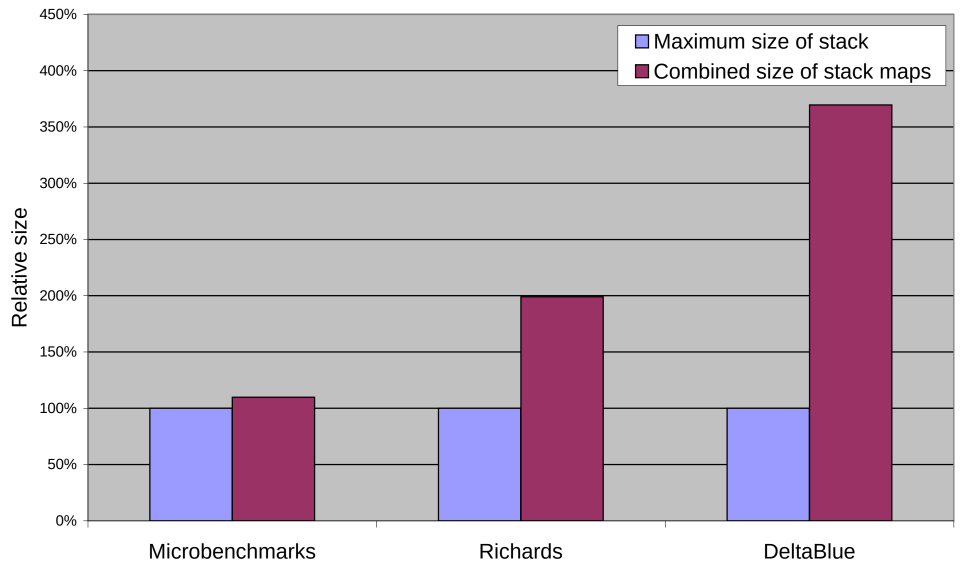
Figure 6.8 Relative effect of optimizations on stack usage



Second, there are differences in the implementation language. For control structures, Smalltalk relies more on method invocations than Java does. We try to even out the effect by inlining as much as possible. Without inlining of control structures, we use 26–52% more stack space. It is quite possible that we can reduce the stack sizes further by inlining more than we do right now. We will have to examine the benchmarks in detail to find out where the stack space is used.

For garbage collection purposes, KVM uses stack maps to describe the layout of execution stacks; see section 6.1.4. Figure 6.9 on the next page compares the combined size of the stack maps for the benchmark classes to the maximum amount of stack space used to run the benchmarks. We have not included the size of the stack maps for the Java libraries, even though garbage collections may occur in library code. As the chart shows, the stack maps take up at least as much memory as the stack. In fact, for the DeltaBlue benchmark, the stack maps are more than three times larger than the maximum used stack space. On average, the stack space used at any given point in time will be much smaller, so this is the best-case scenario for the stack maps.

Figure 6.9 Size of stack maps relative to used stack space in KVM



The stack maps take up so much space because there has to be a stack map for every instruction that can be interrupted in a method. Since methods usually have many interrupt points, and the stacks are small, it is a better idea to keep the layout information on the stacks. CLDC Hotspot does not use stack maps, but instead stores an explicit type tag with each element. The tags are stored as 32-bit words next to the value on the stack; see figure 3.26 on page 38. This doubles the size of the stacks, but as figure 6.9 shows, explicit type tagging still saves memory compared to using stack maps.

6.2 Execution Model

To evaluate the design and implementation of our virtual machine, we have compared its performance to other interpreted virtual machines. We have chosen two Smalltalk virtual machines and two Java virtual machines. Squeak and KVM are described in section 6.1. We expect that the performance of these virtual machines will suffer from the fact that they both have an interpreter written in C.

Smalltalk/X (4.1.7) is a freely available Smalltalk-80 implementation, which we have used to bootstrap our system. Smalltalk/X is developed by eXept Software AG. It dates back to at least 1987, and for that reason we expect the system to be mature and to have good performance. We are interested in the interpreted performance of the system, and therefore we have disabled its just-in-time compiler.

Hotspot (1.4.0-b92) is Suns Java implementation for desktop systems. Its execution engine consists of an interpreter and an adaptive compiler. In all our measurements, we have disabled adaptive compilation. This way, we are only measuring the interpreted performance of Hotspot. Since the Hotspot interpreter is the fastest Java interpreter available, we expect Hotspot to perform well.

Even though our virtual machine runs on both Intel IA-32 and ARM hardware architectures, we have performed all measurements on an Intel Pentium® III with a clock frequency of 1133 MHz. See appendix A for details on the hardware platforms. The reason for this is that Smalltalk/X and Hotspot have not been ported to the ARM architecture. Furthermore, none of the other virtual machines are able to run without an underlying operating system. For that reason, we have chosen to host all virtual machines on Redhat Linux 7.3.

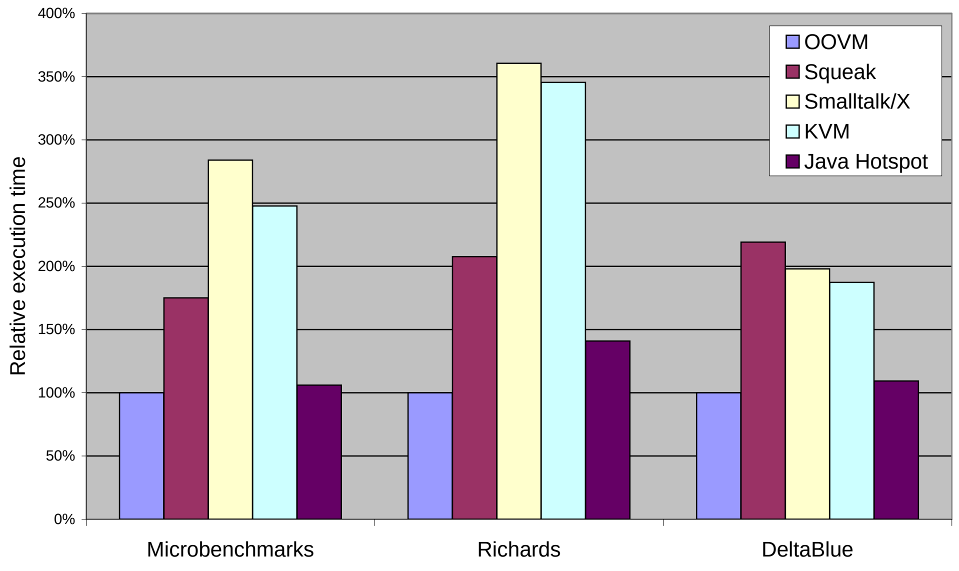
6.2.1 Overall

Figure 6.10 on the facing page shows the relative execution times for the benchmarks on the virtual machines. To allow cross-benchmark comparisons, we have normalized the execution time to that of our system. The graph shows that our virtual machine is 5–29% faster than Hotspot and 43–72% faster than the rest of the virtual machines. We get the highest speedup over Hotspot when running the Richards and DeltaBlue benchmarks. Both benchmarks are very call-intensive. This indicates that we have very fast dynamic dispatches. The primary difference between the two benchmarks is that DeltaBlue allocates a lot of objects while running. Therefore, we expect the performance of DeltaBlue to improve considerably, when we optimize our memory management system.

Figure 6.11 on page 136 shows the relative execution times for the set of micro-benchmarks. Our virtual machine outperforms the other virtual machines on all the dynamic dispatch benchmarks. The fibonacci, towers, dispatch, recurse, and list benchmarks are all very call-intensive. For these benchmarks our virtual machine is 17–36% faster than Hotspot. This is yet another testimony to our fast dynamic dispatches.

Another category of benchmarks consists of the loop, queens, and sum benchmarks. On these benchmarks, Hotspot is 20–49% faster than our vir-

Figure 6.10 Execution time comparison

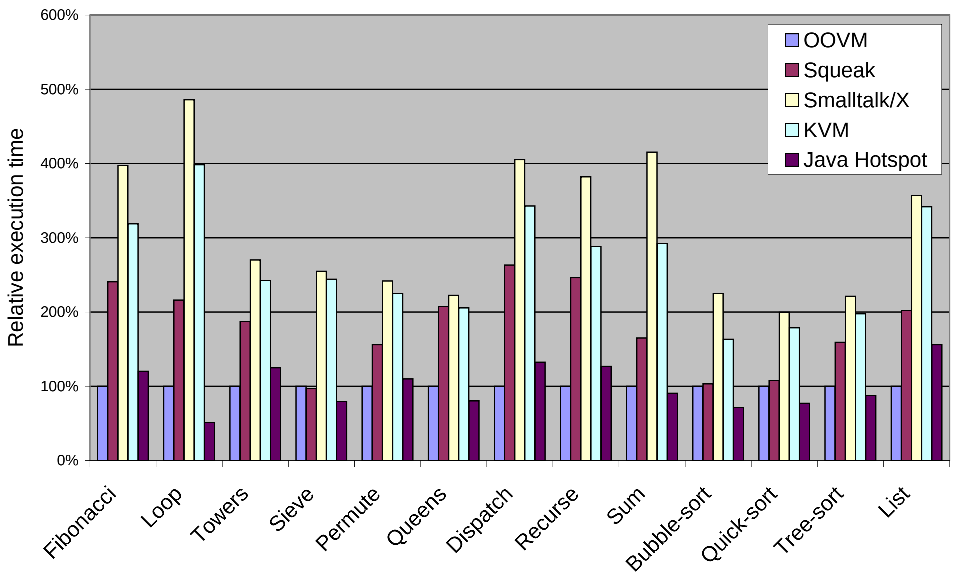


tual machine. This is the result of better branch prediction for instruction dispatches and better native instruction scheduling in the implementation of backward branch instructions. Sections 6.2.3 and 6.2.4 explain why this is the case.

On the sorting benchmarks, Hotspot is 13–29% faster than us. The sorting benchmarks allocate and access memory during execution. This impairs the performance of our virtual machine on these benchmarks, since our memory management system is not yet optimized. The main reason why Hotspot is only 13% faster than us on the tree-sort benchmark is that tree-sort is recursive and contains many method calls.

The permute and sieve benchmarks contain both looping and memory access. The difference between the two is that permute contains many calls. For that reason our virtual machine is 9% faster than Hotspot on the permute benchmark, whereas Hotspot is 21% faster than us on the sieve benchmark. The reason why Squeak is 3% faster than us on the sieve benchmark is that Squeak avoids many index bounds checks by optimizing array filling. We have not yet implemented such optimizations for our system.

Figure 6.11 Execution time comparison on micro-benchmarks



6.2.2 Block Efficiency

The graph in figure 6.12 on the facing page shows the execution time of a simple, recursive, block-intensive micro-benchmark on the Smalltalk virtual machines. The benchmark constructs a linked list and uses blocks and recursion to compute its length. The recursive `length` implementation is shown in figure 6.13 on the next page. It follows from the implementation that the micro-benchmark allocates at least one block-context per level of recursion, and that the non-local return in the `[ ^n ]` block must unwind at least as many contexts as the length of the linked list.

The graph shows that the execution time is linearly dependent on the recursion depth for all virtual machines. It also shows that enforced LIFO blocks makes our virtual machine almost 78% faster than the virtual machines for Squeak and Smalltalk/X, when it comes to interpreting block-intensive code. Better yet, our interpreter outperforms the just-in-time compiled version of the Smalltalk/X system by more than 16%.

6.2.3 Register Caching

Figure 6.14 on page 138 and figure 6.15 on page 138 show the execution time effect of register caching in the interpreter of our system. The graphs

Figure 6.12 Execution time for block-intensive micro-benchmark

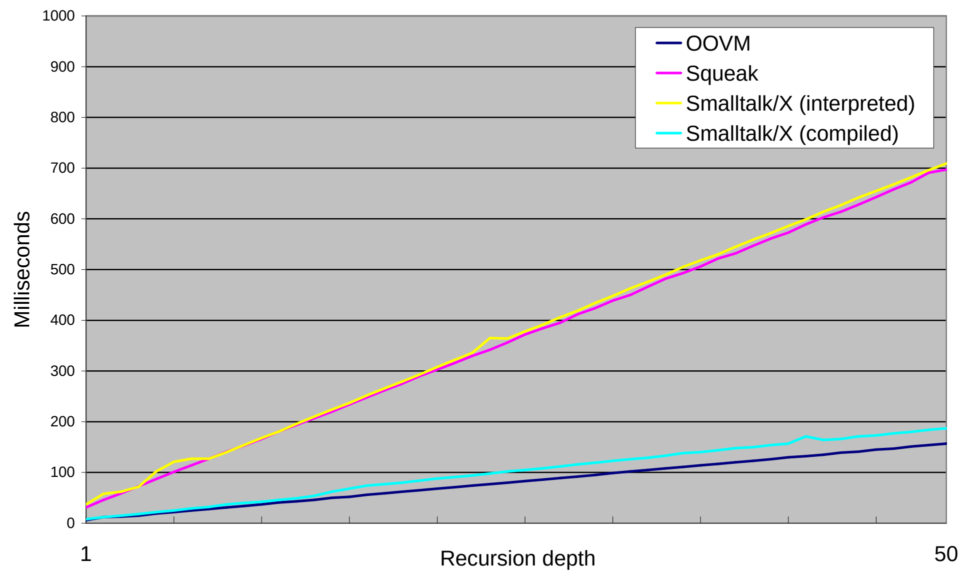


Figure 6.13 Implementation of block-intensive micro-benchmark

```
Element = Object (
  | next |

  length = (
    | n |
    n := 0.
    self do: [ :e | n := n + 1. e ifLast: [ ^n ]. ].
  )

  do: [block] = (
    block value: self.
    next do: block.
  )

  ifLast: [block] = (
    next isNil ifTrue: [ block value ].
  )
)
```

show that our system would have been 35–51% slower without register caching in the interpreter, or equivalently that adding register caching to an otherwise optimal system yields a speedup of 26–34%.

Figure 6.14 Effect of register caching

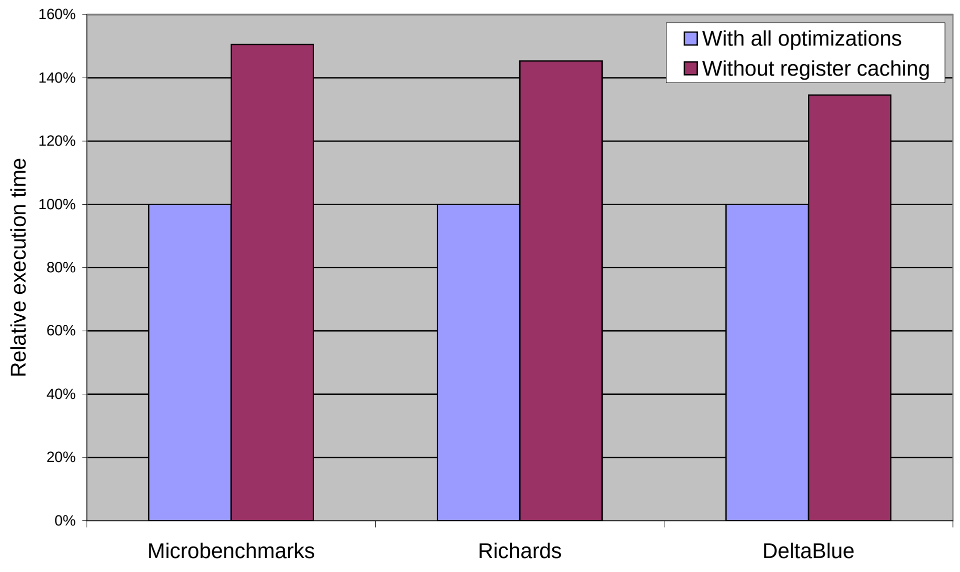
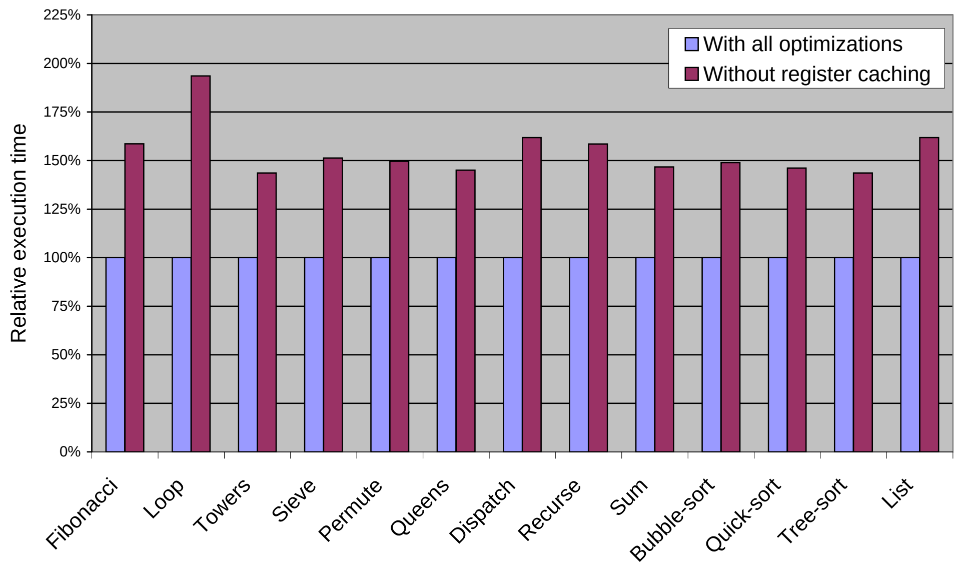
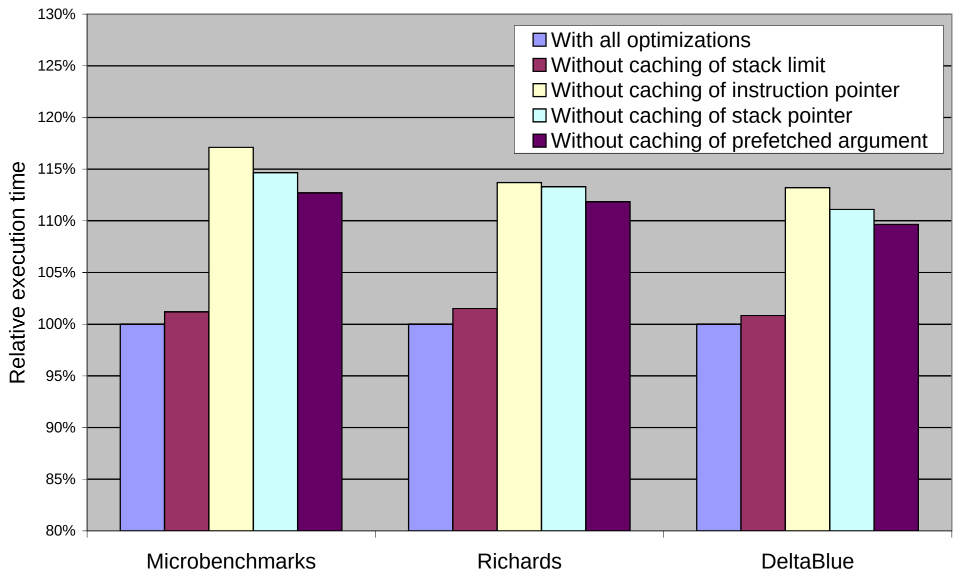


Figure 6.15 Effect of register caching on micro-benchmarks



The graph in figure 6.16 shows the individual execution time effects of register caching. Caching of the stack limit is the least beneficial optimization shown. It only makes our system 0.8–1.5% faster. For that reason, we are considering removing the optimization entirely from our system. The remaining register caches all contribute with speedups ranging from 8.8% to 14.6%.

Figure 6.16 Individual effect of register caching



The individual execution time effects of register caching on the microbenchmarks is shown in figure 6.17 on the next page. It is interesting to see that the loop benchmark runs considerably faster without the stack limit caching. Implementing stack limit caching slows down the benchmark by 19%. An understanding of why this happens provides insights into how lower execution times for the loop benchmark can be achieved.

In the loop benchmark, the only frequently executed instruction that uses the stack limit is the `branch backward` instruction. It is executed at the end of each iteration of the inner loop. Figure 6.18 on the following page shows the implementation of this instruction in the Intel IA-32 interpreter with stack limit caching in register `ebp`. When executing the loop benchmark, the instruction preceeding the `branch backward` is always a `pop` instruction. The `pop` instruction modifies the stack pointer cache in register `esp`. It appears that this register modification causes the processor to stall during the `branch backward` stack overflow check. Without caching the stack limit, the stack limit must be loaded from memory before doing the stack overflow check. Apparently, the instruction scheduling resulting from this is superior to the scheduling shown in figure 6.18. This indicates that it is possible to increase performance by reordering the instructions, without having to remove the stack limit cache.

Figure 6.17 Individual effect of register caching for micro-benchmarks

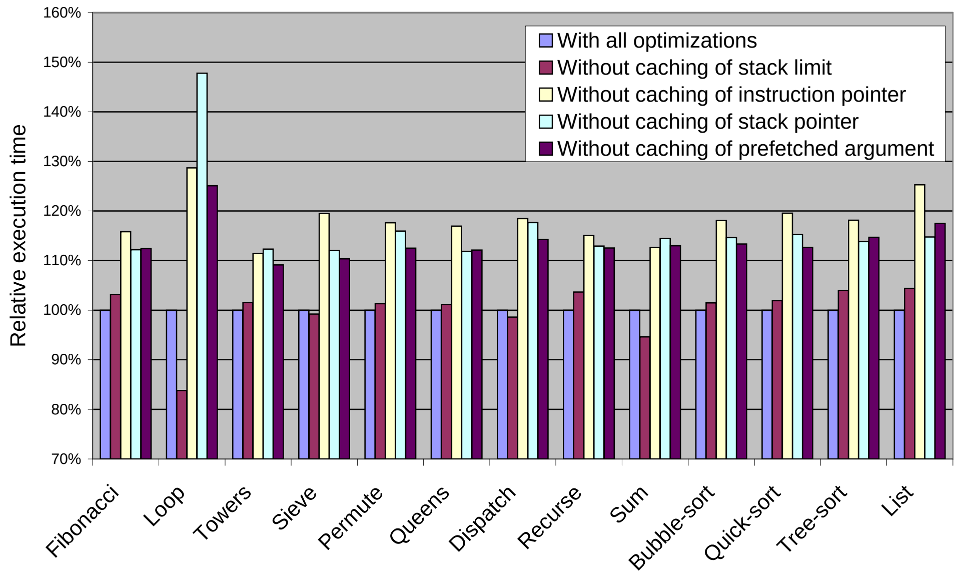


Figure 6.18 Intel IA-32 native implementation of `branch backward`

```
cmp  ebp, esp       ; check for stack overflow to catch interrupts
ja   interrupted

sub  esi, edi       ; subtract the argument from the instruction pointer

...                 ; go to the instruction at the updated instruction pointer
```

6.2.4 Interpreter Threading

Figure 6.19 on the next page and figure 6.20 on page 142 show the execution time effect of threading the interpreter. The graphs show that interpreter threading gives a speedup of 21–32% compared to an otherwise optimized virtual machine. The graphs also show estimated execution times of our interpreted system in the presence of zero-cost interpreter dispatch. We have estimated this base execution time by measuring execution times with a non-threaded interpreter, where we have doubled the cost of dispatch. The following equalities hold:

$$T = B + D \qquad T_{double} = B + 2 \cdot D$$

where T and T_{double} are the execution times of the non-threaded interpreter with and without double dispatch cost, D is the time spent dispatching, and B is the remaining execution time. By subtracting T from T_{double}, we can compute D. This yields a formula for computing the base execution time B; the estimated execution time with zero-cost dispatch:

$$B = T - D = T - (T_{double} - T) = 2 \cdot T - T_{double}$$

The differences between the estimated base execution times and the measured execution times constitute the dispatch overhead. Without interpreter threading, the virtual machine spends 65–85% of its total execution time dispatching from one instruction to the next. Threading the interpreter reduces this fraction to 56–78%.

Figure 6.19 Effect of interpreter threading

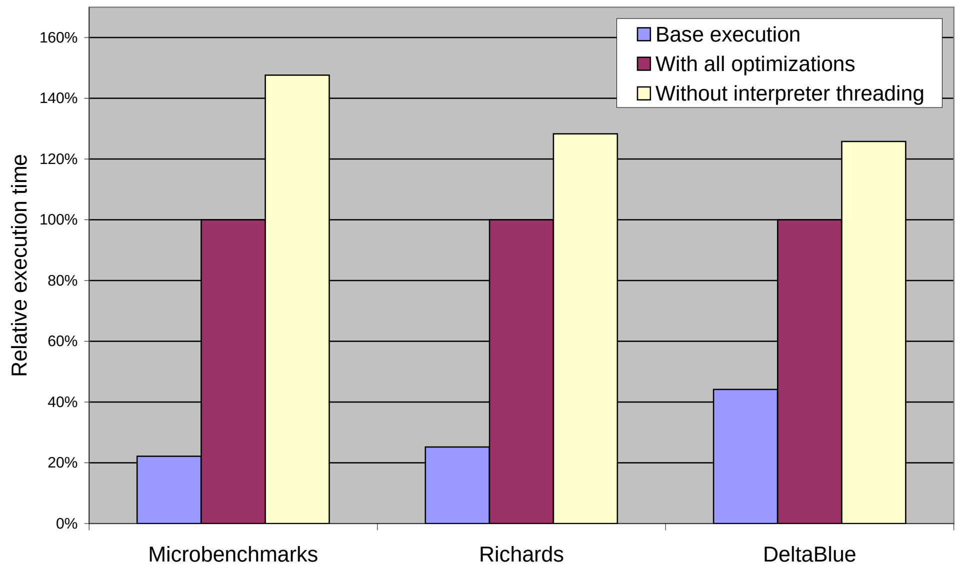
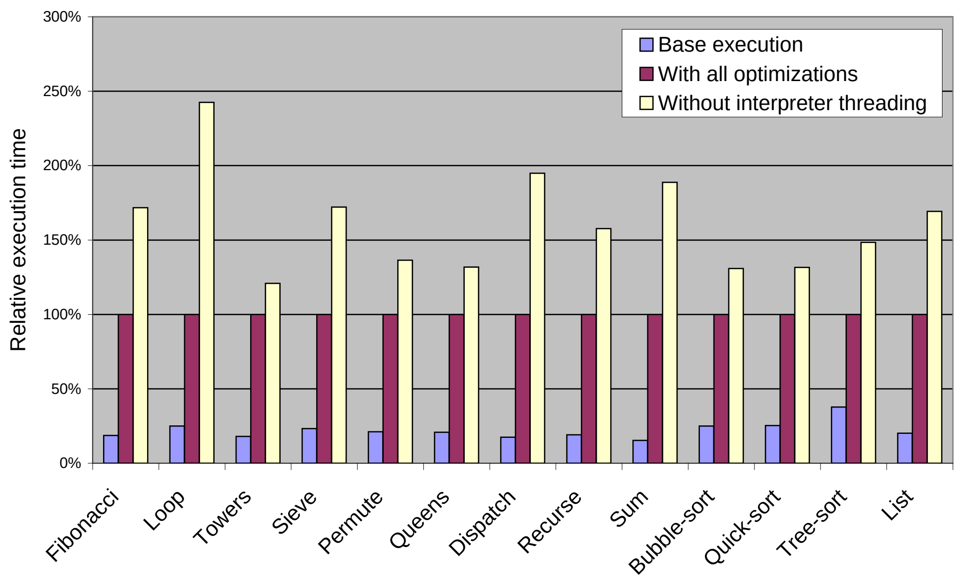


Figure 6.20 Effect of interpreter threading on micro-benchmarks



The high cost of instruction dispatching also explains why Hotspot executes the loop benchmark faster than our virtual machine. Figure 6.21 on the next page shows the unoptimized Java bytecodes for the loop benchmark. Notice how each instruction only appears once in the instruction sequence. This means that the indirect jump at the end of each instruction has a theoretical branch prediction accuracy of 100%. The bytecodes can be optimized by noticing that the `iload`, `iconst`, `iadd`, `istore` sequence is equivalent to a single `iinc` instruction. Unfortunately, the optimized version has two occurrences of `iinc` in the inner loop. This means that the indirect jump at the end of this particular bytecode cannot achieve more than 50% branch prediction accuracy with standard branch target buffering. The result is that the optimized version is 34% *slower* than the unoptimized version, even though the optimized version has three instructions less in the inner loop. Like the optimized version for Java, the inner loop compiled for our virtual machine also has duplicate instructions. This is part of the explanation why the loop benchmark performs better on Hotspot than on our virtual machine.

Figure 6.21 Java bytecodes for the inner loop in the loop benchmark

```
start:
  iload_1
  iconst_1
  iadd
  istore_1
  iinc 3 1
  iload_3
  bipush 100
  if_icmple start
```

Unoptimized

```
start:
  iinc 1 1
  iinc 3 1
  iload_3
  bipush 100
  if_icmple start
```

Optimized

6.2.5 Top-of-stack Caching

The graphs in figure 6.22 on the following page and figure 6.23 on the next page show the performance impact of introducing top-of-stack caching. Overall, the speedup due to top-of-stack caching is between 0.3% and 3.0%. For the micro-benchmarks, the speedup ranges from 1.5% to 11.7%. Even though the overall performance impact is far from impressive, we have chosen to keep top-of-stack caching in our system to improve the performance of critical inner loops.

6.2.6 Frequency of Monomorphic Sends

We have gathered send statistics from our interpreter while running the entire benchmark suite. The graph in figure 6.24 on page 145 shows that 82.0% of all dynamically executed sends are monomorphic. Without inlining of control structures, the percentage drops to 66.5%. This is due to the fact that the receivers in conditional processing expressions frequently change from `true` to `false` or vice versa. For that reason, most of the sends eliminated by inlining of control structures are megamorphic. The graph also shows that out of all send instructions, 84.9% are monomorphic sends.

Figure 6.22 Effect of top-of-stack caching

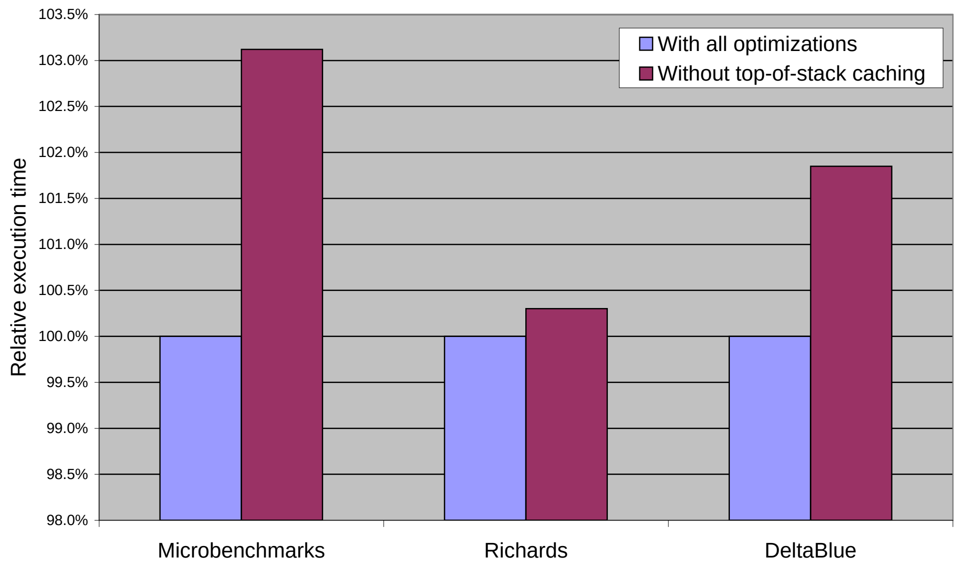


Figure 6.23 Effect of top-of-stack caching on micro-benchmarks

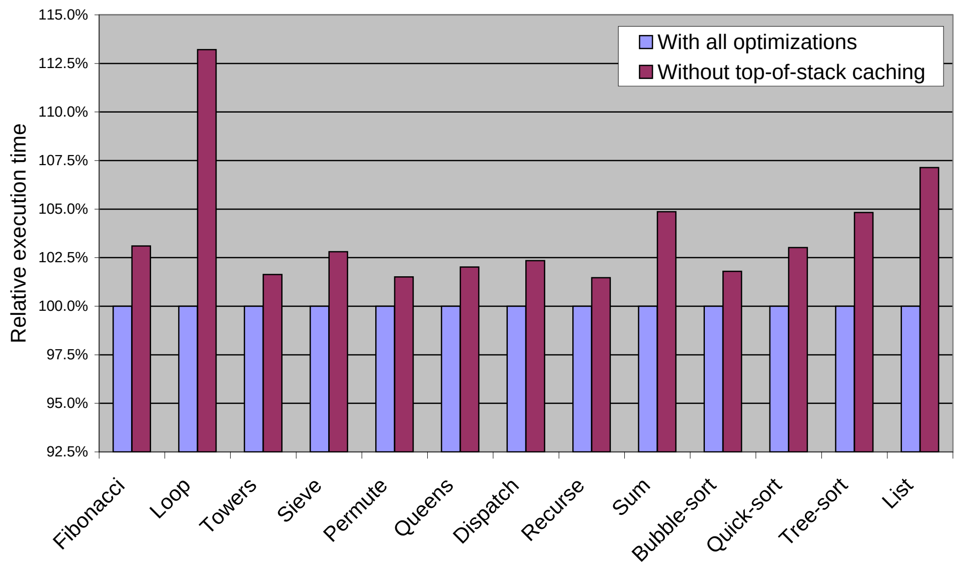
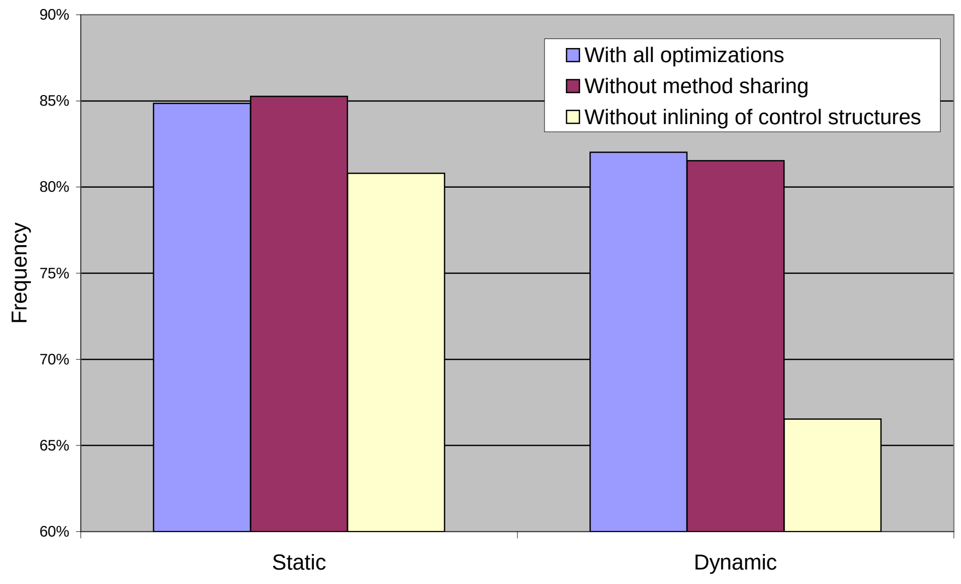


Figure 6.24 Frequency of monomorphic sends



Method sharing can potentially decrease the frequency of dynamically executed monomorphic sends, because sharing methods containing sends, reduces the number of send sites. Our measurements show that this is the case, but that the decrease is negligible. Reducing the number of send sites may also affect the *static* frequencies. The graph in figure 6.24 shows that method sharing increases the static frequency of monomorphic send instructions slightly.

6.2.7 Hash Function

We have measured the lookup cache hit ratio using three different hash functions: `and`, `xor`, and `add`. The results with inline caching disabled are shown in figure 6.25 on the next page, and the results with inline caching enabled are shown in figure 6.26 on the following page.

The hit ratio with inline caching enabled is higher for all functions, most likely because there are fewer cache table lookups due to the inline caching. There is some difference between `xor` and `add` on the smaller cache sizes, but this evens out as the cache size grows. Our measurements show that `xor` is slightly better than `add` on almost all cache sizes, and that `and` is not a good hash function.

Figure 6.25 Lookup cache hit ratio without inline caching

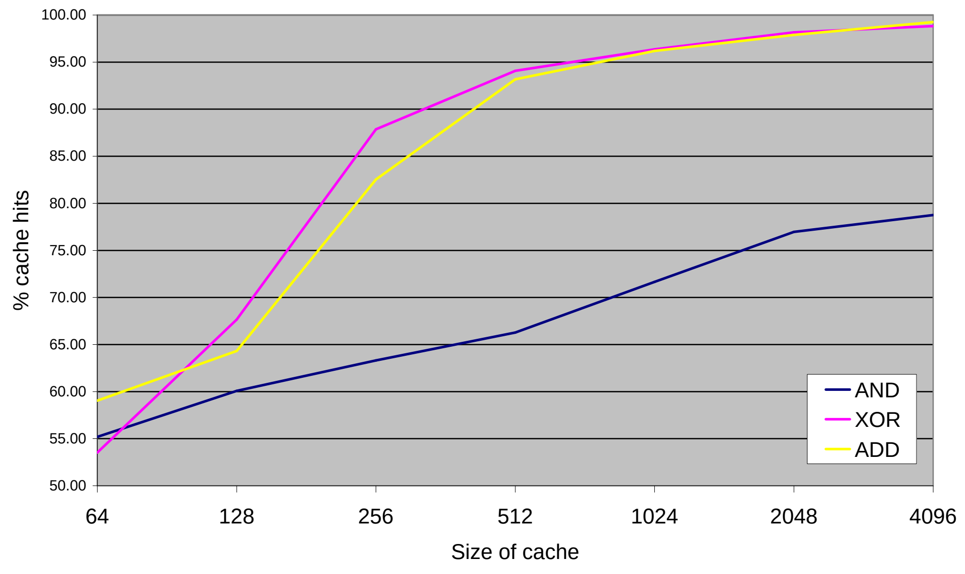
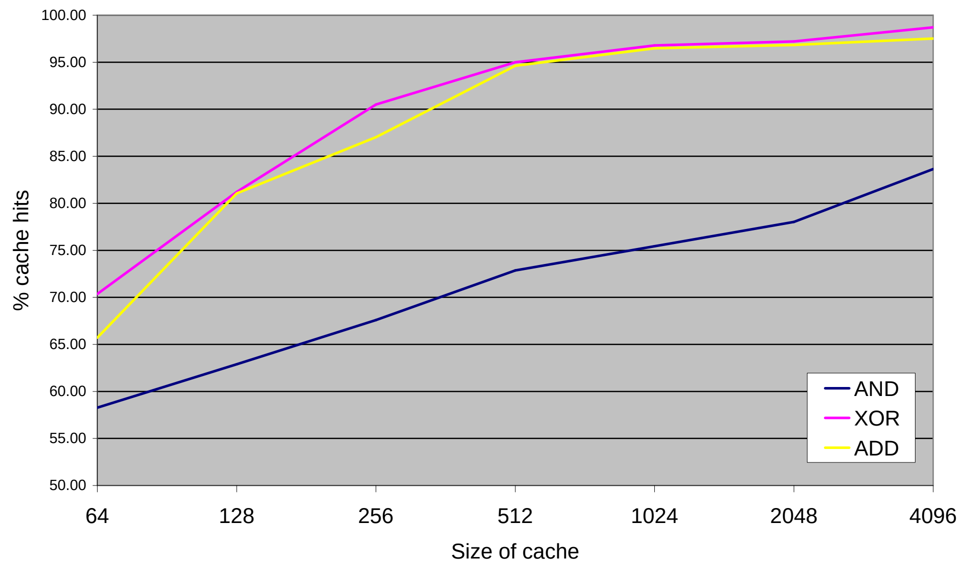


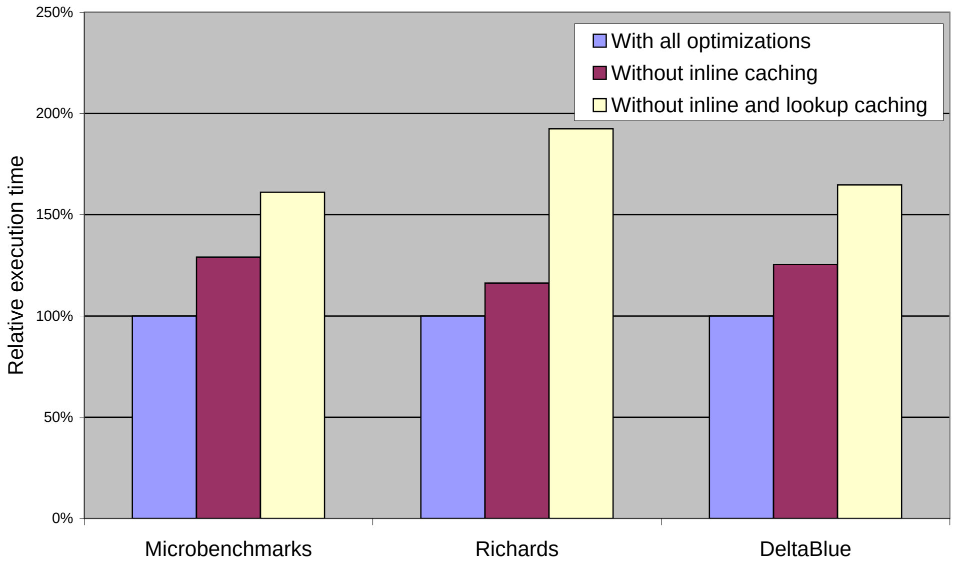
Figure 6.26 Lookup cache hit ratio with inline caching



6.2.8 Lookup Caching and Inline Caching

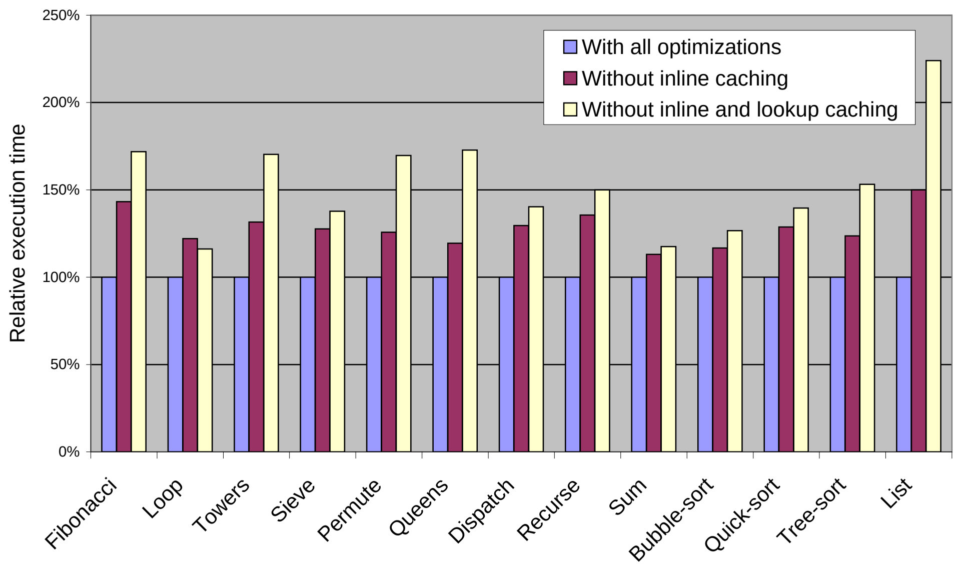
We have measured the effects of lookup caching and inline caching. The graph in figure 6.27 shows the result of these measurements. Lookup caching alone yields a speedup of 20–40% over an otherwise optimized virtual machine. With inline caching, another 14–23% is cut off the execution time. It is important to note that the reported speedup of inline caching includes the effect of inline accessors and primitives. Section 6.2.11 shows the isolated effect of these optimizations.

Figure 6.27 Effects of lookup and inline caching



The graph in figure 6.28 on the following page shows the effects of lookup caching and inline caching on the micro-benchmarks. This graph is similar to the graph shown in figure 6.27, except for the loop benchmark. It is interesting that removing lookup caching actually speeds up this particular benchmark. The only frequently performed method lookup in the loop benchmark is for <= in the small integer class. This method is listed third in the class definition for small integers. This means that looking up the method without a cache only requires traversing the first three elements in the method array of the class. Such a traversal results in three pointer comparisons; one for each of the selectors. With lookup caching, two comparisons are required; one for the selector and one for the receiver class. On top of that, lookup caching requires hashing the selector and the receiver class. The net effect is that lookup caching slows down method lookup, if the target method is implemented as one of the first methods in the receiver class. Lookup caching remains beneficial if the method is implemented in a superclass of the receiver class.

Figure 6.28 Effects of lookup and inline caching on micro-benchmarks

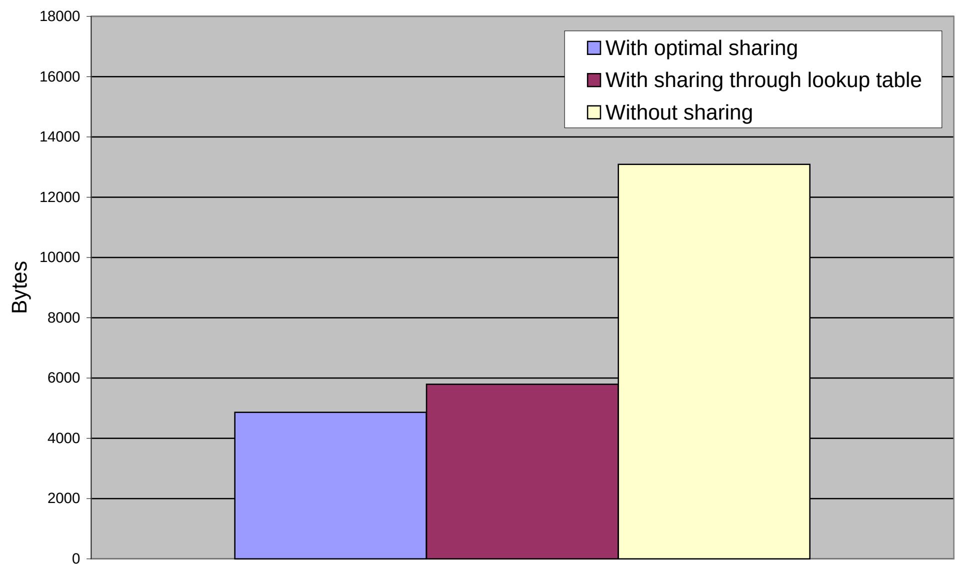


6.2.9 Inline Cache Sharing

To evaluate the effectiveness of our implementation of inline cache sharing, we have computed the number of cache elements needed to run the entire benchmark suite, with and without sharing. The graph in figure 6.29 on the next page shows the total number of bytes spent on cache elements in the two situations. We save more than 7 KB of memory by sharing inline cache elements. This is equivalent to an 18.8% reduction of the combined size of the reflective data and the cache elements.

We have also measured the degree of sharing achieved by our implementation. By counting the number of *unique* cache elements created by a run of the benchmark suite, we compute the size of the cache elements needed with optimal sharing. The graph in figure 6.29 shows that our sharing is 88.9% of the optimal sharing.

Figure 6.29 Effectiveness of inline cache sharing



6.2.10 Polymorphic Inline Caching

We have implemented polymorphic inline caching in our interpreted system. With polymorphic inline caching, monomorphic sends that fail the inline cache check are rewritten to polymorphic sends. At the same time, the cache elements associated with these sends are extended to hold an extra (class, method) pair. The polymorphic sends are thus able to handle two receiver classes without being rewritten into megamorphic sends. The graphs in figure 6.30 on the following page and figure 6.31 on the next page show that polymorphic inline caching in interpreted systems has a negative performance impact for most benchmarks. The only benchmark that benefits from the optimization is the list benchmark, which executes many polymorphic `isNil` sends. The performance penalty incurred for the remaining benchmarks is likely due to instruction cache behavior in the processor.

Figure 6.30 Effect of polymorphic inline caching

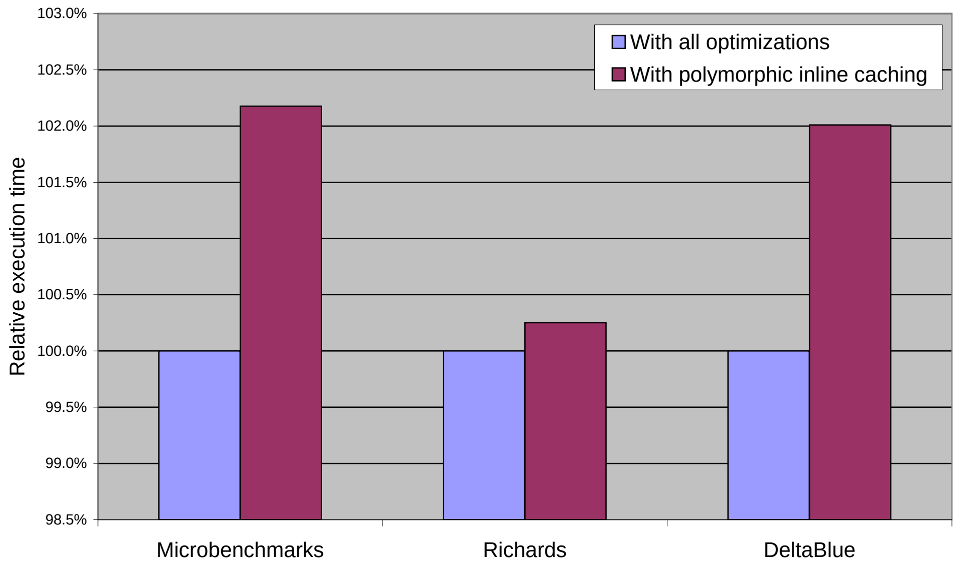
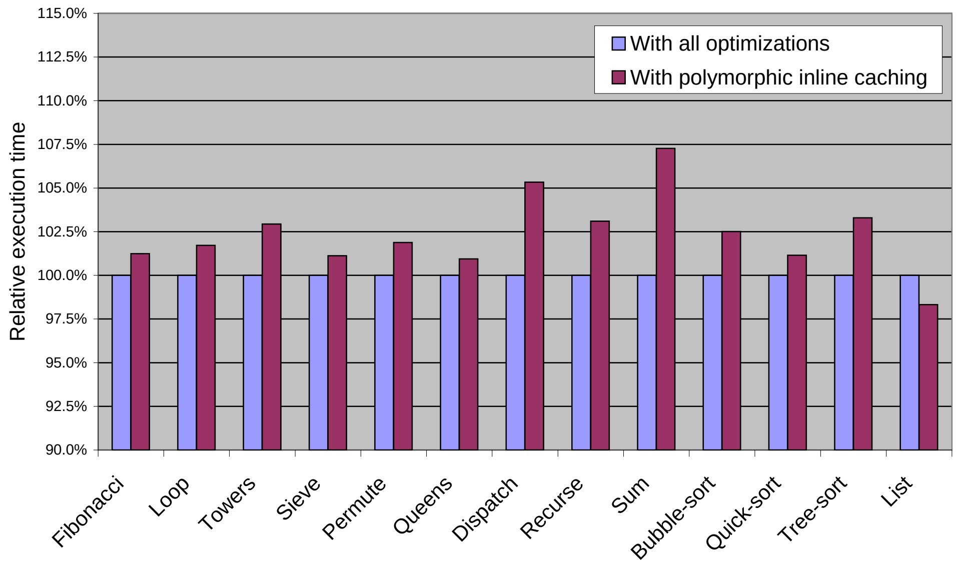


Figure 6.31 Effect of polymorphic inline caching on micro-benchmarks



6.2.11 Dynamic Customization of Sends

Figure 6.32 shows the effect of inline accessors and primitives. We have measured a speedup of 5–8% when using inline accessors, and a 2–9% speedup when using inline primitives. Contrary to the micro-benchmarks, Richards and DeltaBlue benefit more from inline accessors than from inline primitives. This is likely to be the case for real applications as well.

Figure 6.32 Effect of inline accessors and primitives

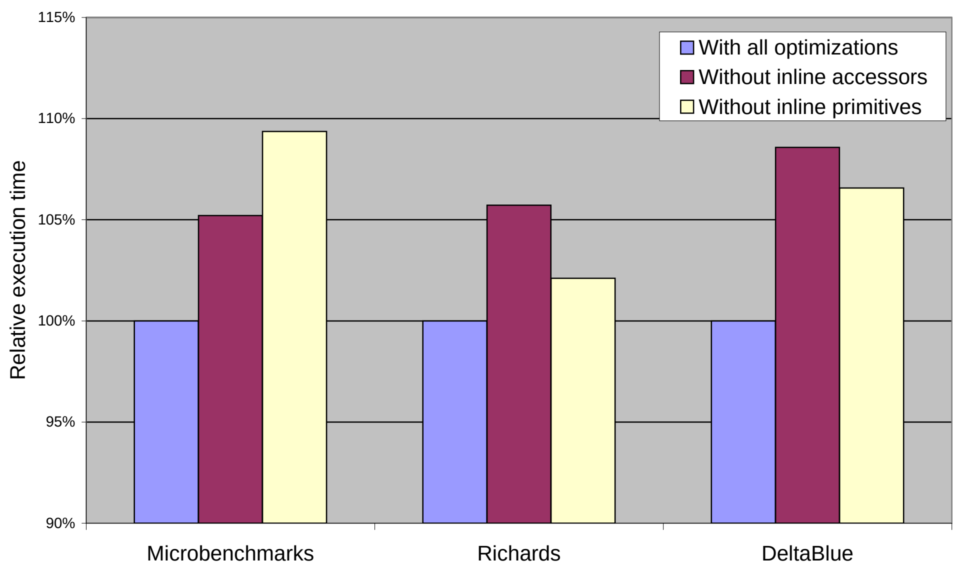
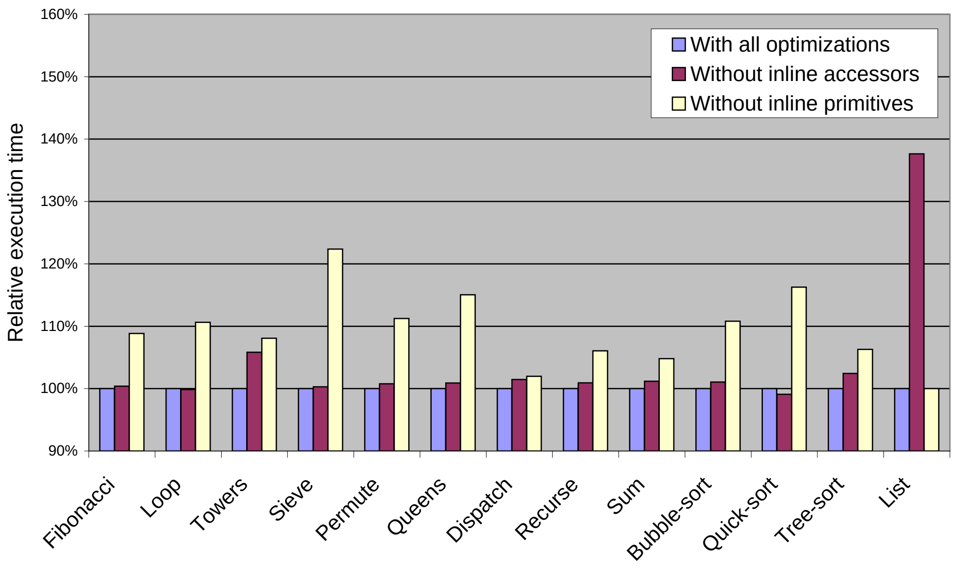


Figure 6.33 on the next page shows the effects on the individual micro-benchmarks. As expected, the benchmarks that depend on array access, such as the sieve, permute, queens, and quick-sort benchmarks, yield a speedup of 10–18% from inline primitives. This is because the frequently used `at:` and `at:put:` methods on arrays are primitive methods. Similarly, the list benchmark uses a lot of accessor sends, and gains a speedup of 27% from inline accessors. The loop and quicksort benchmarks are 0.2–0.9% faster without inline accessors. This is most likely due to caching effects in the processor.

Figure 6.33 Effect of inline accessors and primitives on micro-benchmarks



6.2.12 Inlining of Control Structures

Figure 6.34 on the facing page shows the effect of inlining control structures on execution time. Inlining control structures yields a speedup of 30–57%. Figure 6.35 on the next page shows the effect on each of the micro-benchmarks.

As expected, the micro-benchmarks that measure performance of loops and other control structures benefit most from the inlining. The loop benchmark gains a speedup of 84%, whereas the less control structure intensive benchmarks, such as the fibonacci, towers, and recurse benchmarks, yield a speedup of 28–39%.

Figure 6.34 Effect of inlining control structures

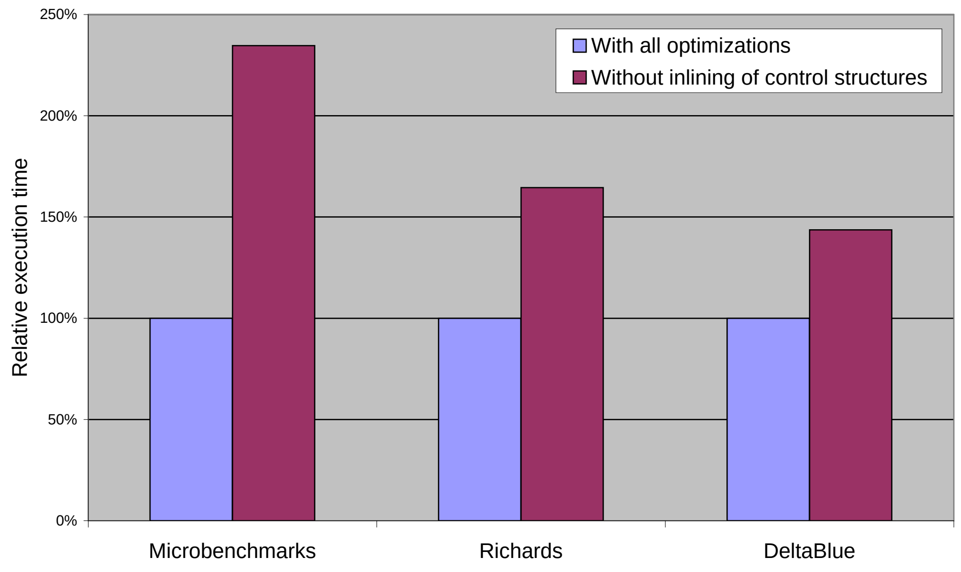
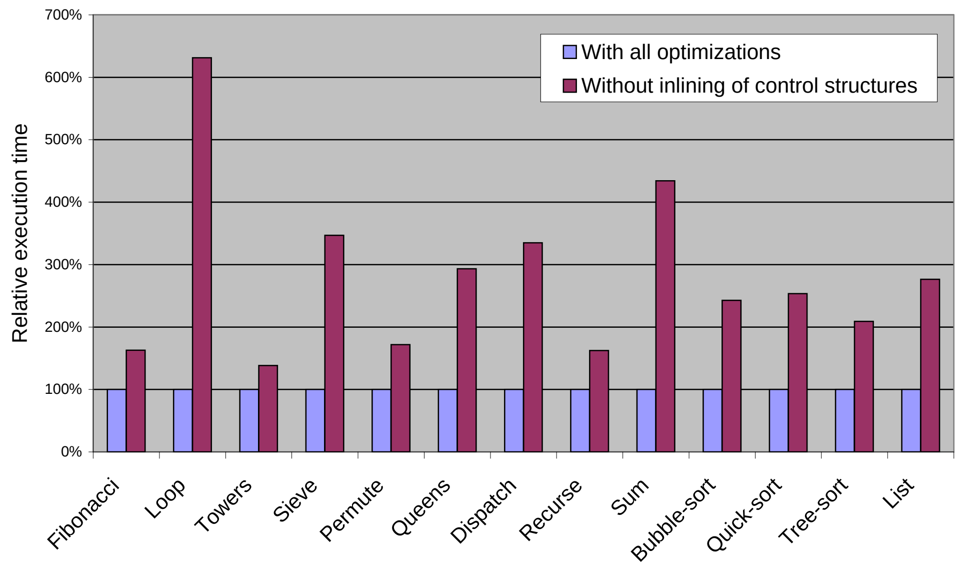


Figure 6.35 Effect of inlining control structures on micro-benchmarks



Chapter 7

Conclusions

Today, it is exceedingly difficult to debug, profile, and update code running on embedded devices in operation. This leaves developers unable to diagnose and solve software issues on deployed embedded systems. This is unacceptable for an industry where robustness is paramount. We have shown that it is possible to build a serviceable software platform that fits on memory-constrained embedded devices; something we believe will revolutionize the way embedded software is maintained and developed.

Developing software for embedded devices has traditionally been complicated and slow. Source code is compiled and linked on the development platform, and the resulting binary image is transferred onto the device. If the source code is changed, the entire process must be restarted. We have shown that it is possible to use an interactive programming environment for developing embedded software, thereby simplifying development and increasing software productivity.

Our software platform is based on virtual machine technology. At the bottom of our software stack, we have replaced real-time operating systems with an efficient 30 KB object-oriented virtual machine. Interrupt handlers, device drivers, and networking protocols are implemented as system software components running on top of the virtual machine. Consequently, we have shown that it is feasible to have full runtime serviceability for system software.

We have designed our system with memory constraints in mind. Compared to other object-oriented virtual machines, our compact memory representation of objects has allowed us to reduce the amount of memory spent on classes, methods, and strings by 40–50%. The result is that our entire software stack fits in less than 128 KB of memory. This makes our technology applicable to a wide range of industrial and consumer devices.

Our virtual machine uses an efficient interpreter to execute both system software and applications. On average, our interpreter is more than twice as fast as KVM, the industry standard Java virtual machine for low-end embedded devices. It even outperforms the fastest Java interpreter available by 5–29%. Thus, we have shown that it is possible to have efficient interpretation of object-oriented software on embedded devices with less than 128 KB of total system memory.

7.1 Technical Contributions

We have provided an overview of state-of-the-art virtual machine technology, and we have described the design and implementation of our system in details. We have made several technical contributions. First, in section 3.2.2.2, we have shown that enforcing LIFO behavior for blocks enables highly efficient interpretation of block-intensive code. Second, we have shown how the memory requirements of inline caching can be reduced by sharing cache elements. Even on rather limited sets of benchmarks, inline cache sharing, as described in section 3.2.4.6, saves several kilobytes of memory. Last, but not least, we have shown how to replace the reflective libraries in Smalltalk with a reflective interface that allows remote debugging, profiling, and updating of running code.

7.2 Future Work

In this section, we will look at the shortcomings of our virtual machine implementation and how we intend to address them. It should be noted that our implementation is not merely a prototype; many parts of our virtual machine are production-quality. The issues discussed here will all be solved as a part of the productization process.

We mentioned in section 6.2.1 that our memory management system is not yet fully optimized. Automatic memory management is an integral part of an object-oriented virtual machine, and our simple copying collector is holding us back on several benchmarks. To improve on this, we have designed a new real-time, concurrent garbage collector which we are in the process of implementing. Some applications of embedded systems have strict real-time demands, and the new garbage collector is essential in meeting those demands.

We will also be focusing on the system software component of our system. There is room for improvement in the scheduling and event handling system, and there are several optimizations pending for the network stack. We will also be implementing more device drivers, 802.11 wireless LAN support, IEEE 1394 (FireWire) and Bluetooth^{TM} communication stacks, and better support for streaming data.

Superinstructions help to reduce the size of methods. There is work to be done in determining an optimal set of superinstructions. This work is dependent on having large amounts of code to analyze so we can find common instruction pairs. We will also need to determine how dependent the superinstruction set is on the code running on the device. Our experience with superinstructions indicates that it is possible to create a superinstruction set that will work well with different applications. Thus, we may not have to optimize the superinstructions for a particular set of applications. Once we have found a good superinstruction set, we will implement it in the interpreter. We expect that superinstructions will provide a reduction in execution time, since the dispatch overhead between the instructions in a superinstruction is eliminated, and we are looking forward to measuring the performance gain.

The streamlined virtual machine is just part of our embedded software platform. We want to improve the way embedded software is developed and maintained by creating a fully serviceable software platform. The virtual machine is the enabling technology that makes this possible. The other part of the platform is the programming environment which includes source code compiler, version control, debugger, and profiler. Currently, the debugger is in the planning stages and the profiler is a standalone tool. Both will be integrated into the programming environment, and we have many features planned for them that we will be implementing at the same time. We have also designed a source code versioning system that we will be implementing. As our focus shifts from the virtual machine to the system software running on it, we will be using the programming environment ourselves to develop and maintain the system software.

We have used the Smalltalk/X system mentioned in section 6 to bootstrap our programming environment. Now that our virtual machine has matured, we will migrate the programming environment to our own platform. The programming environment will be using the network stacks of our system software, and as such provide a perfect testbed for the applicability of our system software.

Finally, the entire system needs tuning. As we have mentioned in this thesis, micro-benchmarks can be useful for tuning small parts of the system. However, to fully recognize which parts of the virtual machine have to be optimized, we need a complete system running realistic applications.

Only then can we gain insight into where time and memory is spent. For this reason, we have focused on implementing a complete system, and have only recently begun tuning it for performance.

7.3 Research Directions

We have focused on building a simple and elegant solution to the serviceability problems in the embedded software industry. For that reason, there are several aspects of our system that we have not yet investigated. Hopefully, future research in these areas will pioneer new and interesting solutions to some of the remaining issues within embedded software.

Adaptive compilation on memory-constrained devices is an interesting research direction. We have based our execution model on interpretation. Even though we have made an effort to push the performance of our system to an acceptable level, it remains unclear if interpretation is fast enough for the majority of embedded systems. It is possible to improve performance by introducing adaptive runtime compilation. Whether or not it is feasible to build an efficient adaptive compiler that fits on low-end embedded devices is an open issue. Another possibility is to use a profiler to determine where time is spent, and a static off-line compiler to compile the methods that are most time-critical.

Another interesting research direction relates to types. We have abandoned static typing and based our platform on a dynamically-typed language. However, static typing can be useful for several things. The most cited benefit of stack typing is *type safety*. Many people have found that type annotations are even more important as checkable documentation. For this reason, it may be beneficial to experiment with *optional* static type systems for dynamically-typed languages. Even though Smalltalk has been retrofitted with a static type system on more than one occasion, the proposed type systems have all been designed with existing Smalltalk-80 class libraries in mind. It is interesting to explore the possibility of co-designing the class libraries and the type system. This may lead to simpler type systems and well-documented class libraries.

Bibliography

[ABD$^+$97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? 1997.

[ADG$^+$99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, 1998.

[Age99] Ole Agesen. Space and time-efficient hashing of garbage-collected objects. *Theory and Practice of Object Systems*, 5(2):119–124, 1999.

[BAL02] Lars Bak, Jakob R. Andersen, and Kasper V. Lund. Non-intrusive gathering of code usage information to facilitate removing unused compiled code. *US Patent Application*, April 2002.

[BBG$^+$a] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, Urs Hölzle, and Srdjan Mitrovic. The Strongtalk bytecodes. Available on *http://www.cs.ucsb.edu/projects/strongtalk/big/bctable.pdf*.

[BBG$^+$b] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, Urs Hölzle, and Srdjan Mitrovic. The Strongtalk system. Available on *http://www.cs.ucsb.edu/projects/strongtalk/*.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[BFG02] David F. Bacon, Steven J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. *Springer LNCS*, 2374, June 2002.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[BG02] Lars Bak and Steffen Grarup. Method and apparatus for facilitating compact object headers. *US Patent Application*, April 2002.

[BGH02] Lars Bak, Robert Griesemer, and Urs Hölzle. Mixed execution stack and exception handling. *US Patent*, July 2002. #6,415,381.

[BKMS98] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[BL02] Lars Bak and Kasper V. Lund. Method and apparatus for facilitating lazy type tagging for compiled activations. *US Patent Application*, April 2002.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: Application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[Bor86] Alan H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40. IEEE Computer Society Press, 1986.

[Boy96] Nick Boyd. Class naming and privacy in Smalltalk. *The Smalltalk Report*, 6(3), November 1996.

[Com97] NCITS J20 Technical Committee. Draft American National Standard for Information Systems – Programming Languages – Smalltalk, December 1997.

[CPL84] Thomas J. Conroy and Eduardo Pelegri-Llopart. An assessment of method lookup caches for Smalltalk-80 implementations. *Smalltalk-80: Bits of History, Words of Advice*, pages 239–247, 1984.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3), 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, 1989. ACM Press.

[DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. 1999.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, 1984.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. *Springer LNCS*, 2150:403–412, 2001.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[Gat03] Bill Gates. The disappearing computer. In *The World in 2003*. The Economist Group, 2003.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[Gos95] James Gosling. Java intermediate bytecodes. In *Proceedings of the First ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118. ACM, 1995.

[GR84] A.J. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, USA, 1984.

[GS93] Steffen Grarup and Jacob Seligmann. Incremental mature garbage collection. Master's thesis, University of Aarhus, August 1993.

[Gud93] David Gudeman. Representing type information in dynamically typed languages. Technical Report 93–27, University of Arizona, 1993.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. *Springer LNCS*, 512, July 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[Ing84] Daniel H. H. Ingalls. The evolution of the Smalltalk virtual machine. *Smalltalk-80: Bits of History, Words of Advice*, pages 9–28, 1984.

[ISO84] ISO. *35.100: Open systems interconnection (OSI)*. International Organization for Standardization, 1984.

[Mad93] Ole Lehrmann Madsen. *Building Abstractions for Concurrent Object-Oriented Programming*. Computer Science Department, Aarhus University, February 1993.

[MB99] Blair McGlashan and Andy Bower. The interpreter is dead (slow). Isn't it? *Position Paper for OOPSLA '99 Workshop: Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley Publishing Company, 1993.

[Mos87] J. Eliot B. Moss. Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, volume 22, pages 229–240, June 1987.

[PJK89] Jr. Philip J. Koopman. *Stack computers: The new wave*. Halsted Press, 1989.

[Pos81a] Jon Postel, editor. *RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification*. Information Sciences Institute, University of Southern California, September 1981.

[Pos81b] Jon Postel, editor. *RFC 793: Transmission Control Protocol - DARPA Internet Program Protocol Specification*. Information Sciences Institute, University of Southern California, September 1981.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, California, 1995.

[Ree97] Glenn. E. Reeves. What really happened on Mars? Available on *http://research.microsoft.com/˜mbj/Mars_Pathfinder/Authoritative_Account.html*, December 1997.

[Rit99] Tobias Ritzau. Real-time reference counting in RT-Java. Master's thesis, University of Linköping, March 1999.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

Appendix A

Configurations

Our virtual machine currently runs in the three configurations listed in this appendix. The benchmark results given in chapter 6 were made using the **i386-linux** configuration. In addition to the three platform configurations of the virtual machine, we have two different system software configurations. We use the Linux configuration when the virtual machine is hosted on a Linux operating system, and the CerfCube configuration when the virtual machine is running without an operating system on the ARM®-based Intrinsyc CerfCube platform.

arm-native

Our primary development platform is the ARM®-based Intrinsyc CerfCube evaluation board. In addition to the specifications shown below, the CerfCube is equipped with a Cirrus Logic CS8900A ethernet chip. We chose the CerfCube because it met our demands, documentation was readily available, and it came at a low cost. Even though the CerfCube has 32 MB RAM available, we use as little as 128 KB. We have chosen this limit since it matches our target market.

Processor	Intel® StrongARM SA-1110
Processor revision	B-4
Core speed	206 MHz
Volatile memory	32 MB RAM
Persistent memory	16 MB Intel StrataFlash®
Operating system	None

i386-linux

For evaluation purposes, we have ported our virtual machine to a Linux-based system with an IA-32 processor. We chose this platform because competitive virtual machines are readily available for it.

Processor	Intel® Pentium® III
Processor revision	01
Core speed	1133 MHz
Volatile memory	256 MB RAM
Persistent memory	40 GB harddrive
Operating system	Red Hat® Linux® 7.3

arm-linux

The CerfCube ships with a Familiar-based Linux system. Since we have both an ARM processor version and a Linux operating system version, it was natural to combine the two in an ARM-based Linux configuration.

Processor	Intel® StrongARM SA-1110
Processor revision	B-4
Core speed	206 MHz
Volatile memory	32 MB RAM
Persistent memory	16 MB Intel StrataFlash®
Operating system	Intrinsyc Linux® 4.0

Appendix B

Benchmarks

Fibonacci
: Measures the performance of recursive sends by computing fibonacci using a recursive algorithm.

Loop
: Measures the performance of loops by iterating through two nested loops.

Towers
: Measures the performance of array access and recursion by solving the Towers of Hanoi problem.

Sieve
: Measures the performance of loops and array access by computing a number of primes using the Sieve of Eratosthenes algorithm.

Permute
: Measures the performance of loops, array access, and recursive sends by permuting elements in an array using a recursive algorithm.

Queens
: Measures the performance of boolean logic, array access, and loops by solving the Queens problem.

Dispatch

Measures the performance of repeated sends.

Recurse

Measures the performance of recursive sends.

Sum

Measures the performance of loops and simple arithmetic.

Bubble-sort

Measures the performance of array access and loops by sorting an array using the bubble-sort algorithm.

Quick-sort

Measures the performance of array access, loops, and recursive sends by sorting an array using a recursive quick-sort algorithm.

Tree-sort

Measures the performance of object allocation, loops, and recursive sends by sorting an array using an unbalanced binary search tree.

List

Measures the performance of object allocation and recursive sends.

Richards

Measures overall system performance by simulating the task dispatcher in an operating system kernel.

DeltaBlue

Measures overall system performance by solving a constraint system incrementally; see [FBMB90].