Controlling run-time compilation

Timothy Harris, APM Ltd*, tim.harris@citrix.com

Poseidon House, Castle Park, Cambridge, CB3 0RD.

Tel: +44 1223 515010. Fax: +44 1223 359779.

November 13, 1998

Keywords: Java implementation, just-in-time compilation,
run-time compilation, controllable compilation, real-time systems.

Abstract

This paper describes a technique for integrating run-time compilation which is effectively *pause free* and for which the worst-case impact can be bounded. Three extensions to a JVM implementation are used. Firstly, a new scheduler allows the allocation of CPU time to threads to be controlled. Secondly, a code generator provides a mechanism for run-time compilation. Finally, a control interface allow application-specific compilation policies to be specified. By defining a compilation policy in which native code is generated in a designated *compiler thread* with a limited CPU allocation, it is possible to bound the worst-case impact of the compiler.

1 Introduction

The widespread deployment of the Java language is curtailed by the performance of existing implementations of the Java Virtual Machine (JVM, [LY97]). Although there are numerous and well known techniques for optimizing object-oriented programming languages, the extent to which these can be applied within the JVM is limited by the need to perform this optimization at run-time and the trade-off between start-up latency and subsequent performance.

In this paper the JVM is assumed to be executing applications loaded from standard *class* files (as opposed, for example, to pre-compiled or *fat binary* files). This assumption precludes the use of ahead-

*APM Ltd is a subsidiary of Citrix Systems Inc

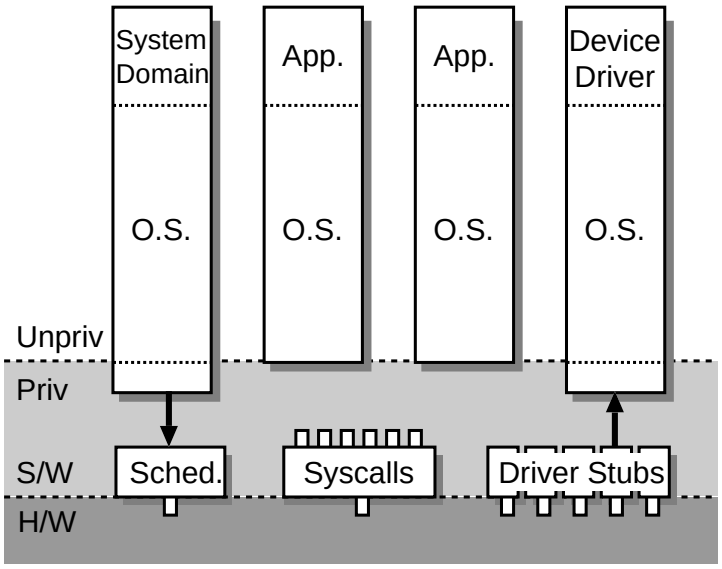

Figure 1: The structure of the Nemesis operating system.

of-time compilation since the complete set of classes which comprise a partiticular application is known only at run-time. There are two key motivations for providing of a controlable framework for run-time compilation – either an increase in the performance of an application, or a reduction in the resources required by an application for it to run at a particular rate.

This paper introduces extensions to a JVM implementation which increase control over *when* and *where* run-time compilation occurs. Section 2 describes the environment in which this work has taken place, Section 3 describes the implementation of a new thread scheduler, Section 4 describes the compiler and Section 5 the compilation-control interface. Section 6 discusses related work and Section 7 concludes.

2 Nemesis

This work has been undertaken over Nemesis [LMB+96], an operating system being developed to provide fine-grained resource management with specific attention to the needs of interactive soft-real-time multimedia and networked applications.

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, consider a program receiving, processing and displaying data from the network. Wherever possible the network protocol processing and screen updates are implemented within shared libraries and performed by threads within the application. In contrast, for a similar program executing over UNIX, many of these operations would occur within the kernel (and would not be accounted to any application) or the X server (again, unaccounted to the specific application).

3 Thread scheduler

The thread scheduling policy described here is broadly similar to that provided by the Nemesis process scheduler [Ros95]. The CPU requirements of a thread are expressed as a (p, s, x, t) tuple, encoding a *period*, *slice*, *extra time flag* and *priority* respectively. For example a requirement of $(10ms, 1ms, True, 1)$ represents an allocation of 1ms CPU time every 10ms of real-time and that it has priority '1' for receiving any 'extra' time that remains if all the allocations are met.

3.1 Scheduler implementation

The scheduler uses an *earliest deadline first* (EDF) policy. Each thread is assigned a deadline of the end of its current period, by which time it should have received its slice of the CPU. The scheduler conceptually maintains three data structures:

1. A list containing all of the threads, irrespective of whether they are runnable or blocked.

2. A *run queue* containing threads which are eligible for execution. This excludes threads which are not runnable, or which have already expended their CPU allocation and do not wish to receive a share of extra time. Threads on the run queue are ordered (1) the thread currently inside a critical section, (2) threads with some remaining CPU allocation, held in EDF order and (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).

3. An *allocation queue* containing threads which have a non-zero CPU allocation, ordered according to the start of their next period.

When activated, the thread scheduler performs three tasks. Firstly, it charges the previously executing thread with the CPU time that it has just received. Secondly, it allocates new slices of CPU time to any threads which have reached the start of their next period. Finally, if there are any threads eligible to be

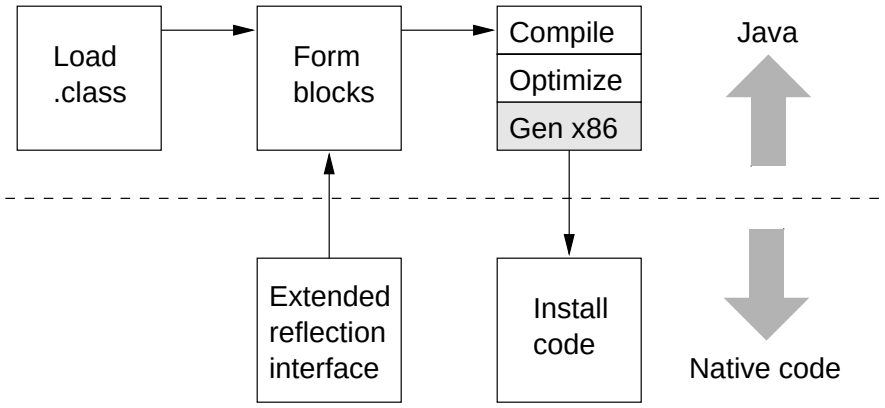

Figure 2: The structure of the compiler.

executed, it selects the thread at the head of the run queue and sets an alarm timer for when any other thread will obtain an earlier deadline.

4 Native code generator

The compiler used here is relatively straightforward since the motivation is to provide a controllable implementation rather than to establish new optimization techniques. The structure of the compiler is shown in Figure 2. Compilation is a three phase process. Firstly, the bytecode implementation from the `.class` file is broken into basic blocks and the contents of the constant pool are resolved. Secondly, the code generator produces native code for each basic block. Finally, the generated code is installed so that it will be executed if the method is invoked in the future.

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

The platform-independent section generates RISC-like 3-address-code directly from the Java bytecode. The optimizer, if present, maintains details of outstanding assignment instructions which have yet to be passed to the target-dependent code generator. These mappings are used to rename operands on subsequent instructions in an attempt to reduce the frequency of memory accesses. The technique is essentially the same as that described in [ATCL+98]. Forward branches are handled by back-patching the generated code when the target address becomes known.

Test	Without optimizer	With optimizer
Add integer	2.9	10.0
Add float	1.2	1.2
Cast integer to byte	6.5	10.5
Cast integer to float	1.5	1.6
Method call	2.5	2.8
Method call (4 arguments)	2.7	2.9
Static method call	1.1	1.2
Interface method call	1.4	1.5
Enter and leave monitor	1.2	1.3
New array	5.7	6.6
New instance	0.8	0.8

Figure 3: Microbenchmarks showing the speed-up of individual operations relative to the original JDK 1.1.4 interpreter.

4.1 Results

Figure 3 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.

5 Control of compilation

Compilation is controlled by *dispatchers* which are associated with particular sub-trees of the package hierarchy and on which `dispatchMethodImpl` is invoked whenever a method in the sub-tree is called for the first time. The mapping from package names to dispatchers is maintained by static methods of `DispatcherRegistry`. This provides a mechanism for introducing class-specific processing into standard method invocations in a similar manner to the meta-class facilities provided in other object-oriented languages such as Smalltalk or extended dialects of C++.

Figure 4 illustrates how dispatchers may be used to implement a particular execution policy. The standard '`java.*`' classes are registered with a dispatcher which loads a pre-compiled implementation (perhaps one generated off-line with a highly optimizing compiler). Classes whose names begin `UK.ac.cam.cl.tlh20.*` will be compiled in the background (see section 5.1). 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. Ambiguity is resolved by selecting the most specific match. The

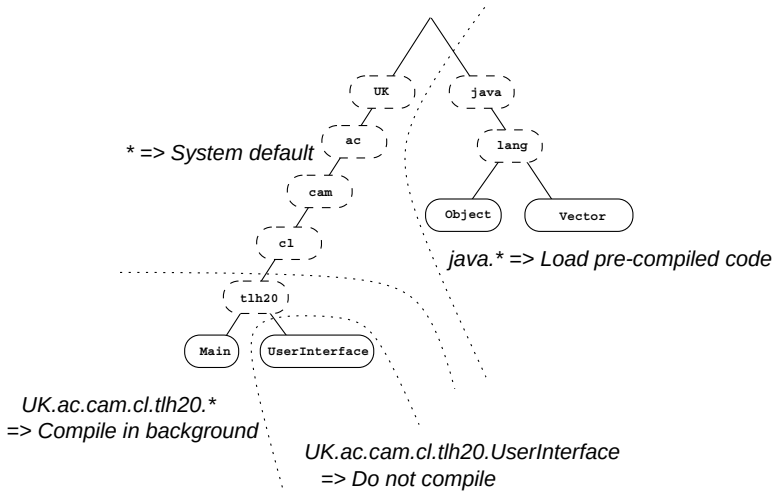

Figure 4: *Dispatchers* are used to control how different sections of the package hierarchy are executed. In this example, standard classes are loaded from pre-compiled versions, part of an application is compiled in the background and another part will not be compiled at all.

```
public class JITDispatcher extends Dispatcher {
  private static Compiler compiler = new Compiler ();
  public void dispatchMethodImpl (KCMethod m)
         throws DispatcherException {
    compiler.compileMethod (new CompilableMethod (m));
  }
}
```

Figure 5: The Java source code for a dispatcher which implements a 'just in time' compilation policy.

implementation of a dispatcher can be very straightforward. Figure 5 shows the complete source code for a dispatcher which eagerly compiles methods as soon as they are invoked.

Dispatcher lookup is implemented efficiently by caching lookup results on a per-class basis. A 32-bit sequence number is associated with the current mapping from the package name-space to dispatchers. This sequence number is increased whenever the mapping could potentially change and a full lookup operation is only performed if the sequence number recorded in a class is stale.

5.1 Background compilation

By arranging that compilation happens in designated *compiler threads* it is possible to bound the worst-case effect 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

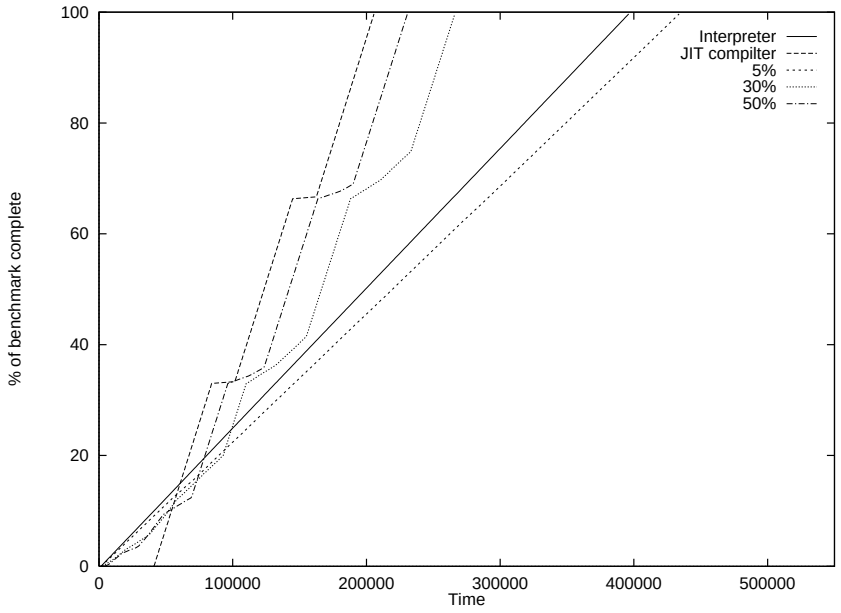

Figure 6: JIT compilation, interpreted execution and background compilation with 5%, 30% and 50% CPU allocations.

the impact that it may have on other concurrently executing tasks. For example it is possible to allocate some percentage of the CPU to compilation and a separate percentage to the interpreter. This control, coupled with fine-grained thread switching, means that a user will simply see their application executing slowly during compilation, rather than stopping completely.

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 without any allocation corresponds to compiling during idle time whereas a 100% allocation provides JIT compilation. This technique is possible as a consequence of the thread scheduler described in section 3 and demonstrates the value of this scheduling policy over the normal priority-based scheme.

This trade-off is illustrated in Figure 6 which compares JIT compilation and interpreted execution against background compilation with a 5%, 30% and 50% CPU allocation to the compiler. The JVM as a whole had a 70% 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 steeper line with some abrupt steps. 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

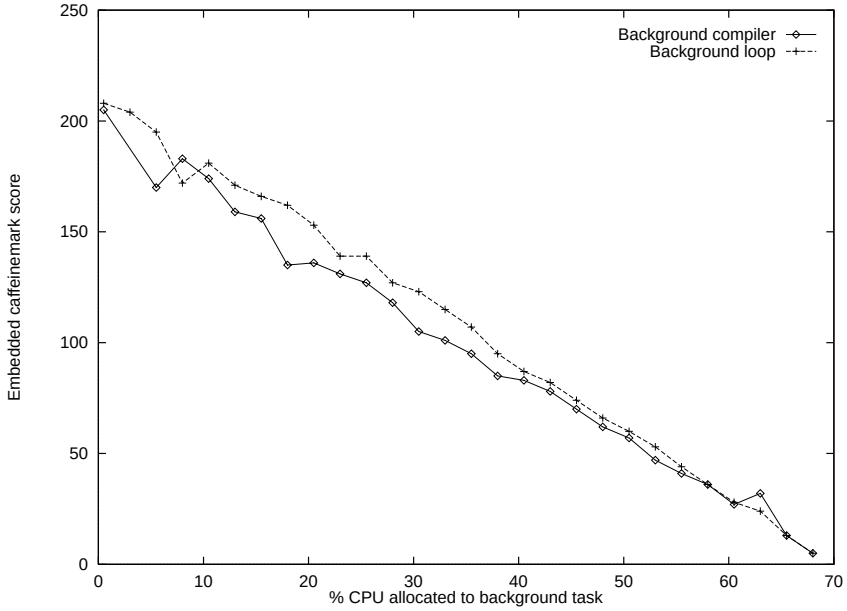

Figure 7: Degradation of run-time performance due to background compilation.

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 because the compiler is operating slowly and fails to finish compiling methods before execution shifts to another part of the benchmark. As the allocation is increased the trace approaches that of the JIT compiler.

5.1.1 Crosstalk between compilation and execution

An obvious concern with this approach is the extent to which background compilation may interfere with the execution of the application – that is, whether the worst-case impact is solely the proportion of the CPU time allocated to compilation, or whether significant degradation is caused by increased contention in the processor caches or perhaps for mutual-exclusion locks within the implementation of the JVM.

Figure 7 compares the performance of a Java application while a percentage of the CPU is allocated to either background compilation or to a busy loop. These results indicate that background compilation causes a slight additional degradation of performance, although the relationship between performance and the CPU allocated to the foreground task remains approximately linear. The performance figures were recorded using Version 2 of the CaffeineMark embedded application Java benchmark.

6 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. The Deutsch-Shiffman Smalltalk-80 implementation [DS84] uses run-time code generation to achieve acceptable performance on conventional hardware. 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.

Self [CUL89] is a dynamically-typed pure prototype-based object-oriented language in which message sends are extremely frequent. 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].

The NewMonics PERC system is a dialect of Java that is '*designed to support development of cost-effective portable real-time software components*' [NL97]. 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 (i.e. the maximum amount by which the time that they receive their allocation of the processor may vary from period to period).

7 Conclusion

This paper has described the integration of a controllable native code generator and a new thread scheduler with the JVM. Although the compiler presented here is straightforward, the technique of *background*

compilation could be used to integrate more 'heavyweight' run-time optimization while allowing the worst-case impact of the compiler to be bounded and the responsiveness of interactive applications to be maintained.

References

[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.

[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.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. 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.

[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.

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

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

[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.

[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.

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

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