Timothy L Harris
Churchill College
tlh20@cam.ac.uk
July 30, 1998

First year report

Contents

1 Introduction

The implementation of the Java Virtual Machine (JVM, [LY97]) has been the focus of widespread effort over the last two years. It is now a standard component of popular web browsers such as Netscape Navigator, Microsoft Internet Explorer and HotJava. It has been implemented for commodity operating systems such as Microsoft Windows and Sun Solaris [sun97]. It has even been implemented in hardware in the picoJava processor [sun98]. However, users and potential users still claim that '*Java is slow*' [HG98].

The work described here does not directly address the techniques for optimizing the performance of the JVM. Instead it concerns two related issues:

- How can the resources of the JVM be shared in a controllable way between different applets and different applications that are being executed concurrently?
- How can existing approaches to optimization be more effectively integrated into the JVM?

This work was undertaken by the author between October 1997 and July 1998 in the context of the Pegasus II project. It is hoped that this work will form deliverable 4.1.4 and much of 4.1.5 and 4.3.

1.1 Overview

Aside from attempts to optimize generated bytecode [BK97], most existing work on improving the performance of Java applications relates to the design and implementation of just in time (JIT) compilers which translate platform-independent Java bytecode into native code at run-time [CFM+97, Age97, Har97].

The fundamental problem with this approach is that programs stall while compilation is in progress. This requires that the compiler-induced delays are reduced to an acceptable level – traditionally by sacrificing optimization, by only compiling some methods [Age97] or by ahead-of-time annotation with optimization hints [HAKN97].

The approach taken here is to provide a separate mechanism through which the programmer, system administrator or user can control *whether*, *when* and *where* compilation occurs. By providing control over *whether* methods are compiled it is possible to avoid unnecessarily compiling large and rarely executed methods (such as class initializers). By controlling *when* methods are compiled it is possible to schedule pauses to occur at convenient times (such as when the user is idle, or before a large computationally-intensive calculation). By controlling *where* methods are compiled it is possible to choose which threads perform compilation (to separate, for example, compilation work from maintaining a responsive user interface).

The techniques described in section 4 can support a wide spectrum of policies – including, for example, traditional JIT compilation, compilation guided by run-time feedback and compilation in a separate background thread.

The issue of resource control is addressed in section 3, which describes the implementation of a new user-level thread scheduler which is able to provide fine-grained

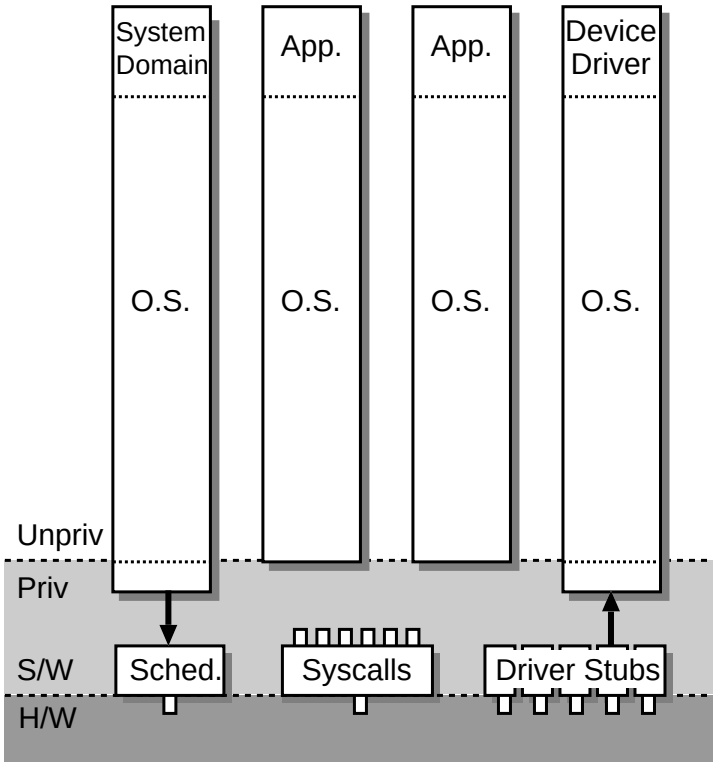

Figure 1: The structure of the Nemesis operating system.

control over the allocation of CPU resources to Java threads. Section 5 describes the implementation of a concurrent garbage collector.

2 JDK port

2.1 Nemesis

The JDK port described here operates over Nemesis [LMB+96], an operating system being developed from scratch at the University of Cambridge, the University of Glasgow, the University of Twente, the Swedish Institute of Computer Science and APM Ltd. Nemesis is designed to provide fine-grained resource control with specific attention to the needs of soft-realtime multimedia and networked applications. Although the work described here operates only on Intel x86-based machines, Nemesis also supports Alpha and ARM platforms.

The structure of Nemesis is summarized in figure 1. Note that there are no 'kernel threads', the use of privileged code is minimized, and applications, system domains and device drivers operate almost-entirely within user space. The motivation for this *vertically structured* design is that it allows accountability between applications and the resources that they use. This is because applications are doing more of their work for themselves rather than relying on the kernel or on shared servers to operate on their behalf.

For example, suppose that a program is receiving data from the network, processing it, and then displaying it on the screen. Wherever possible the network protocol processing and the display operations will be implemented within shared libraries which are accessed by threads running within the application. In contrast, if a similar program was executing over Unix, much of the network protocol processing would occur within the kernel (and would not be accounted to any application) and much of the display work would be undertaken by the X server (again, unaccounted to the application in question).

A consequence of this approach is that *cross-talk* between applications can be reduced so that meaningful resource allocations can be made. This is achieved without sacrificing security – each application can reside in a separate *protection domain* which defines the extent of its access to memory. Packet-filtering techniques control the kinds of network traffic that an application can generate and receive.

2.2 Features

The Nemesis JVM implementation is based on version 1.1.4 of the Sun JDK. This code is licensed under the terms of the *Java internal use source license* which places restrictions on the extent to which the source code to the JVM and any derived binaries can be distributed. However, a clear separation has been maintained between this restricted material and the work described in sections 3-5 so that the latter is not encumbered by the JDK source license.

This section describes different aspects of the port and the extent to which they currently operate.

2.2.1 Core JVM implementation

The core parts of the JVM have no known problems. For example, `.class` files can be read directly from a local filesystem, or via NFS, can be stored individually or in a `.jar` or `.zip` archive. The main interpreter loop is un-modified from the existing Intel platforms supported by the JDK.

Performance, measured using the standard Caffeinemark tests, is comparable to the Linux JDK port running on a similar machine.

`http://www.webfayre.com/pendragon/cm201/tests.htm`

2.2.2 Native methods

Native methods can be accessed through the standard JNI interface. In other JDK ports native methods are loaded from Windows DLLs or from Unix shared libraries – in Nemesis they are loaded from Nemesis modules.

2.2.3 File-system access

Simple open/read/write/close access to files is supported. However the underlying Nemesis local file-system implementation is read-only. Access to file modification times, file renaming and directory operations is not supported. This is a consequence of the fluidity of the Nemesis interfaces.

2.2.4 Networking

Network support is rudimentary.

Straightforward send/receive operations on UDP sockets are supported. This allows APM's FlexiNet [HF97] work to be demonstrated over Nemesis and allows, for example, communication between applications running in the Nemesis JVM and applications running in Windows or Unix JVMs.

TCP sockets cannot be correctly implemented as a consequence of the lack of a complete Nemesis TCP implementation.

DNS queries have not been implemented.

2.2.5 User interface

The standard user interface support in Java is provided by the *Abstract Window Toolkit* (AWT). This has been implemented elsewhere on several platforms – for example Windows, Motif and the Macintosh. It does, however, rely on the availability of a heavyweight underlying window system. This is expected to provide facilities such as pop-up menus, buttons, scroll-bars etc. Providing a complete AWT implementation over the lightweight Nemesis 'client renders' system would therefore involve a large amount of work and would be impracticable.

However, a restricted subset of AWT operations have been implemented. These are sufficient for the new *Swing* [And98] user interface classes to be supported. Swing provides standard window system components by rendering them in Java. The simple operations that it requires map reasonably cleanly onto the existing 'client renders' system.

3 User-level thread scheduler

The Java Language Specification (JLS, [GJS97]) requires that '*when there is competition for processing resources then threads with high priority are generally executed in preference to threads with low priority*'. This rather loose condition leaves many aspects of the thread scheduling policy dependent on the implementor. The motivation here is that it allows Java threads to be implemented in a reasonably natural way over a variety of hardware and operating systems.

As discussed in [Bla94] and [Ros95], this kind of priority-based scheme is inherently unsuitable for many applications. For example, suppose that a single JVM contains two threads, each running on behalf of a separate Java applet. Based on the language specification it is not possible for the JVM's user to ensure that both threads even run at all – perhaps the JVM's implementor has used a non-preemptive threads package and one of the applets never blocks or yields. Even if a preemptive system is available then it remains impossible to control the allocation of CPU time on a fine granularity – all that can be adjusted is the threads' relative priorities.

This section describes the thread scheduling policy used in the Nemesis JDK port along with some details of its implementation.

3.1 Scheduling policy

The thread scheduling policy is broadly similar to that described in [Ros95] and implemented in the Nemesis *atropos* scheduler.

Threads' CPU requirements are expressed as a (p, s, x, t) tuple, encoding a *period*, *slice*, *extra time flag* and *priority* respectively. For example a requirement of $(10ms, 1ms, False, 0)$ means that that thread should be allocated 1ms of CPU time every 10ms of real-time and that it does not want to receive any 'extra' time that remains when all the allocations have been met.

The *priority* field is used to control how extra time is divided between any threads whose flag is set. This is done on a strict-priority basis with time distributed evenly amongst the threads that have the highest priority. This integration of priorities allows the scheduler to meet its requirements under the Java Language Specification.

The standard Nemesis interface to the threads package allows any thread to enter a critical section – meaning that it will continue running unless it blocks or explicitly yields. Although this facility is still provided, it is not made available to Java applications because it could be used to subvert allocations made by the scheduler.

The ways in which a programmer, user or system administrator should control CPU allocation and the mapping between Java and scheduling-priorities are not addressed here (the corresponding issues in Nemesis more generally are work in progress [Opa98]).

An issue which this scheduler does not address is how to control jitter – since the CPU allocated to a thread may be offered at any time within its period it is possible for a thread to be scheduled slightly irregularly. In the worst case, a thread allocated $1ms$ every $5ms$ could receive $2ms$ of CPU time consecutively – a first $1ms$ slice at the end of a period, followed immediately by a further $1ms$ slice at the start of the next. It is, unfortunately, not possible to address this issue within the context of a Nemesis user-level scheduler since the jitter experienced by the domain as a whole cannot be controlled.

3.2 Scheduler implementation

Two priority queues are maintained: a *run queue* on which all runnable, non-suspended threads are placed and an *allocation queue* containing all threads which have a CPU allocation. Threads on the allocation queue are ordered according to when their next

allocation is due (soonest first). Threads on the run queue are maintained in the following order (first to last):

1. The thread currently inside a critical section.
2. Threads with some remaining CPU allocation. These are held in EDF order according to the end of their current period.
3. Threads which have exhausted their CPU allocation, but which have requested a share of extra time. These are held in priority order (highest priority first).
4. Threads which have exhaused their CPU allocation and which have not requested a share of extra time. Note that this means that threads towards the far end of the run queue may be in a *runable* state (ie neither blocked nor suspended), but that they should not be scheduled.

Let H_r be the thread at the head of the run queue, H_a be the thread at the head of the allocation queue, $a(t)$ be the remaining time allocated to thread t during its current period, let $d(t)$ be the *deadline* of thread t (ie the time by which it should have been offered $a(t)$ CPU time), let $p(t)$, $s(t)$ and $x(t)$ be the period, slice and extra time flags of thread t. NOW is the current time.

When activated, the thread scheduler scheduler performs two tasks. Firstly it charges the current thread with the CPU time that it has used and then updates the run queue and allocation queue. Secondly it uses the items at the heads of the queues to decide how to proceed:

- While $NOW > d(H_a)$ set $d(H_a) = d(H_a) + p(H_a)$, $a(H_a) = s(H_a)$ and re-adjust the queues.
- If the run queue is empty then block the domain until at most $d(H_a)$.
- If the run queue is non-empty and H_r is in a critical section then set an alarm timer for $d(H_a)$ and schedule H_r.
- If the run queue is non-empty, H_r is not in a critical section and $a(H_r) > 0$ then set an alarm timer for $min(d(H_a), NOW + a(H_r))$ and schedule H_r.
- If the run queue is non-empty, H_r is not in a critical section, $a(H_r) = 0$ and $x(H_r)$ then set an alarm timer for $min(d(H_a), NOW + q$ and schedule H_r. q is the maximum amount of extra time that a thread may receive uninterrupted – currently $10ms$.
- Otherwise the run queue is non-empty but the threads on it should not be scheduled. Block the domain until at most $d(H_a)$.

A count is maintained of how long is spent maintaining these queues and selecting threads to scheduler. This count is incremented each time the thread scheduler is activated by the difference between NOW and the time that atropos reports that the domain was scheduled.

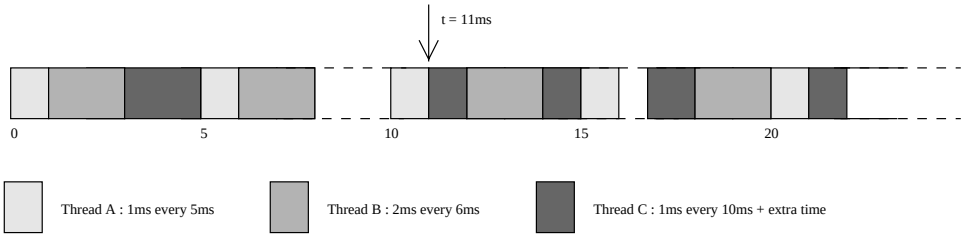

Figure 2: An example of how the CPU may be multiplexed between multiple threads. Real-time runs horizontally from left to right. Shaded regions show where particular threads are being given their portion of the CPU time. The regions with a dashed outline correspond to when threads belonging to a different domain are being executed, outside the control of this scheduler. The time $t = 11ms$ is used as an example in section 3.3.

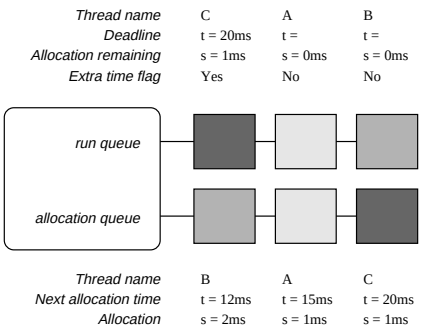

Figure 3: The state of the scheduler's run queue and allocation queue at the time indicated $t = 11ms$ in figure 2.

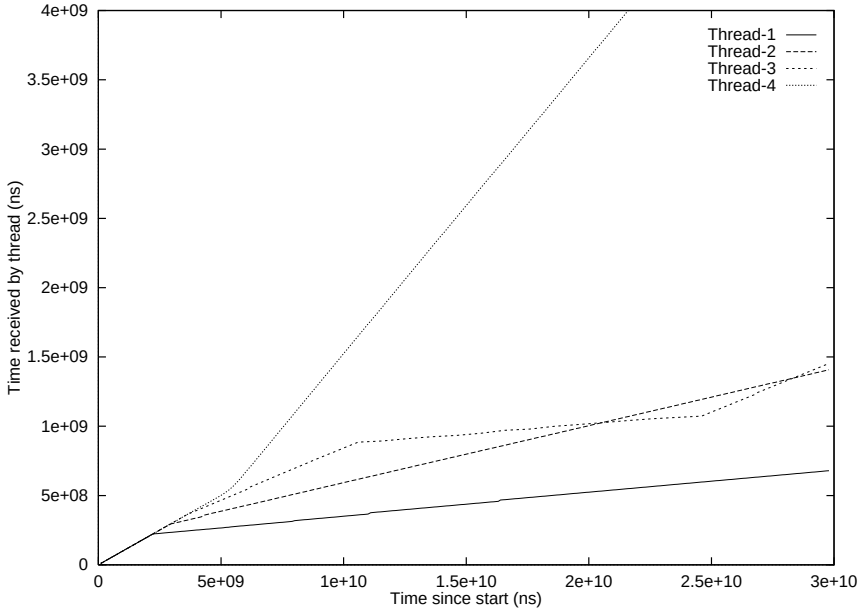

Figure 4: The cumulative time received by a number of threads with varying guarantees.

3.3 Scheduling example

Throughout this example there are assumed to be three runnable threads: thread A is allocated $1ms$ every $5ms$, thread B is allocated $2ms$ every $6ms$ and thread C is allocated $1ms$ every $10ms$. Only thread C is allocated a share of extra time.

Figure 2 shows a possible way in which the thread scheduler may proceed. At time $t = 11ms$, thread A has already received its second allocation of $1ms$ (between $t = 10ms$ and $t = 11ms$) and thread B has already received its second allocation of $2ms$ (between $t = 6ms$ and $t = 8ms$). The only thread which still has any allocated time remaining is thread C which received its first allocation of $1ms$ between $t = 3ms$ and $t = 4ms$. It is allocated $1ms$ every $10ms$ and is therefore due a further $1ms$ before $t = 20ms$. This means that thread C is at the head of the run queue. Although threads A and B remain on the run queue, they will not be scheduled because they have expended all of their allocated time within their current periods. This situation is shown in figure 3.

Figure 5 shows the stability with which the scheduler is able to provide a CPU-bound thread with its allocated time. It shows the time that was received by a thread allocated 10% over a 10ms period during 300 consecutive periods for which it was executing. There were also four other CPU-bound threads running with allocations between 10% and 20% over a 10ms period.

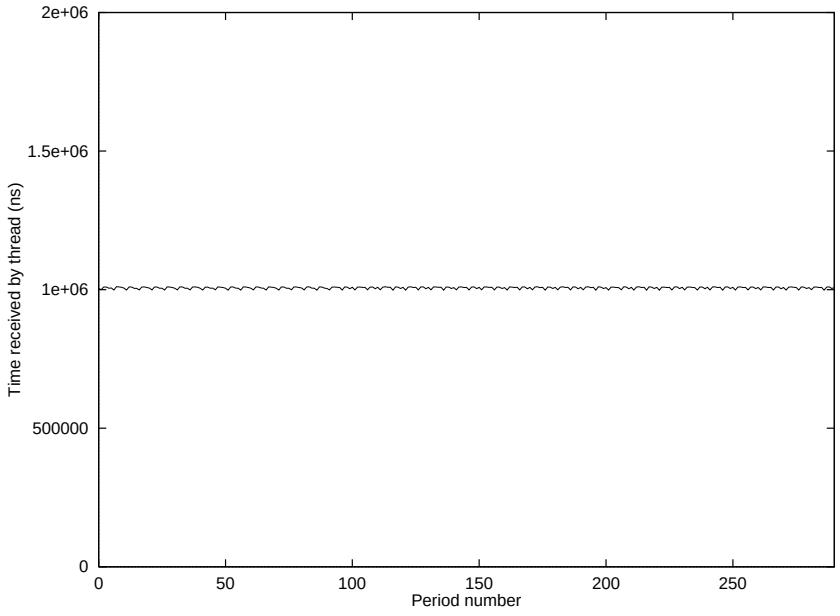

Figure 5: The time received during a single period by a CPU-bound thread. The thread was allocated 10% over a 10ms period.

3.4 Related work

The NewMonics PERC system [Nil98, NL97] is a dialect of Java that is '*designed to support development of cost-effective portable real-time software components*'. It therefore addresses the issue of processor scheduling. It is designed to support hard real-time tasks through the use of language extensions which express timing constraints and the use of bytecode analysis to determine an upper bound on execution time (for a restricted subset of the Java language). PERC employs rate-monotonic scheduling to control the execution of real-time tasks with periodic operation.

Unlike the scheduler described above, the PERC API allows threads to specify the maximum jitter that they will tolerate (ie the maximum amount by which the time that they receive their allocation of the processor may vary from period to period).

4 Run-time compilation

Java bytecodes were always designed with rapid native-code generation in mind [Gos95] and there have already been several attempts at integrating run-time compilation with the JVM [HG98, Bot97, Har97].

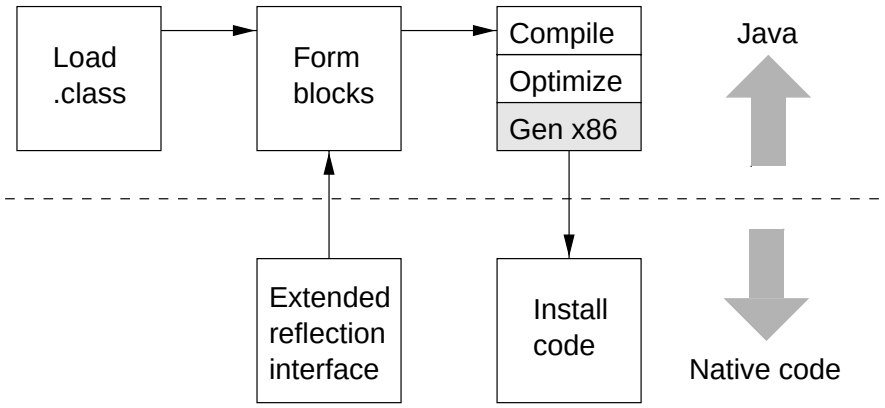

Figure 6: The structure of the compiler.

The work described in this section can be divided into two sub-topics – providing a native-code generator and providing machanisms through which it can be controlled.

4.1 Design of the compiler

The compiler used here is relatively straightforward. As described in section 1, the motivation is to provide a controllable implementation rather than to establish new optimization techniques.

It is apparently novel amongst Java native code generators in that the compiler is written almost entirely in Java (in contrast to the implementations mentioned above). The situations in which native code is required in this implementation illustrate a number of potential weaknesses in the design of the JVM. These issues are discussed in [Har98].

4.1.1 Overview

The structure of the compiler is shown in figure 6. There are three phases involved in the compilation of a Java method:

1. Firstly, the bytecode implementation from the `.class` file is broken into basic blocks and the contents of the constant pool are *resolved* as described in [LY97].

2. Secondly, native code is generated for each basic block. This is produced directly from the stack based bytecode operations (unlike [Bot97, Har97] in which an intermediate 3-address-code [ASU86] representation is produced). Forward branches are handled by back-patching the generated code when the target address becomes known.

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

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

4.1.2 Register allocation

The Intel Pentium Processor [int96] has four 32-bit general-purpose registers (`%eax`, `%ebx`, `%ecx`, `%edx`) and a further four 32-bit registers which may be used with some restrictions (`%esi`, `%edi`, `%ebp`, `%esp`). The compiler reserves these for the following purposes within generated code:

1. `%edi` – Current *execution environment* which contains pointers to the current Java stack frame, the current thread and the current exception (if any) that is being propagated.
2. `%ebp` – The bottom of the Java operand stack.
3. `%esp` – The native stack pointer.
4. `%esi` – The bottom of the current stack frame's local variables table.

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

4.1.3 Optimization

Before optimization (see section 4.1.3) each bytecode instruction is translated to a section of native code which is produced from a fixed template parameterized by the current depth of the Java stack. For example, the code generated for an `iload_1` instruction, when the Java stack already contains two elements, will load the value from the first local variable and then store it into the third slot of the Java stack:

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

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

This effect is illustrated in figure 7 – the Java instruction `r = r * m` is implemented by four bytecodes `iload_2`, `iload_1`, `imul`, `istore_2`.

This provides very little benefit compared to an interpreted implementation or one using threaded code [Bel73]. The increased code size may even harm performance [DMH].

The simple optimization layer aims to improve the quality of native code by lazily generating `movl` instructions and by renaming operands.

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

Figure 7: A Java method (left) and its bytecode implementation (right).

The optimizer maintains details of outstanding `movl` instructions which have yet to be generated. In the above example, the effect of the `iload_2` and `iload_1` instructions is to record that local variable 2 has been transferred to stack slot 1 and that local variable 1 has been transferred to stack slot 2 – no native code is generated at this stage. These mappings can be used when generating code for the subsequent `imul` instruction – instead of accessing data in the Java operand stack it can use values directly from the local variables.

4.1.4 Results

The quality of the generated code is shown in table 1 which compares micro-benchmark scores achieved with and without optimization. Since neither the optimizer nor the interpreter will perform inter-block optimization it is reasonable to believe that these results will scale to large applications and that the overall benefit will therefore depend primarily on the dynamic instruction mix. Measurements with larger applications (such as the `javac` compiler) show that a two-fold speedup is typical.

4.2 Control of compilation

The compiler described above provides a mechanism for compilation at class or method granularity. This section describes how the compilation process is controlled, so that appropriate methods are compiled at appropriate times.

4.2.1 Dispatchers

Compilation is controlled by associating *dispatchers* with particular sub-trees of the package hierarchy. The method `dispatchMethodImpl` is invoked on the dispatcher whenever a method in the sub-tree is called for the first time. A dispatcher is implemented by an instance of a sub-class of `dispatcher.Dispatcher`. The mapping from package names to dispatchers is maintained by static methods of `dispatcher.DispatcherRegistry`.

	Without optimizer	With optimizer
Nop	5.1	6.6
AddInt	2.9	10.0
AddLong	5.4	5.7
AddFloat	1.2	1.2
AddDouble	1.4	1.4
AddByte	2.4	5.8
AddChar	2.5	5.7
AddShort	2.4	5.8
CastToByte	6.5	10.5
CastToChar	6.1	12.0
CastToShort	5.9	11.9
CastToLong	6.0	7.5
CastToFloat	6.2	7.4
CastToDouble	4.9	5.6
CastFromFloat	1.5	1.6
CastFromDouble	1.4	1.5
MethodCall	2.5	2.8
MethodCall (2 arguments)	2.4	2.9
MethodCall (3 arguments)	2.8	2.8
MethodCall (4 arguments)	2.7	2.9
StaticMethodCall	1.1	1.2
InterfaceMethodCall	1.4	1.5
SuperclassMethodCall	1.0	1.1
SynchronizeOnThis	1.2	1.3
CatchSameMethod	1.1	1.1
CallInterpretedMethod	0.9	0.9
NewArray	5.7	6.6
ArrayAccesses	4.9	5.5
NewInstance	0.8	0.8
IterativeFactorial	3.4	8.1

Table 1: Microbenchmark results showing the speed-up of individual Java operations when compiler with and without optimization. Results are expressed relative to the original JDK 1.1.4 interpreter which would score 1.0 in each category.

This provides a mechanism for introducing class-specific processing into standard method invocations. It is therefore similar to the meta-class facilities provided in Smalltalk [GR83] or in some extended dialects of C++ [MMAY95]. However, unlike traditional meta-class designs, the *dispatcher* scheme provides clearer separation between the implementation (and perhaps the *implementor*) of a class and meta-class.

For example, figure 8 shows how dispatchers may be used to implement a particular execution policy. The standard Java classes whose names begin `java.*` are registered with a dispatcher which will load a pre-compiled implementation (perhaps one that was generated off-line with a highly optimizing compiler). Classes whose names begin `UK.ac.cam.cl.tlh20.*` will be compiled in the background (see section 4.2.3). The single class `UK.ac.cam.cl.tlh20.UserInterface` will not be compiled at all. Other classes will be handled according to the system's default policy. Ambigu-

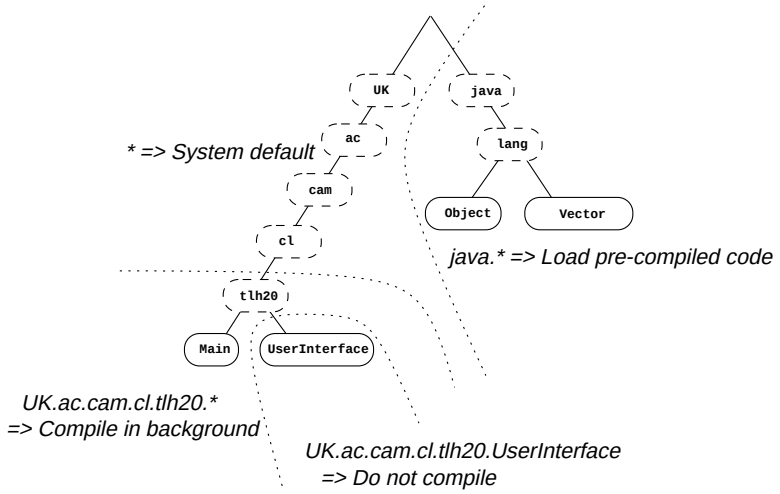

Figure 8: *Dispatchers* are used to control how different sections of the package hierarchy are executed. In this example, standard classes are loaded from pre-compiled versions, part of an application is compiled in the background and another part will not be compiled at all.

ity is resolved by selecting the most specific match.

The implementation of a dispatcher can be very straightforward, for example figure 9 shows two complete dispatchers. The first eagerly compiles methods as soon as they are invoked. The second leaves the methods to be interpreted.

4.2.2 Dispatcher lookup

Dispatcher lookup is implemented efficiently by caching lookup results.

A 32-bit sequence number is associated with the current mapping from the package namespace to dispatchers. This sequence number is increased whenever the mapping could potentially change – ie whenever a new dispatcher is registered. A record is maintained in each class structure of the last cached lookup and the then-current sequence number. A full lookup operation is only performed if the recorded sequence number is stale.

Figure 10 shows the time taken by lookup operations. These results were measured during the execution of the `javac` Java-to-bytecode compiler on an 800-line program.

4.2.3 Background compilation

By arranging that compilation happens in designated *compiler threads* it is possible to bound the impact that compilation can have on the progress made by an application.

This approach relies on using a thread's CPU allocation as an upper limit on the resource that it may consume and therefore on the impact that it may have on other concurrently executing tasks. For example it is possible to allocate some percentage of the CPU to

```
/**
 * A dispatcher which causes a method to be
 * compiled the first time that it is executed.
 * The compilation occurs within the thread which
 * invoked the method.
 */
public class JITDispatcher extends Dispatcher
{
  private static Compiler compiler = new Compiler ();

  public final void dispatchMethodImpl (KCMethod m)
         throws DispatcherException
  {
    compiler.compileMethod (new CompilableMethod (m));
  }
}

/**
 * A 'no-op' dispatcher.  The JVM will execute
 * the method in its usual way.
 */
public class NullDispatcher extends Dispatcher
{
  public final void dispatchMethodImpl (KCMethod m)
         throws DispatcherException
  {
  }
}
```

Figure 9: The Java source code for two dispatcher classes – `JITDispatcher` compiles methods upon their first invocation whereas `NullDispatcher` leaves methods to be interpreted.

compilation and a separate percentage to the interpreter. This control, coupled with fine-grained thread switching, means that a user will simply see their application executing slowly during compilation, rather than stopping completely.

This approach is possible as a consequence of the thread scheduler described in section 3 – it simply cannot be achieved with the normal priority-based scheme.

It is possible to have more than one compiler thread – for example one per application – so that the resources used during compilation can be attributed to the application that requested it.

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

This trade-off is illustrated in figures 11 and 12 which compare JIT compilation and

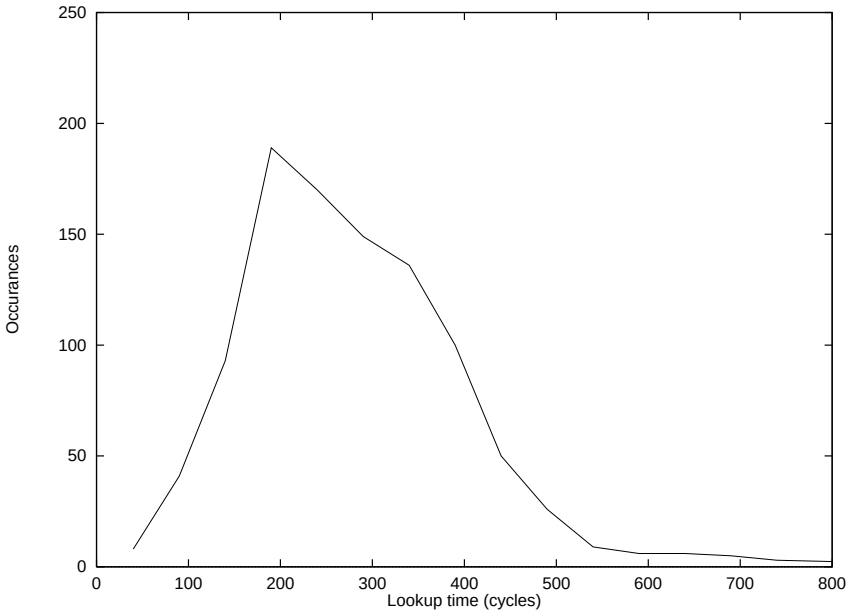

Figure 10: Time taken to lookup the dispatcher associated with a method.

interpreted execution against background compilation with a 5%, 20%, 30%, 50% or 75% CPU allocation to the compiler. The JVM as a whole had an 80% allocation of the CPU.

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

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

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

4.3 Related work

The technique of run-time generation of native code has been widely used as a means of improving the performance of an interpreter.

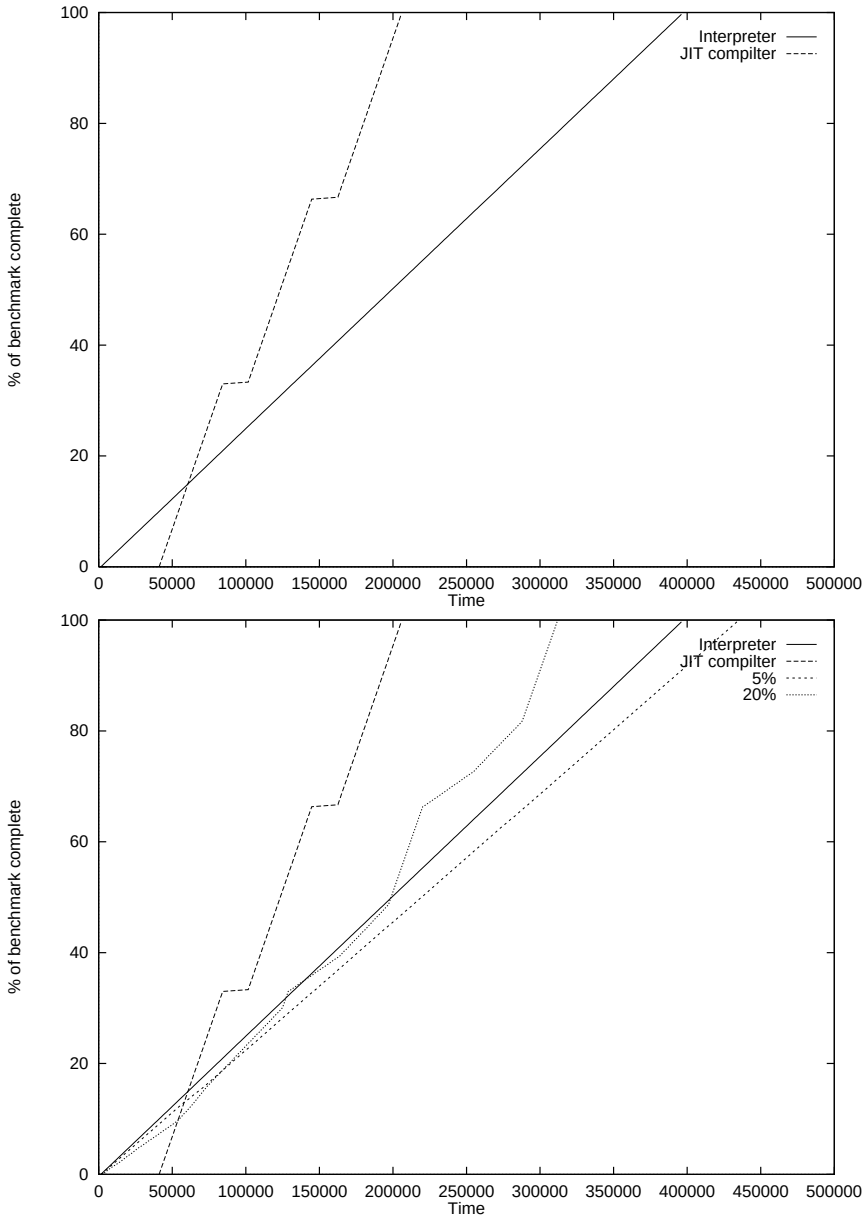

Figure 11: JIT compilation and interpreted execution (top), background compilation with 5% and 20% CPU allocation (bottom).

The implementation of a more aggressively optimizing Intel x86 Java JIT compiler is described in [ATCL+98]. This employs a similar approach as section 4.1.3 to lazy code generation, but also integrates a limited form of common sub-expression elimination.

A potentially more effective register allocation policy is used, in which seven of 32-bit

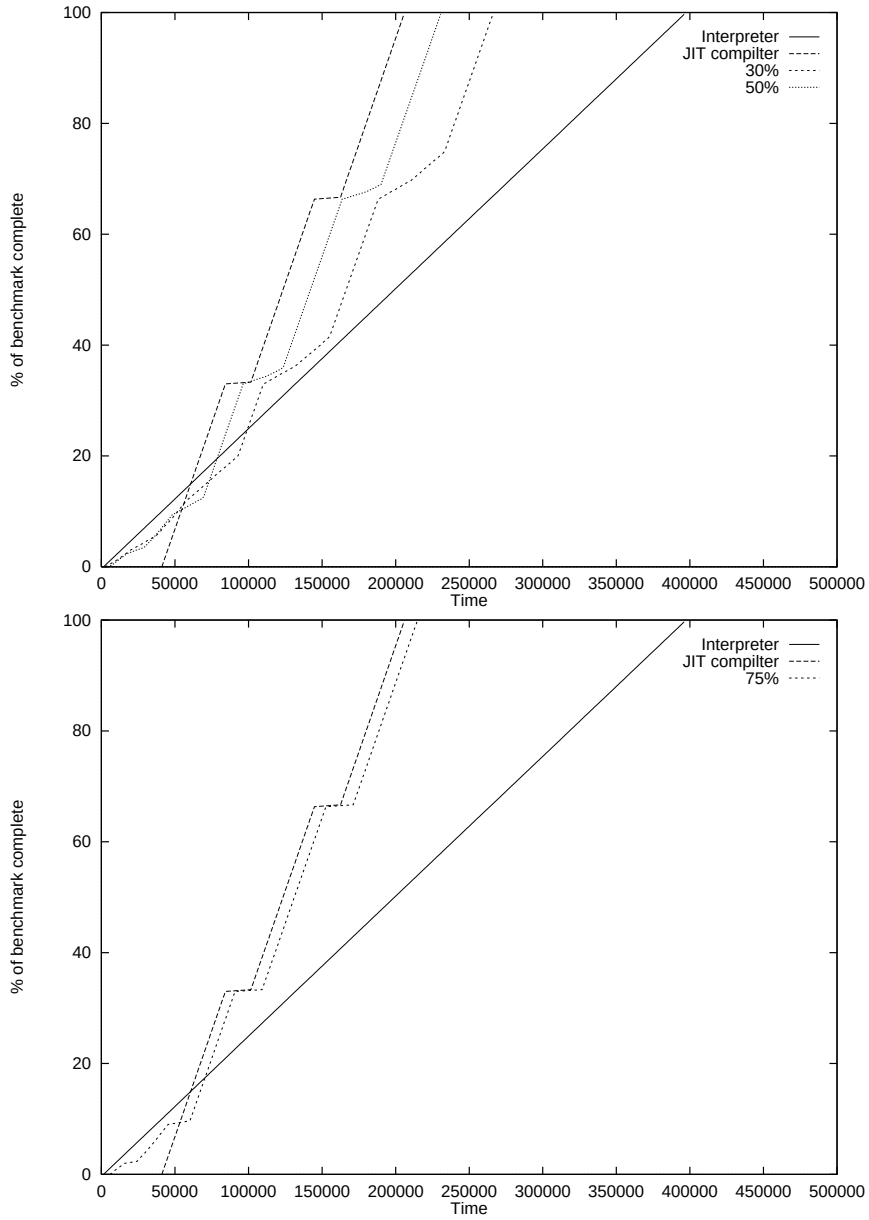

Figure 12: Background compilation with 30% and 50% allocation (top), and 75% allocation (bottom). The JVM as a whole was allocated 80% CPU time.

registers are available for general use in the generated code. It is not clear to what extent compiled and non-compiled code can interoperate (enabling this was one of the motivations for the simple register allocator of section 4.1.2). The impact of compilation on interactive performance is not analyzed.

The Deutsch-Shiffman Smalltalk-80 implementation [DS84] uses run-time code generation to achieve acceptable performance on conventional hardware (at the time meaning not user-microprogrammable). The generated code is cached in physical memory since the time taken reloading code after it was paged out was perceived to be larger than the time taken to regenerate it.

Consistency between source and generated code is checked at the start of each method body. Some aspects of the Smalltalk language make it more amenable than Java to run-time compilation. In particular only an object's methods are able to access its fields directly. This provides the implementor with a great deal of flexibility in designing an object's internal representation.

Many of their observations are still relevent – for example the convenience of a stack-based intermediate code for generating code quickly, but the need to convert to a register based form when generating optimized code.

The issue of how to control compile-induced pauses in execution is not addressed. This is perhaps a consequence of the necessity of run-time compilation for acceptable performance (many of the initial interpreted implementations of the Smalltalk-80 virtual machine [GR83] were found to be intolerably slow [Kra84]).

Self [CUL89] is a dynamically-typed pure prototype-based object-oriented language. It encourages a style of programming in which message send operations are extremely frequent and, as with Smalltalk, it is designed to provide an exploratory programming environment.

It has traditionally been implemented using dynamic compilation and *dependency links* between source and compiled methods [HCU92, CU91, HU94].

Particular attention is paid to optimizing message passing and to avoiding intrusive pauses during compilation [Höl94]. Polymorphic inline caching is used to implement message sends efficiently and to provide type feedback information to guide inlining and compilation. Optimization is performed adaptively and only on heavily-used methods.

These techniques are applied to Java in the Pep compiler, [Age97]. Pep executes Java applications by automatically converting them to Self bytecodes and then re-using the implementation of the Self virtual machine.

Future releases of the JDK are believed to use a more integrated implementation of the same technique [Gri98].

5 Garbage collection

As described in section 1.1, the third important aspect of this implementation is the garbage collector. The standard JDK [YLJ97] uses a collector which is:

- *Partially conservative*, meaning that it cannot always distinguish between pointer and non-pointer values. All word-aligned 32-bit quantities on the Java and native stacks are assumed to be pointers if their values are plausible. Implementing a non-conservative collector for the JVM involves at least a non-trivial analysis to determine whether Java stack locations hold references or scalar quantities [ADM98].
- *Optionally compacting*, meaning that the collector can compact regions of free space in the heap. This is usually achieved by copying live objects so that the free regions become adjacent and can be coalesced. The extent to which this is possible is limited by the use of a partially conservative collector.
- *Stop and copy*, with the consequence that the JVM must stop for a complete collection cycle to occur if memory is to be recovered.

Furthermore, the `System` and `Runtime` classes provide methods to trigger a collection explicitly. This can cause extensive cross-talk between threads - even if a thread has been guaranteed resources by the scheduler, it may be prevented from running because another thread has triggered a synchronous collection.

The implementation I have developed addresses this issue by providing a concurrent collector, allowing threads to continue executing during garbage collection. As with the existing collector, the new implementation is partially conservative.

The standard terminology of tri-colour marking [Wil92] is used throughout this section. Essentially, a *white* object is one which has not been scanned, a *gray* object is one which needs to be scanned and a *black* object is one which has been scanned. As with most collection algorithms, the one described here can be divided into three phases:

1. Constructing the *root set* of objects. These are objects which are certainly reachable (for example because there is a reference to them in the native stack). They form the initial set of gray objects.
2. Recursively scanning the gray objects for references. Any referenced objects are grayed and will themselves be scanned. The scanning algorithm is described in section 5.1
3. When there are no more gray objects to scan, merging the storage allocated to any remaining white objects with the free list. When the scanning algorithm terminates it will have ensured that no gray objects remain and that all reachable objects are black.

5.1 Baker's treadmill collector

Baker's treadmill [Bak91] is essentially a non-copying implementation of the traditional two-space garbage collector [Bro84]. It was originally proposed to avoid the

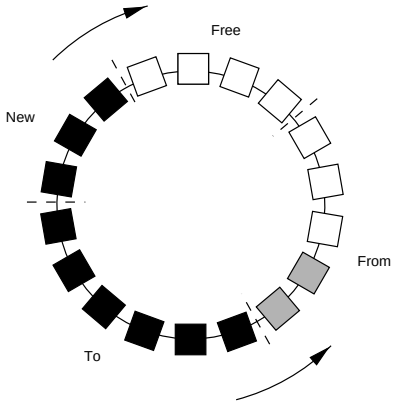

Figure 13: Baker's treadmill collector. As objects are allocated the boundary between the *new* and *free* regions circulates clockwise. As objects are scanned the boundary between the *to* and *from* regions circulates counter-clockwise. If memory is not to be exhausted then the two boundaries must be prevented from colliding.

repeated and often unnecessary copying of all live objects during each collection cycle.

Objects are arranged in a circle within which they are grouped into four non-overlapping sections:

1. *Free* section, containing blocks which are available for allocation.
2. *New* section, containing objects which have been allocated since the start of the current collection cycle.
3. *To* section, containing objects which were allocated before the start of the current collection cycle and which must be preserved beyond the end of the cycle.
4. *From* section, containing objects which were allocated before the start of the current collection cycle.

This organization is shown in figure 13 in which a collection cycle is partially complete – there are two gray objects left to be scanned and three white objects which are potentially garbage. At the start of a collection cycle, the *new* and *to* sections are empty and the *from* section contains all of the allocated objects.

Each instance has two components: an object handle and a field table. This is illustrated in figure 14. It differs from the standard JDK implementation [YLJ97] in that a 64-bit header on the field table is replaced by the 32-bit *next* and *prev* pointers in the handle.

5.1.1 Mark bits

Two *mark bits* bits are reserved in each object handle to store the object's current colour. These are stolen from the bottom two bits of the *prev* pointer since it is constrained to point to a word-aligned address.

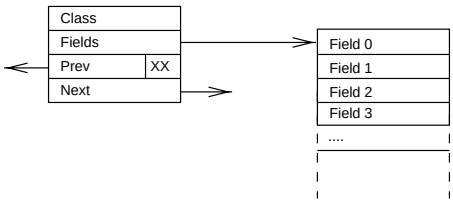

Figure 14: The representation of objects within the JVM. An *object handle* (left) contains pointers to the class structure and to the field table. The method table is reachable from the *class* structure. The *next* and *prev* pointers are used to organize all objects into a doubly linked circular list. The bottom two bits (*XX*) of the *prev* pointer are used as mark bits to hold the object's colour.

Value	Sense 1	Sense 2
00	White	Black
01	Gray	Black
10	Black	White
11	Black	Gray

Table 2: The two possible interpretations for an object's mark bits. The sense changes between each collection cycle.

There are two possible mappings from mark bits to colours, as shown in table 2. The mapping changes between each collection cycle so that the sweep operation (see section 5.3) can be implemented trivially.

These encodings allow objects to be grayed in an identical way, irrespective of the current mapping. This means that the same write barrier (see section 5.1.2) can be used in each case.

5.1.2 Write barrier

As described in [DLM+74], it is necessary for the collector and mutator threads to co-operate to ensure that only unreachable objects are collected. The problem is essentially that the mutator may move the only reference to a white object into one which the collector has already scanned. This would prevent the white object from being marked as reachable and it could be collected.

The approach taken here is to ensure that, if an object was reachable from the root-set at the start of a collection cycle, then at all times during the cycle it will be either:

1. Blackened and in the *to* section.
2. Reachable from a gray object in the *from section*.

In particular, when the collection cycle is complete, this requires that all objects which were reachable from the root-set are now blackened and in the *to* section. Note that new objects are allocated black and placed in the *new* section.

```
  testl %edx, %edx     ; was old value NULL?
  jz    wb_done
  orl   8(%edx), 1     ; mark old value gray
wb_done:
  ...
```

Figure 15: The implementation of the write barrier.

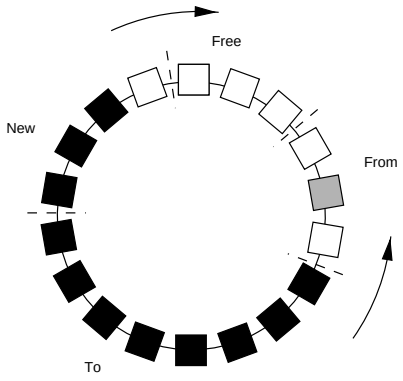

Figure 16: Objects which have been grayed by the write barrier may exist anywhere in the *from* section. These are moved to the boundary between *to* and *from* sections and are then scanned as normal.

This invariant is enforced by introducing a *write barrier* which grays any reference that is overwritten by an `aastore`, `putfield` or `putstatic` bytecode operation. This ensures that, if an object would otherwise cease to be reachable from the gray objects in the *from* section then it is itself grayed and will be scanned.

The write barrier is implemented efficiently by a single comparison and `or` operation (see figure 15). The comparison is unfortunately necessary to trap situations where a `Null` reference is overwritten. The `orl` operation sets the least significant bit in the *prev* pointer in the object's handle (see figure 14 and table 2).

This implementation requires a further modification to the original treadmill algorithm. This is because there may be objects which have been grayed by the write barrier and are left 'floating' in the *from* section. The original algorithm would stop when the boundary between the *to* and *from* sections reaches white objects. This means that cases such as that shown in figure 16 would leave some gray objects un-scanned.

The change required is to make a pass over the remaining *from* section when the collection cycle appears to be complete. Any gray objects that are found there (such as the one in figure 16) are moved to the boundary between the *to* and *from* regions. These objects are then scanned in the normal way.

Note that the process of examining the remaining *from* section only needs to be performed once. This is a consequence of the invariant described above.

5.2 Finalizers

A Java class may define a *finalizer* method. This method is executed when an instance of the class has been detected as unreachable. Finalizers are typically used when an object is associated with a corresponding resource in the run-time system - for example an instance of a Java class which represents open files may need to close the underlying operating system file descriptor. However, there are no special restrictions on the operations which a finalizer may perform. For example it can 'resurrect' the object by making it reachable again. It could also access other objects which are only reachable through the finalizable one. This life-cycle is explained in more detail in [GJS97].

There are two consequences for the implementation being described here: detecting finalizable objects and accounting the resources used by executing their finalizers.

5.2.1 Detecting finalizable objects

The collector must identify objects which are not reachable, cause their finalizers to be executed, and then if they remain unreachable, allow their space to be re-used.

The approach taken is to treat potentially finalizable objects as *white roots* which are treated as normal at the start of a collection cycle, except that they are initially coloured white rather than gray. These *white roots* remain white after scanning, but if one is reachable from another object then it is grayed as normal.

When the scanning phase is complete note that any finalizable objects are in the *to* space and their storage is therefore safe from re-use. They can be discovered by examining the mark bits - any reachable objects with finalizers will have been grayed and any finalizable objects will have remained white.

5.2.2 Accounting finalizers' CPU usage

Another issue, which is particularly pertinent in this implementation, is how the resources used by a finalizer are accounted. The JLS gives the implementation freedom to choose which thread executes the finalizer.

It is tempting to dismiss finalizer execution as a control-path operation and accept the (hopefully small) amount of cross-talk that may be introduced by a single shared finalizer-executing thread. This fits with the intention that finalizers are only used to free system resources which are not managed automatically [GJS97].

However, the unrestricted nature of finalizers' behaviour can make this unacceptable. For example, the following Java source code is legal:

```
void finalize()
{
  while (true)
  {
  }
}
```

If this is executed in a common finalizer-executing thread then it will loop continuosly and prevent any other object's finalizers from being executed. Sun's *Java Developer Connection* bug-tracking system (`http://developer.javasoft.com`) documents this problem, and a related one in which a finalizer containing a synchronized region can cause a deadlock.

The approach taken here is to use a separate finalizer-executing thread for each class which has been instantiated and whose instances require finalization. This situation is clearly not ideal since a particular class may be accessed by threads which form part of more than one application. It does, however, segregate the finalization of completely unrelated classes – such as those that have been loaded by different class loaders.

5.3 Sweeping unreachable objects

When a collection cycle is complete, the *from* region will consist solely of white unreachable objects. This is the same situation as in a traditional two-space copying collector, in which a collection cycle is finished when all of the reachable objects have been evacuated from *from-space*.

Merging unreachable objects with the free list is trivially achieved as a consequence of holding the objects on a doubly-linked list. It is similarly possible to move the objects from the old *new* and *to* regions to form the new *from* region for the start of the next collection cycle.

It is also necessary to change the colours of the objects to reflect their new status. This is achieved by changing the meaning of the mark bits rather than by adjusting the values on each object.

5.4 Results

The standard JDK garbage collector runs when an object allocation cannot be satisfied by the remaining free space, when it is explicitly triggered through `System.gc()` or `Runtime.gc()` and as a low-priority thread that may run during idle time [YLJ97]. While it is possible to use this new collector in the same manner, it is also possible to cause it to run continuously in a similar way to the background compiler (as described in section 4.2.3).

Figure 17 illustrates some results from this approach. These results were obtained with a fixed 50% CPU allocation given to a mutator thread and various allocations provided to a concurrent collector thread. The mutator repeatedly compiled and re-compiled a piece of bytecode to native code using the compiler described in section 4. In these experiments, the collector was always allocated sufficient time to avoid the mutator blocking or the heap being expanded. This accounts for the comparatively high CPU allocations which were made to the collector thread. It is, of course, possible to use a much-reduced CPU allocation at the risk of blocking a thread during an allocation.

5.5 Related work

There have been numerous papers on automatic storage management. [Wil92] contains a survey of garbage collection techniques, including those that are suitable for real-time use. [WJNB95] contains a similar survey of allocation techniques.

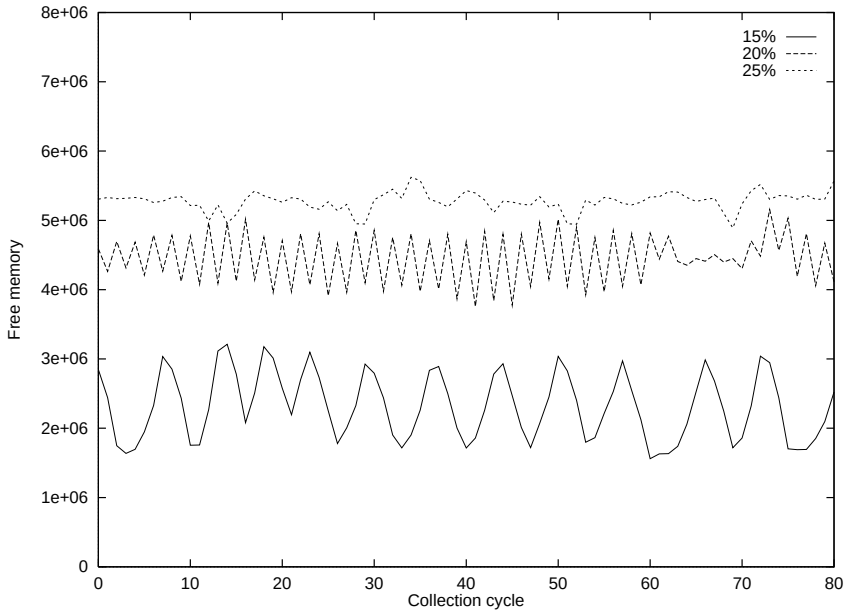

Figure 17: The trade-off between CPU allocated to the garbage collector and the amount of free memory at the end of each allocation cycle.

As with the treadmill collector, many of the existing approaches cannot be used unmodified as a consequence of the requirement to support finalization and the freedom with which native code may access heap-allocated objects.

[Ben97] describes the memory manager for the Aurora JVM which supports persistent as well as 'traditional' objects. It uses a hybrid collector which employs different techniques for the different object memories that it supports, including a mostly-copying generational collector for new objects and a mark/sweep collector for objects which have been tenured after surviving multiple generations.

The Kaffe JVM [kaf97] uses a conservative incremental collector which maintains explicit lists of white, gray and black objects and uses a conceptually similar write-barrier to the one described above. However, the implementation of the write-barrier is more complex because it must test the colours of the two objects and potentially move an object to the gray list.

An approach used elsewhere [Hen96] is to perform a small amount of collection work whenever an allocation is made. If the maximum quantity of simultaneously live memory is known then the amount of collection work, in terms of the number of words to scan, can be related to the allocation size. This technique has the consequence that threads which perform frequent and large allocations will contribute most to collecting garbage. Conversely this approach raises some new problems. Firstly, if an application has idle time then it may be more appropriate to perform garbage collection at that point. Secondly, if cross-talk is to be avoided, then multiple threads must be able to invoke the collector concurrently.

6 Conclusion

This report has described techniques that have been employed to produce an implementation of the JVM with improved support for interactive applications and improved control over how its resources are used.

A new thread scheduler has been implemented (section 3) which allows CPU time to be shared between applications in a more flexible way than traditional priority-based approaches. A native-code generator has been implemented (section 4), as have mechanisms which can be used to control how it is deployed at run-time. Finally, the existing stop-and-copy garbage collector has been replaced by one that can operate concurrently (section 5).

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *ACM SIGPLAN Notices*, 33(5):269–279, May 1998.

[Age97] Ole Agesen. Design and implementation of Pep, a Java just-in-time translator. *Theory and Practice of Object Sytems*, 3(2):127–155, 1997.

[And98] Mark Andrews. Introducing swing. *The Swing Connection*, 2(6), May 1998. `http://www.javasoft.com/products/jfc/swingdoc-static/what_is_swing.html%` .

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[ATCL+98] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.

[Bak91] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Position paper. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992.

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

[Ben97] Peter Benson. The memory manager for the Aurora Java Virtual Machine testbed. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[BK97] Z. Budlimic and K. Kennedy. Optimizing Java: theory and practice. *Concurrency - Practice and Experience*, 9(6):445–464, June 1997.

[Bla94] Richard Black. Explicit Network Scheduling. Technical Report 361, University of Cambridge Computer Laboratory, December 1994. Ph.D. Dissertation.

[Bot97] Per Bothner. A gcc-based Java implementation. In *IEEE Compcon*, February 1997. `ftp://ftp.cygnus.com/pub/bothner/gcc-java.ps.gz`.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262. ACM, ACM, August 1984.

[CFM+97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, May 1997.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical l. In Andreas Paepcke, editor, *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 1–15. ACM Press, October 1991.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, October 1989.

[DLM+74] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Language hierarchies and interfaces*, volume 46 of *Lecture Notes in Computer Science*. Springer Verlag, 1974.

[DMH] James K Doyle, J Elliot B Moss, and Antony L Hosking. When are bytecodes faster than direct execution? Submitted for publication.

[DS84] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302. ACM, ACM, January 1984.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[Gos95] J. Gosling. Java intermediate bytecodes. *ACM SIGPLAN Notices*, 30(3):111–118, March 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gri98] David Griswold. The Java HotSpot virtual machine architecture. March 1998. `http://www.javasoft.com/products/hotspot/whitepaper.html`.

[HAKN97] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. Technical Report ICS-TR-97-01, University of California, Irvine, Department of Information and Computer Science, April 1997.

[Har97] Tim Harris. A just-in-time Java bytecode compiler, part 2 project dissertation, May 1997. University of Cambridge Computer Laboratory. `http://www.cl.cam.ac.uk/users/tlh20/dissertation.ps.gz`.

[Har98] Tim Harris. Thesis proposal. July 1998. University of Cambridge Computer Laboratory.

[HCU92] Urs Hoelzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Notices*, 27(7):32–43, 1992.

[Hen96] Roger Henriksson. Adaptive scheduling of incremental copying garbage collection for interactive applications. Technical Report 96–174, Lund University, Sweden, 1996.

[HF97] Richard Hayton and Matthew Faupel. FlexiNet – automating application deployment and evolution. *Workshop on Compositional Software Architectures*, December 1997. `http://www.objs.com/workshops/ws9801/papers/paper025.html`.

[HG98] Tom R Halfhill and Al Gallant. How to soup up Java. *Byte*, May 1998.

[Höl94] Urs Hölzle. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. Thesi CS-TR-94-1520, Stanford University, Department of Computer Science, August 1994.

[HU94] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. *ACM SIGPLAN Notices*, 29(6):326–336, June 1994.

[int96] *Pentium Pro Family Developer's Manual, volume 2, programmer's reference manual*. Intel Corporation, 1996. Order number 242691-001.

[kaf97] Kaffe source code, version 0.9.2. October 1997. http://www.kaffe.org.

[Kra84] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, 1984.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MMAY95] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. *ACM SIGPLAN Notices*, pages 300–315, October 1995.

[Nil98] Kelvin Nilsen. Adding real-time capabilities to Java. *Communications of the ACM*, 41(6):49–56, June 1998.

[NL97] Kelvin Nilsen and Steve Lee. PERC real-time API. July 1997. NewMonics, Inc.

[Opa98] Don Oparah. Adaptive resource management in a multimedia operating system. In *NOSSDAV '98: 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 91–94. University of Cambridge Computer Laboratory, July 1998.

[Ros95] Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical Report 376, University of Cambridge Computer Laboratory, August 1995.

[sun97] Java on Solaris 2.6, a white paper. September 1997. `http://www.sun.com/solaris/java/wp-java/`.

[sun98] picoJava-II data sheet. (Sun microsystems document number 805-4634-01), April 1998.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 September 1992. Springer-Verlag.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[YLJ97] Frank Yellin, Tim Lindholm, and JavaSoft. Java runtime internals. In *JavaOne Worldwide Java Developer Conference*, 1997.