CS Report 01/2001
Proxy Compilation*

Matt Newsome and Des Watson †
{mattn,desw}@cogs.susx.ac.uk

January 2001

Abstract

In this paper, we outline new research concerning dynamic compilation of Java applications in environments where system resources are significantly limited. In such environments, which include "smart" mobile telephones and Personal Digital Assistants, memory and processor cycles can be scarce, making current techniques for the runtime translation of Java programs or program fragments inappropriate. We propose an alternative technique, *proxy compilation*, which makes use of idle, connected devices on a network to compile code on its behalf.

1 Introduction

The Java programming language[10, 21], although commonly associated with Internet programming, is a general-purpose object-oriented programming language. Many of its features, such as the use of a garbage-collected memory allocation scheme, a virtual machine execution model and single class inheritance, have been highly lauded within the computer programming industry and academia.

The traditional role of the compiler[1, 13, 42, 23] has been to facilitate one-time translation of human-readable source programs into machine-readable object programs. The generated software is then suitable for deployment on a particular computer, where the program is run. Typically this means synthesis of the program by the machine's actual processor. In the Java model, however, a compiler is primarily used to convert Java source code, which is written directly by the programmer, to the Java Virtual Machine (JVM) classfile format[21, 6]. Virtual machines (VMs) are hypothetical processors implemented in software. Execution of a program for such a machine typically requires translation of the VM program to one which can be directly executed by the underlying processor (often that on which the virtual machine software is executing), while retaining functional equivalence. In this case, the VM language is being used as an intermediate representation in the implementation of the language. Alternatively, the VM program can be interpreted by passing it through an interpreter program running on the target machine. The JVM is designed to allow convenient representation of Java programs. Such VM code is relatively efficient for transfer across networks, such as the Internet. Additionally, the program code contained in JVM classfiles (referred to hereafter as *JVM code*) can generally be translated more easily than the high-level program constructs in the originating Java programs.

Several techniques exist for the execution of JVM code. In the traditional Java model, each individual JVM instruction is interpreted each time the program is run. This process involves translation of the JVM instruction into a semantically equivalent instruction (or sequence of instructions) for the target computer to execute; hence interpretation can slow program execution, typically by an

*This work is sponsored by Hitachi Micro Systems Europe Limited
†School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, UK

order of magnitude. One common modification to this scheme, made in pursuit of faster program execution, has been to translate JVM code into native code a single time at runtime when such code is loaded (termed *Just-in-Time (JIT)* compilation[20, 18, 3, 38, 34]), or incrementally when system activity is below a specified threshold (known as *continuous compilation*[27, 28], as the program is continually replaced with more highly-optimised versions of itself). JVM JIT compilers often compile individual methods, rather than entire classes or programs, and defer compilation of each method until just prior to its execution. In both JIT and continuous compilation, a native code version of the abstract JVM code is stored and executed in place of the JVM code throughout the remainder of the program. This results in a significant execution speed increase as interpretation is no longer required.

For powerful machines, such as desktop computers or servers, JIT compilation is generally considered to be a good solution. JIT systems often introduce user-perceivable delays, however, while code is loaded and then compiled. Additionally, JIT systems are often several megabytes in size and generally consume significantly more RAM than an interpreter while executing[16].

In the context of resource-constrained products, for example, embedded consumer devices (such as Personal Digital Assistants (PDAs) and so-called "smart" mobile phones), in which memory and CPU cycles are a scarce resource, the overhead of a dynamic compiler upon the system's ROM/RAM usage and upon execution speed is often impractical. An alternative compilation scheme for Java which is gaining popularity in resource-constrained environments is *Ahead-of-Time (AOT)* compilation[33, 30, 9, 25, 24, 31, 11, 5], in which all program code is compiled off-line (i.e. before the program is run on the target computer). This is, of course, the classical model of compilation as used for ANSI C/C++[37] and many other programming languages, and can result in highly optimised machine code without the need for runtime translation.

The central problems with AOT compilation are the loss of object portability and the question of how to resolve runtime linkage of pre-compiled binaries with dynamically loaded JVM classfiles. Dynamic class loading is one of the few features which truly differentiates Java from C++, as it allows completely new sections of a program to be supplied and translated at runtime. Specific uses include the loading of Java applets into a web browser, and updates to a server application's code without interruption. The original proposal for Java AOT compilation by Proebsting et al[30], which forbade dynamic class loading, has since been refined[24]. Unfortunately, the refinement introduces an interpreter to support execution of dynamically loaded code. As discussed above, interpreters are inefficient and JIT compilers are generally impractical in resource-constrained environments.

Consequently, efficient translation/execution of dynamically loaded classes in a resource-constrained environment remains a fundamental problem in the field. This paper describes research being conducted at the University of Sussex exploring alternatives to the schemes outlined above. The goal of the work is to find more efficient execution schemes for Java programs executing in resource-constrained environments.

The remainder of the paper is organised as follows. In Section 2, an overview of Java Virtual Machine architecture is presented. Some alternatives to current Java execution/compilation schemes are then outlined in section 3; section 3.1 describes a lower-level representation for JVM code and An alternative model for dynamic compilation is presented in section 3.2. We present an overview of our current activity and intended future work in section 4. Related work is described in section 5. Finally, acknowledgements are made in section 6 and a summary is given in section 7.

2 Java Virtual Machine Architecture: An Overview

In order to support an overview of our research, a brief overview of the implementation of a Java Virtual Machine is now presented. A thorough examination of JVM architectural details is neither possible nor warranted in this paper. For detailed information, readers are directed to the JVM literature[21, 40].

2.1 Major components

The Java Virtual Machine (JVM) is an abstract machine which processes JVM classfiles. Such classfiles contain, broadly speaking, representations of the Java methods and member fields forming a single class's definition, information to support the exception mechanism and a system for representing additional class attributes. The JVM itself exists primarily to load and link classfiles into the running machine on demand (performed by the *Class Loader*), represent those classes internally by means of a number of runtime data structures and facilitate execution (a role shared between the *Execution Engine* (which is responsible for execution of JVM instructions) and the *Native Method Interface* which allows a Java program to execute non-Java code, generally ANSI C/C++.

2.2 The JVM Class Loader

Most JVM implementation require the ability to load JVM classfiles, and it is the JVM class loader's role to load referenced JVM classes which have not already been linked to the runtime system. A repository of previously loaded classes is maintained in the interests of efficiency.

Classes may be loaded implicitly for several reasons. Firstly, the *initial classfile* - the classfile containing the `public static void main(String args[])` method - must be loaded on startup. Depending on the class policy adopted by the JVM, classes referenced by this initial class may be loaded in either a *lazy* or *eager* manner. An eager class loader loads the complete transitive closure of the initial class - that is, all the classes comprising the application code - at startup[1]. Lazy class loaders wait until the first *active use* of a class before loading and linking its classfile. The first active use of a class occurs when (1) an instance of that class is created; (2) an instance of one of its subclasses is initialised; or (3) when one of its static fields is initialised[2]. Certain classes, such as java.lang.String, may be loaded implicitly by the JVM to support execution (in this case, to represent String literals). Classes may also be loaded explicitly using the java.lang.Class.forName() method in the Java API, or through the creation of a user class loader.

In either situation, the Class Loader's job is, briefly, that of *loading*, *verifying*, *preparing*,*resolving* and *initialising* a class from a JVM classfile.

Loading involves obtaining the byte array representing the Java classfile. Most often, this is either retrieved from a file store, or received across a network. A representation of the loaded class is usually then assigned to an internal database, known as the method area. The method area may contain the entire runtime representation of a class, as suggested by Venners[40], or merely the methods from loaded classes. In the latter case, supporting data, such as the constant pool, is stored in separate data structures. The *verification* of a JVM classfile is the important process of checking the structural well-formedness of the classfile and then inspecting the classfile contents to ensure the code does not attempt to perform non-permitted operations (such as executing beyond the end of a method, which might cause type safety to be circumvented or the JVM to crash)[3].

Preparation involves allocation and default initialisation of storage space for static class fields. Method tables, which speed virtual method calls, and object templates, which speed object creation, are also often created at this stage.

Initialisation involves execution of the class's class initialisation method, if defined, wherein static class fields are initialised to their user-defined initial values (if specified).

Symbolic references within a JVM classfile, such as to classes or object fields in order to reference a field's value, are only resolved to direct references (generally a direct pointer to a record in one of the JVM's internal data structures) at runtime. This process of *resolution* may occur after preparation but prior to initialisation, or more typically at some point following initialisation, but prior to

[1] In fact, if user class loaders are used, this might not be the case as some classes cannot be loaded until the appropriate Class.forName() call is evaluated

[2] Unless the static field is declared final and initialised to a compile-time constant expression[40]

[3] Note this verification step is separate to Java's security framework, which includes notions such as security policies and protection domains.

the first reference to that symbol. The delay is generally in pursuit of increased execution speed: not all symbols in a classfile will be referenced during execution, so by delaying resolution, fewer symbols may need to be resolved with less runtime overhead. Additionally, the cost of resolution is amortised over the total execution time. Many JVM implementations, including Sun Microsystems' JDK, modify JVM bytecodes which reference a resolved constant pool entry to use so-called `quick_` opcodes. These instructions ensure the direct reference rather than the symbolic reference is used in subsequent invocations of the method code.

2.3 The JVM Execution Engine

The JVM's execution engine is responsible for synthesis, in some manner, of the set of JVM classfiles representing the original Java program. The requirements for JVM startup and the semantics for each JVM instruction in the JVM instruction set have been defined both informally[21, 40] and formally[6]. Briefly, the JVM, at startup, is responsible for loading and linking the *initial class*, which is specified in some manner to the JVM.

Following successful startup, the `public static void main(String args[])` method of the initial class is invoked. The execution engine is generally responsible for method invocation and execution. Again, with concessions to brevity, JVM instructions can be broken into groups: instructions for object creation, control flow modification, value storage/loading, operand stack manipulation, type conversion, arithmetic, type inspection, array manipulation, exception handling and thread synchronisation.

Object creation involves resolution of all supertypes of the object's class, which in turn may require loading and linkage of the corresponding JVM classfiles by the class loader. Object creation itself requires allocation of memory, generally on the JVM's heap, for object fields (including supertype instance fields). These fields are initialised to default values, though may be reinitialised by a subsequently invoked class constructor. The JVM heap is garbage collected rather than explicitly deallocated.

In addition to a heap for the storage of objects, an untyped operand stack is generally used by a JVM implementation to hold intermediate values, pass parameters to methods, receive returned values from methods and support the exception mechanism. A number of instructions are available for *operand stack manipulation*, for example to duplicate values on the stack or to clear values from the stack. While a different operand stack is created with each method invocation (and persists until method return), each operand stack is held in a frame within a single, system-wide Java stack. Frames also hold additional housekeeping information for the JVM, including a predetermined number of untyped local variables slots[4].

The JVM operates on a set of primitive types which is similar to the set of types supported by the Java language itself. Generally, integral types which are represented by less than 32-bits are promoted to integers. Consequently, the JVM's *arithmetic instructions* mainly consist of instructions which operate on integer (32-bit signed two's complement integer), long (64-bit signed two's complement integer), float (32-bit IEEE 754), double (64-bit IEEE 754) and object reference types[5]. A number of *type conversion instructions* exist to convert between these different types, while *value storage/loading instructions* allow movement of values between the operand stack and the local variable slots.

Control flow modification instructions are provided to allow selection (conditional jumps, etc.), iteration, method invocation (and consequently recursion), constructor execution and Java's try..catch..finally exception mechanism. Support for returning control (and optionally a value) from a method invocation is also supported.

[4] In fact, local variables and operand stack values are typed, though any value can be stored in any local variable or stack slot. As `long` and `double` types are 64-bits wide, they require two locals or two stack slots for their value to be stored. The verifier ensures this property is not abused.

[5] Initial incompatibilities between Java floating point types and the IEEE 754 standard have since been resolved with the introduction of the `strictfp` keyword.

The JVM supports arrays as first class objects, with multidimensional arrays represented as arrays of references to array objects. There are certain subtleties relating to array objects, such as them having java.lang.Object as a supertype, method invocation through array pointers being valid and a number of issues concerning run-time type comparisons with primitive types and other array object references. A number of *array manipulation instructions* exist to support transfer of values between the operand stack and array components.

Finally, *type inspection instructions* support run-time type checks on objects and primitive values and *thread synchronisation* instructions support Java's built-in multi-threading facilities.

2.4 The Java Native Interface

Java provides a mechanism, known as the *Java Native Interface* (*JNI*), which facilitates linkage of Java code with legacy software. This mechanism is optional in JVM implementations, and may be complemented or replaced by a proprietary native code invocation mechanism. Commonly, however, the JNI is supported and provides a facility for dynamically linking an ANSI C library at runtime and executing functions from that library. An ANSI C library is generally supplied by the JVM which allow access to the current JVM state; for example to receive pointers to JVM objects and values which can be modified by the ANSI C code, to allow return of data to the calling method in the JVM, and more complex interactions, such as catching exceptions in JVM methods invoked from the ANSI C code.

Native methods are also commonly used within the Java API wherever interaction with the executing environment is required (such as to display user interfaces or access system-specific features). Additionally, native methods have the potential to speed critical operations by bypassing the JVM execution engine.

3 Alternatives to the Java Compilation Model

The remit of our research is to explore ways in which entire Java programs or dynamically loaded JVM classes can be executed with minimal code *size* overhead for the target system. This is fundamentally different to the usual concerns of dynamic compilation, which are typically ease of development and speed efficiency of compilation/optimisation at runtime. In view of this, fundamentally different techniques are required.

We propose a lower-level representation of the JVM classfile format (LVM) and a translator from JVM to LVM. In the interests of space efficiency on the resource-constrained system, we propose a technique we refer to as *proxy compilation* in which the translation of JVM to LVM may be performed by another computer using some form of network connection between the two systems.

3.1 A Lower-Level Alternative to the JVM Classfile Format

We are currently researching an implementation of the Java runtime environment which is implemented as a combination of a load-time Java classfile translator and lower-level virtual machine, which will be referred to as LVM for ease of reference.

By "lower-level", we refer to a virtual machine that can represent JVM programs with identical semantics, but using a representation which is platform-specific. Virtual machine design usually brings a requirement for generality - the VM must be easily implementable on a wide array of microprocessors. The role of a dynamic-compiler, however, is to generate target-specific code with identical semantics to the input. We propose to retain the benefits of an interpreter-based virtual machine but specialise that virtual machine to a specific processor.

Some innovative alternatives to the JVM classfile format have already been proposed[19]. We plan to conduct some research into alternative representations, though the majority of our work will be in developing an effective instruction set.

The anticipated benefits of this approach are that execution speed or size improvements can be obtained without compiling the program to native code, which requires significantly more storage (particularly where a RISC machine is the target). Scope exists for specialising the instruction set for particular applications or code idioms and for including direct mappings between virtual machine opcodes and hardware opcodes. The inefficient stack-based architecture, which was chosen for the JVM as it is conducive to rapid development of interpreters on a wide variety of targets, could be rewritten using virtual registers, which is likely to result in more speed-efficient code with a tolerable code-size increase.

To enable use of the proprietary LVM format, a Java classfile translator will convert Java classfiles into the LVM file format, most likely at load-time. It is proposed that all Java features will ultimately be supported by the low-level virtual machine, though our initial research is likely to use a representative subset to expedite experimentation.

The LVM itself is expected to include an interpreter and, optionally, a garbage collector (not all applications require memory deallocation). The principal design goal with regard to the LVM interpreter is that its instruction set is sufficiently close to the hardware to support direct mappings between certain instructions, registers, or other hardware attributes, but sufficiently high-level as to make run-time translation of Java bytecode to the LVM executable format tolerable (either in terms of the ROM/RAM/stack overhead of the translation process, or the time required to execute, or a combination of both).

The translator will also be able to function much like an Ahead-of-Time compiler for JVM classfiles which generates LVM format classfiles. This will permit users to compile the majority of their code to the speed- and size-efficient LVM format off-line, at which time aggressive or lengthy optimisations can be performed. Optimisations which are generally performed by native Java compilers, namely static, whole-program analysis, are inappropriate at this stage, however, as dynamic class loading renders part of the application unknown until runtime. Although a proportion of applications can be proven to not require dynamic class loading, we are choosing to focus on those which do, rather than researching aggressive, off-line whole-program analysis, which is the subject of many current Java research projects. There is scope for the use of annotations within JVM classfiles, such as those suggested by Hummel et al[14]. Such annotations would be introduced by the translator when generating target code to aid the runtime system in maximising execution efficiency. Again, however, we are unlikely to focus on this aspect as it has been considered by other researchers and would detract from our proxy compilation research.

The proposed system is a compromise between program size and execution speed efficiency. As the LVM instruction set is designed to represent lower-level operations, a JVM program converted to the LVM format is expected to require more storage space than the original JVM program. The LVM system startup is also expected to be slower than an interpreted JVM system. Moreover, we anticipate that LVM execution speed will be significantly faster than an interpreted JVM system, although slower than a JIT-based JVM system. Fundamentally, however, we expect the runtime footprint of the LVM system to be significantly less than a JIT-based JVM, application execution to be significantly faster than an interpreted system and for the memory requirement for translated LVM code to be significantly less than that for native code.

Should critical portions of the application require compilation to native code, either Ahead-of-Time or during program execution (i.e. a runtime profiler identifies program hot spots), we would ideally like to be able to natively compile these sections. In view of this, we propose to support a native interface between the LVM and native code. Such decisions can be influenced either by user intervention (static compilation of critical code), or by profiling at runtime.

3.2 Proxy Compilation

One possible outcome of the research work described above could be that, owing to limitations on available time for the translation of classfiles at load-time, the runtime translation of JVM to LVM is impractical within a specific ROM, RAM or stack limit (above which, assumedly, the translation would be too intrusive or expensive). A sufficiently complex application or sufficiently scarce system resources will inevitably bring about this situation. We propose to increase performance using a technique we refer to as *proxy compilation*.

In a proxy compilation scheme, idle peer devices to which the device is connected via some form of network or direct connection are candidates for remote compilation on behalf of the resource-constrained system. The remote compilation may involve translation of a dynamically loaded JVM classfile to LVM, native code, or some other representation.

It is difficult to gauge the extent to which a proxy compilation service might be required by a resource-constrained system like a PDA. It is our impression, nevertheless, that dynamic compilation and dynamic class loading would be valuable additions to a resource constrained system. Additionally, we feel that the feasibility of dynamic class loading and runtime compilation in such environments increases when the overheads of translation are transferred from the client to the server, such as one achieves with our proxy compilation proposal.

This technique is expected to become very important as micros are increasingly embedded into products where, owing to production costs or other concerns, insufficient resource exists to effect a translation of a program from JVM to LVM (or some other format), but a connection to a network of peer machines has been included. Such networks could include other small-scale embedded devices, but also, potentially, very powerful machinery which has the capability to trivially perform the compilation tasks on behalf of peers. The advent of innovations such as the Bluetooth[8] standard for wireless local networking is likely to make such schemes highly attractive.

One particularly important aspect concerning a proxy compilation scheme of this nature is that of how to ensure the validity of incoming LVM or native code. We do not currently plan to focus on authentication or scheduling aspects of such distributed execution, but rather to concentrate on studying the utility and efficacy of such a system, assuming that such issues can be overcome with negligible runtime overheads.

We intend to use proxy compilation for both JIT compilation of JVM to LVM and native code and also specialisation of the LVM instruction set itself in response to profile results. A compiler whose runtime does not result in a linear increase in the target system execution time permits iterative optimisation and specialisation[39] of both the runtime system and the target program throughout application execution. This is analogous to continuous compilation[27], but within the framework of a resource-constrained system. Clearly, however, the client's resource constraints impose restrictions on the applicability of specialisation of the VM to a particular application. Whereas a single application might benefit, multiple applications would require multiple VMs, which would trade off against the efficiency increases introduced by the specialisation.

The intermittency and reliability of the network or cable connection between the two machines needs careful consideration. Should the connection be lost while proxy compilation is in progress, the client device must have a contingency policy. This policy is likely to be dictated by the resources the client has available. If a previously-compiled version of the code has been cached, this can be executed. Alternatively, if an interpreter is available, the client may choose to interpret the code. If the connection problem proves to be specific to a particular machine, an alternative may be found and the proxy compilation requested anew. Finally, if none of these options are available, the client is entirely dependent on its network connection and must wait until that connection is restored, at which point either the interrupted proxy compilation session is resumed or a new session started.

4 Next Steps

We are currently implementing a dynamic compilation system to allow us to experiment with the ideas outlined above. The system is research-driven; consequently, we plan to support representative, but minimal Java programs. This implies supporting a subset of the Java 2 Platform API with, for example, complex I/O and networking facilities omitted. The AWT, Swing and JFC sections of the Java 2 Platform API will also be omitted, allowing us to focus on console-based applications.

Firstly, we are defining and implementing an initial LVM in C++, together with a JVM classfile JIT compiler for producing LVM or target machine code at runtime. We expect to either implement a very simplistic garbage collector or use one of those freely available[2] under the GNU Public License[36]. We will consider the interaction of the LVM with the garbage collector during this work; there may be some advantages to be gained from representing objects using a load/store VM rather than a stack-based system. The JIT system will be used as a control for the proxy compilation experiments (although other research and commercial systems will also be used). Implementation of a proxy compiler reading JVM classfiles and generating LVM or target machine code is underway. We are implementing the proxy compiler in Java to facilitate rapid prototyping but intend to compile the JVM bytecode to target code using an Ahead-of-Time compiler so as to maximise efficiency before testing the final system.

The system is being designed to permit dynamic re-compilation of the LVM system to maximise runtime efficiency. One benefit of this facility is the option to optimise the system or adapt the LVM instruction set in response to feedback gathered during a previous run, assuming resources permit methods such as execution profiling.

An initial protocol for proxy compilation requests, together with matching client/server code is almost complete.

To speed initial development and research, we will be using Intel x86 as both the server and client platform. This is largely immaterial as the systems are being designed for resource-constrained environments. Later in the project, we expect to re-implement the system to target a real client system. This is likely to be either a Palm Pilot m100 PDA or Hewlett Packard Jornada Windows CE HPC. The Palm uses a 16MHz Motorola Dragonball EZ (MC68EZ328) processor and has 2MB RAM while the Jornada has a 133MHz Hitachi SH3 (SH7709A) processor with 16MB RAM. For the purposes of simplicity, we will assume the system will be stored and executed from RAM, although clearly any static portions of our proposed runtime system could theoretically be stored in ROM.

Following completion of these software components, we will measure the runtime efficiency of the system using a set of representative benchmarks. We will consider both code-size, including the memory usage of the runtime system and the overheads of the proxy compilation technique on the network; and total execution time, including proxy compilation and any local translation time.

We plan to publish our findings in due course, contrasting our results with those obtained by other researchers in the field and relevant commercial software.

5 Related work

For reasons outlined above, interpreter and JIT-based systems are inefficient in resource-constrained environments. Systems such as Insignia Solutions' Jeode product[17], which reduce the overhead of JIT technology for embedded systems still impose a footprint upon the runtime system; the compiler must be stored on the target system and must compile code in place of or in competition with the application.

McGlashan and Bower[22] claim that an interpreter-based virtual machine delivers "entirely adequate performance" for consumer embedded systems products such as the SEGA Dreamcast. While they recognise the importance of runtime minimisation in such environments, and commend

interpreter-based VMs for their high-speed startup times and small memory footprint, their argument is based upon their own Smalltalk VM. The relevance of their findings to Java systems is unclear, though assumedly a proxy compilation system benefits from the minimal VM while also reaping the benefits of a powerful and flexible dynamic compilation system.

In the Ahead-of-Time domain, numerous static Java compilation systems exist[30, 9, 25, 24, 31, 11, 5, 38] though most are oriented to desktop systems with plentiful resources. All systems generate native binary code either directly or via ANSI C, which is subsequently compiled off-line. This approach is generally speed efficient, but often forbids use of dynamically loaded classes (e.g. [4]) and suffers the overheads of native code relative to compact VM bytecode.

The COMPOSE group's Harissa[25, 24] system notably acknowledges the requirement for statically-compiled Java applications to execute dynamically loaded classes, however, its solution - an interpreter - results in a slow, if compact solution. An equivalent interpreter has recently been added to the GNU gcj compiler[31]. Neither system addresses efficient runtime compilation of dynamically loaded Java code in resource-constrained environments.

Roelofs[32] notes the characteristics of resource-constrained systems, but is chiefly concerned with using connected devices to allow remote execution of application code. Our research instead seeks to use more powerful peers to speed translation of the device's core program for subsequent execution on the device itself.

Wakeman et al[26, 15] have worked on research which similarly acknowledges the problems of environments in which resources are limited. Their approach uses a proxy device to serve suitably compressed or scaled versions of requested data in accordance with client-specified constraints expressing, for example, degradation limits. This is analogous to the notion of proxy compilation, though the authors have not specifically proposed it. The work also proposes that clients inform the proxy of their resources. This is a potentially attractive technique which would allow the server to specialise a code fragment or application for the specific resources available to the client. In situations where low resources prohibit execution profiling, this may be the only feedback the client can provide regarding the runtime environment. An additional, albeit lesser, consideration is that their implementation uses Java and RMI on the client side. Our work directly addresses the question of proxy compilation and is designed to scale to very simple clients where a Java runtime environment may not be feasible.

The vast majority of dynamic compilation systems require storage of a dynamic compiler system in the runtime environment, and must execute on the target system. The small number of projects which do not employ this model are now described. Voss and Eigenmann[41] detail a system which is notionally similar to proxy compilation, but assumes various system characteristics. These include a requirement for NFS mounted storage to be shared between systems and a reliance on RPC facilities. We believe such a solution would not scale well to resource-constrained systems (particularly single threaded applications which use a minimal operating system or do not require an OS). Additionally, this project has focussed on ANSI C and FORTRAN applications rather than Java.

Bell Labs' Inferno system[44, 43, 7] and Tao Systems' Elate/Intent system[12] both use a low-level VM instruction set to increase the efficiency of Java code. These two systems are now contrasted with our proposed systems.

Inferno's use of a memory-to-memory virtual machine results in a virtual machine architecture which is superficially similar to our proposed LVM system. There are a number of critical differences, however. Firstly, Inferno is target-independent, supporting Intel x86, SPARC, ARM, PowerPC, MIPS and other devices. Although the principle of a low-level virtual machine is applicable to targets with a load/store architecture, we expect to increase efficiency by creating specialised versions of the LVM instruction set for individual processors. Furthermore the Inferno virtual machine (Dis) has an instruction set which has been designed for the Limbo programming language, not Java. Although there are many similar features, including objects and garbage collection, supported by an extended DIS VM instruction set, Inferno was not, to our knowledge, designed for Java. This may result in inefficiencies. Finally, although designed for network environments, Inferno neither directly supports nor proposes a proxy compilation scheme.

Elate/Insight also uses a low-level, target-independent instruction set, however it, like the Kimera project[35], use a form of remote compilation which relies on shared memory and persistent network connections. Such systems fail to acknowledge the often intermittent nature of network connections to resource-constrained devices. As described above, our proxy compilation scheme is designed to scale to a wide variety of machines through techniques such as caches of previously compiled code and efficient, client-side interpretation.

Fraser and Proebsting[29] have proposed a technique for the automatic generation of custom instruction sets in pursuit of space efficiency. Their system generates an interpreter and interpretive code with functional equivalence to the original program. Although the authors were interested in code compression, the techniques employed to detect common patterns in the input ANSI C program are likely to be relevant to our own work on automatic specialisation of the LVM.

An interesting extension of our research could be to empirically evaluate the interaction of proxy compilation with proposed continuous compilation strategies[27, 28].

6 Acknowledgements

We would like to thank Hitachi Micro Systems Europe for supporting this research.

The notion of proxy compilation described in this paper evolved from a conversation with Ian Wakeman, University of Sussex, wherein a similar concept was discussed in the context of active networks.

Ulrik Page Schultz of IRISA, University of Rennes, provided useful comments during the initial stages of the research. Thanks also to Michael J. Voss and Rudolf Eigenmann of Purdue University for comments and feedback.

Finally, we would like to thank our peers in the Software Systems group at the University of Sussex for useful feedback throughout the project.

7 Summary

We have outlined our reservations with regard to the appropriateness of current dynamic compilation schemes for Java in resource-constrained environments. We have summarised a proposal for a lower-level virtual machine, which benefits from both a target-specific instruction set (potentially generated through a combination of proxy compilation and partial evaluation using profile results, assuming the device has sufficient resources to gather execution profiles) and a compact bytecode representation. In this context, we have also proposed a proxy compilation mechanism, in which idle networked machines may be employed to perform translation and compilation on behalf of the target system. Our expectation is that our research will prove proxy compilation is a powerful and flexible general purpose technique for dynamic compilation. We also anticipate the value of such techniques will increase as small-scale mobile devices proliferate.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Hans-J. Boehm. C/C++ Garbage Collector Libraries. `http://reality.sgi.com/boehm_mti/gc.html`.

[3] Borland/Inprise. JBuilder Homepage. `http://www.borland.com/jbuilder/`.

[4] Natural Bridge. Bullet-Train Homepage.
`http://www.naturalbridge.com/bullettrain.html`.

[5] Department of Computer Science and Engineering, University of Washington.
Cecil/Vortex Homepage.
`http://www.cs.washington.edu/research/projects/cecil/www/index.html`.

[6] Stephan Diehl. A Formal Introduction to the Compilation of Java. *Software – Practice and Experience*, 28(3):297–327, March 1998.

[7] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *Proceedings of the IEEE Compcon 97 Conference*, pages 241–244, San Jose, 1997.

[8] Ericsson Mobile Communications AB. Official Bluetooth website.
`http://www.bluetooth.com`, 1999.

[9] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, 2nd edition, 1999.

[11] Silicomp Group. Turbo-JHomepage.
`http://www.ri.silicomp.fr/adv-dvt/java/turbo/`.

[12] Tao Group. Elate/Insight Homepage.
`http://www.tao.co.uk`.

[13] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley and Sons, Ltd., 2000.

[14] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java Bytecode in Support of Optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

[15] Ian Wakeman, Andy Ormsby, and Malcolm McIlhagga, School of Cognitive and Computing Sciences, University of Sussex. An Architecture for Adaptive Retrieval of Networked Information Resources. *IEE Colloquium on "Issues for Networked Interpersonal Communicators"*, May 1997.

[16] Insignia Solutions plc. Jeode Platform Data Sheet.
`http://www.insignia.com/product/jeode_ds.htm`, November 2000.

[17] Insignia Solutions plc. Jeode Platform White Paper.
`http://www.insignia.com/product/white.htm`, November 2000.

[18] kaffe.org. Kaffe Project Homepage.
`http://www.kaffe.org`.

[19] T. Kistler and M. Franz. A Tree-Based Alternative to Java Byte-Codes. *International Journal of Parallel Programming*, 27(1):21–34, February 1999.

[20] Andreas Krall and Mark Probst. Monitors and Exceptions: How to Implement Java Efficiently. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15–24, February 1988.

[21] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[22] Blair McGlashan and Andy Bower, Object Arts Ltd. The Interpreter is Dead (Slow). Isn't it? Position Paper for OOPSLA'99 Workshop: Simplicity, Performance and Portability in Virtual Machine design.
`http://www.squeak.org/oopsla99_vmworkshop/`, October 1999.

[23] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[24] G. Muller, B. Moura, F. Bellard, and C. Consel, IRISA / INRIA, University of Rennes. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *3rd Usenix Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pages 1–20, Portland, Oregon, June 1997.

[25] Gilles Muller and Ulrik Pagh Schultz. Harissa: A Hybrid Approach to Java Execution. *IEEE Software*, pages 44–51, March/April 1999.

[26] School of Cognitive and Computing Sciences, University of Sussex. Lowband Homepage.
`http://www.cogs.susx.ac.uk/projects/lowband/`.

[27] Michael P. Plezbert. Continuous Compilation for Software Development and Mobile Computing. Master's thesis, Department of Computer Science, Washington University, May 1996.

[28] M.P. Plezbert and R.K. Cytron. Does "Just in Time" = "Better Late than Never". In *Proceedings of POPL'97*, pages 120–131, Paris, France, January 1997.

[29] Todd A. Proebsting. Custom instruction sets for code compression. Unpublished., October 1995. `http://www.research.microsoft.com/~toddpro/papers/pldi2.ps`.

[30] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. *Toba: Java for Applications: A Way-Ahead-of-Time Compiler.* Technical report, University of Arizona, January 1997.

[31] Redhat. GCJ Homepage.
`http://sources.redhat.com/java/`.

[32] Brian Roelofs. Java Custom Class Loaders. *Dr.Dobb's Journal*, 25(6):74–80, June 2000.

[33] Govind Seshadri, Editor-in-Chief. The VM Wars: JITs vs. Native Java Compilers. *Java Report Online*, 1999.

[34] Kazuyuki Shudo. Shujit Homepage.
`http://www.shudo.net/jit/`.

[35] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *17th ACM Symposium on Operating System Principles (SOSP'99), published as Operating Systems Review*, volume 34, pages 202–216. University of Washington, Department of Computer Science and Engineering, December 1999.

[36] Free Software Foundation. GNU General Public License.
`http://www.fsf.org/copyleft/gpl.html`.

[37] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 3rd edition, 1997.

[38] Sun Microsytems. The Java Hotspot Performance Engine Architecture (White Paper).
`http://www.javasoft.com/pr/1999/04/pr990427-01.html`, November 2000.

[39] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in Program Compilation by Interpreter Specialization. Technical Report 3588, INRIA, Rennes, France, December 1998.

[40] Bill Venners. *Inside the Java Virtual Machine.* McGraw-Hill, 2nd edition, 1999.

[41] Michael J. Voss and Rudolf Eigenmann. A Framework for Remote Dynamic Program Optimization. In Jong-Deok Choi, editor, *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, volume 35, pages 32–40, Boston, MA, July 2000. Association of Computing Machinery, ACM Press.

[42] Des Watson. *High-Level Languages and their Compilers.* Addison-Wesley, 1989.

[43] Phil Winterbottom and Rob Pike, Bell Labs, Lucent Technologies. The Design of the Inferno Virtual Machine.
`http://www.cs.bell-labs.com/cm/cs/who/rob/hotchips.html`.

[44] C. F. Yurkoski, L. R. Rau, and B. K. Ellis, Bell Labs, Lucent Technologies. Using Inferno to Execute Java on Small Devices. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98*, volume 1474 of *Lecture Notes in Computer Science*, Montreal, Canada, June 1998. Springer.