A DYNAMIC BINARY TRANSLATOR IN A JAVA ENVIRONMENT

Ian Rogers

Abstract

Dynamic binary translation looks to map one computer architecture to another. Java is unusual in that it compiles to a platform independent bytecode that runs on a virtual machine. This paper shows a dynamic translation and compilation environment that can provide a Java virtual machine (JVM). Furthermore, dynamic compilation has the ability to perform run-time optimisations that are unavailable to a conventional static compiler. A technique is shown that translates Java bytecodes in sympathy with dynamic compiler optimisations. This is done with the aim of rivalling and surpassing static optimisation techniques using dynamic ones. This will enable Java to stay true to its goal of being platform independent and yet running at comparable speeds to statically compiled code. Results from the Dynamite JVM, developed as part of this research, are presented.

1 Introduction

The Java environment was developed in 1995 [Gosling, McGilton, 1995]. Features from many programming languages and program language libraries were combined into a system that set out to be platform independent. The growth in Java's libraries and some alterations to the original language specification have led to Java's continual growth as a commercial programming language and tool. For instance, in a recent survey Java was cited as the most in demand IT job skill [Computer, 2000].

By being so ambitious with its features, Java has suffered from lower performance than rival programming languages such as C++, which it was hoped it to would replace. A key reason for the low performance was the execution of the platform independent bytecodes, with in a Java virtual machine, rather than static compilation. To speed up the execution of these bytecodes Java virtual machines optimised the bytecodes they were interpreting [Lindholm, Yellin, 1999]. Hardware was also developed that could execute Java bytecodes [O'Connor, Tremblay, 1997]. The hardware technique has found a niche in embedded devices where a Java environment needs to be compact as well as fast [Cormie, 2000]. Legacy computers and CPUs unwilling to compromise their native instruction speed were left with interpretation as their Java virtual machine environment. This has led to the invention of Just-in-Time (JIT) and dynamic Java compilers[1].

Just-in-Time compilation compiles a Java class or method the first time it is accessed. Example Just-in-Time compilers are Symantec's JIT compiler that came with Microsoft Windows versions of the Sun Java Development Kit [Symantec, 1998] up until Java 2 version 1.3, and Microsoft's own JVM [Microsoft, 2001]. Just-in-Time compilation slows the execution of the Java program, but it is hoped by executing the compiled methods rather than interpreting the Java bytecodes the time will be made up. Often this approach is naïve as not all code in a class or a method is executed within a run of a program. Some bytecodes are only executed once and don't warrant compilation as interpretation would be faster.

Dynamic compilers compile Java bytecodes when it is appropriate. The compiled bytecodes may be a small part of a method or they could span several methods. The dynamic compiler may also have an interpreter to interpret bytecodes which won't be executed frequently. It is appropriate to compile bytecodes when the cost of the time spent compiling will be more than regained by executing the faster compiled code. This can only be known in retrospect, so dynamic compilers rely on profiling information to gather statistics about the run of a program and to predict where a speed up can be achieved. This also allows expensive compiler optimisations, such as method inlining, to be targeted. These optimisations may break the Java virtual machine specification, but by placing checks around the optimised code a safe fall back can be used when the optimisation is unsafe. Run-time optimisation can't be performed by a static compilers as they must ensure the code produce will work in all circumstances.

Dynamic compilers are a new form of Java virtual machine that have only appeared recently. This paper looks at the development and optimisation of the Dynamite JVM which falls into the dynamic compiler category of Java virtual machine. The Dynamite JVM is novel as it builds on work for the Dynamite dynamic binary translator [Souloglou,

[1] Static compilers for Java have also been developed [Free, 2001]. As these are unable to load and execute Java class files dynamically they do not meet the Java virtual machine specification [Lindholm, Yellin, 1999]. The author therefore omits their discussion.

Ph.D. Research undertaken at the Department of Computer Science at the University of Manchester

1996]. Section 2 talks about dynamic binary translators and there recent emergence as a commercial tool. Section 3 discusses the features of a Java virtual machine that make it difficult to compile and optimise for. Section 4 presents the Dynamite JVM and how it addresses the problems of section 3. Section 5 shows preliminary results of using the Dynamite JVM with certain kernel benchmarks. Finally, section 6 looks at the continued development of the Dynamite JVM and the Dynamite dynamic binary translator.

2 Dynamic Binary Translators

Dynamic binary translation is a technique for recompiling code from one instruction set (the subject instruction set) to another instruction set (the target instruction set) whilst the program is running. This allows legacy computer programs to be run on new faster or lower power computer architectures. Hardware and software techniques are in commercial use today. The AMD K6 3D processor performs hardware translation of an IA32 instruction to several RISC86 Ops in the instruction decode phase of its pipeline [Shriver, Smith,1998]. Transmeta in their Crusoe product [Transmeta, 2000] perform software translation. Software translation runs a translation program on the native processor and caches the results in a translation cache (typically an area of DRAM). Code from the translation cache is then executed on the native processor.

Optimisations in hardware translation are limited by how long the processor can spend decoding an instruction (long or slow pipelines reduce the number of instructions per clock) or by the size of the window of code the decoder is translating. Software translation must appear transparent to a user, meaning quick translations are often needed. Software translation has the ability to perform expensive code optimisation (unavailable to a hardware translator) in idle computer time or when profile information says it would be beneficial. A software translator will significantly simplify an instruction decoder in a processor pipeline. An example of this is in the Crusoe processor pipeline where precise exceptions aren't available. Instead precise exceptions can be emulated by rolling the processor state back to before the exception and then using differently translated code to find the exact subject instruction that causes an exception. The potential of dynamic optimisation has been seen in the Dynamo dynamic binary translator. On HP PA-RISC code Dynamo was able to translate –O2 optimised code and execute it at –O4 speed [Bala, Duesterwald, Banerjia, 2000].

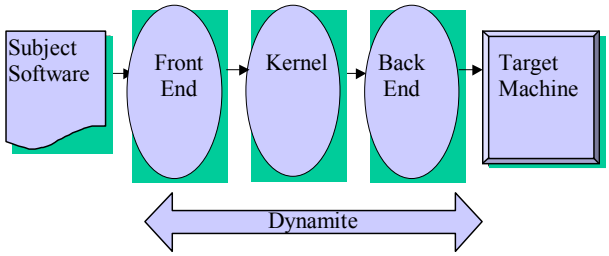

Figure 1 The Dynamite dynamic binary translator

Dynamite is a dynamic binary translator created at the University of Manchester [Souloglou, 1996]. It is now being developed by a start-up company called Transitive Technologies [Transitive, 2001]. Dynamite provides a backend that is tailored for a specific computer architecture. Existing backends are for the IA32 (a CISC architecture) [Sandham, 1998], Sparc and ARM (both RISC architectures) [Souloglou, 1996] [Linfoot, 1999], and PlayDoh (a VLIW architecture in many ways similar to IA64) [Howson, 1999]. The Dynamite kernel provides the translation cache, an optimised intermediate representation and a hot-region[2] optimiser. The Dynamite frontend drives the translator and provides mechanisms for handling operating system calls and/or peripheral hardware. Dynamite frontends exist for IA32 [Sandham, 1998], 68000, ICL 2900, PostScript, STUMP (a simple 16bit RISC architecture based on ARM), LARD (a hardware description language) [Rogers, 1998] and Java.

3 Java Virtual Machine

The Java virtual machine is ambitious in the features it makes routinely available to the programmer. The execution model is entirely dynamically linked making all references symbolic. It supports virtual methods and interfaces, that provide single inheritance and approximate multiple inheritance. The class loader, which is responsible for the loading of the program and libraries, can be replaced by a user defined one that could, for example, load parts of the program from over the internet. The class loader also has to support a reflective API which allows a user to load and modify a classes behaviour. To enable the large number of Java libraries to work, the Java virtual machine needs to be able to call out to software routines written for the native processor. The Java virtual machine provides a garbage collected

[2] Hot-regions are areas of code that profiling has shown to be executed frequently. Optimising these regions has the biggest pay off due to the 90/10 rule [Hennessy, Patterson, 1996] that states 90% of execution time is spent in 10% of code.

memory manager. Java also has in built support for threading with locking and synchronisation primitives.

Java methods contain additional information for its exception model. The Java exception model allows programs to be written for the default case with exceptional cases thought of separately. The exception model requires extra checking to be performed by the virtual machine, as well as adding multiple possible pathways through a program. So as not to preference any particular underlying register architecture, the Java virtual machine uses a 0-address, stack based instruction set (Java bytecodes). The stack increases the number of instructions that a simple routine like add would require to be performed. For example, on a RISC architecture add would simply be add 2 registers and store the result in a 3rd (1 instruction), with the Java virtual machine stack this becomes push 2 values, add the top 2 values then pop the result (4 instructions).

4 Dynamite JVM

The Dynamite JVM can potentially provide all the features of a Java virtual machine. Currently a key set of features have been focussed on. Full support for the native API, threading and exceptions is being worked on. The description below describes the eventual complete mechanisms.

The Dynamite JVM builds up intermediate representation a basic block[3] at a time. Bytecodes are translated using a symbolic stack and with local variables mapped in to registers. The way Java programs are compiled means that the stack is empty over 93% of the time on basic block boundaries [Krall, 1998]. This means that the stack overhead is removed in the majority of basic blocks. The dynamite kernel performs dead code removal and common sub-expression elimination. Dead code is detected by watching which registers are written to by the intermediate representation. If a register is written to twice without the first value being used then the intermediate representation that generates the first value can be safely be removed.

[3] A basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [Aho, Sethi, Ullman, 1986]. Basic blocks in dynamic translators are different as they are discovered on the fly which means not all entry points are known. Because of this dynamic translators have to allow basic blocks to overlap to avoid re-translation. This isn't true for basic blocks with in static compilers where all entry and exit points are known in advance.

After the intermediate representation has been created then the backend code generates over it.

As basic blocks are executed with the Dynamite JVM profiling statistics are created. Hot basic blocks are detected and joined with other hot basic blocks covering a hot-region and generating a group block. When a group block is generated dead code elimination is performed. Register allocation is tuned with in a group block to prefer hot basic blocks and thus reduce the number of register spills (based on the assumption that basic blocks spill an equal amount).

To maximise the amount of code that can be optimised with in a group block we need to keep local and stack variables visible over method boundaries. In conventional compilers this is done using techniques such as register colouring over method boundaries or method inlining. Register colouring has proven to be an NP complete problem [Chow, Hennessy, 1990] and therefore not suitable for use in a dynamic compiler. Method inlining has appeared in the Sun HotSpot Java virtual machine and is run as a separate optimisation phase which requires de-optimisation if assumptions later prove incorrect [Sun, 1999].

A novel approach to increasing the visibility of variables is used in the Dynamite JVM with an infinite abstract register pool as shown in figure 2. Every translated method is given a portion of the register bank to use. Parameter passing is performed by writing the parameters straight from the symbolic stack into the called method. When the called method is unknown, for instance in virtual method calls, the values are written into registers which are then read into the symbolic stack of the called method. Unnecessary register copying that isn't eliminated by the use of the symbolic stack will be removed in the dead code removal phase of group block creation. A memory backup scheme is used when a method's frame is required to be used more than once (i.e. recursive method calls). For typical applications studies have shown the register pool need be little larger than 8000 registers. If a larger register pool is required then parts can be reclaimed (by deleting the dynamic code that uses it) or we can share sections by noting that leaf methods can never call each other (by definition).

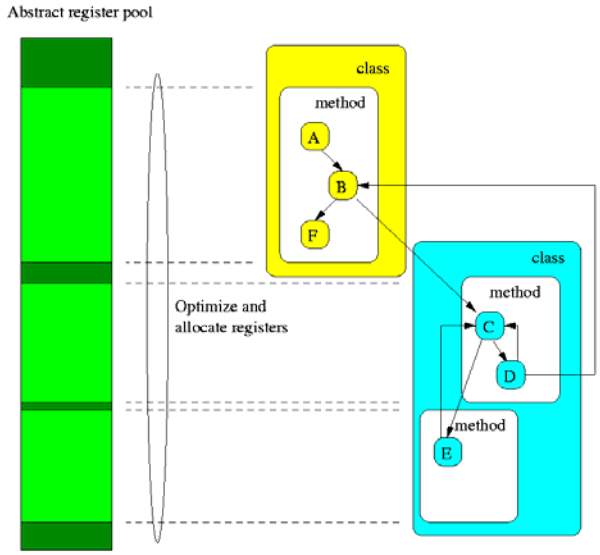

Figure 2 Mapping methods to abstract register pool

As abstract register pool registers are swapped in and out of target machine registers an area of memory is used for storing the backed up values. Threading is supported by swapping the pointer to the backed up values to those of another thread. Currently no locking and synchronisation on objects is available.

Exceptions are supported by mapping target machine instructions back to Java bytecodes. The exception handler is then found by searching through the Java virtual machine stack. A slight optimisation is used so that signal trapping isn't used to catch null pointer exceptions. Instead a null object is created which catches reads and writes by protecting the memory it is created in. Branches are permitted but they go straight to the Dynamite JVM exception handling mechanism.

Garbage collection is only performed when a program terminates. An interface is available so that a new garbage collector can be written.

5 Results

To determine the efficiency of the abstract register pool the Kaffe [Wilkinson, 2001] JVM interpreter was instrumented. When running the javac benchmark (part of the OSG SPEC JVM98 benchmark suite [SPEC, 1998]) we recorded information on hot-regions and local variable usage. The model of the register allocation algorithm allocated registers to local variables based on their run-time contribution per local variable. Registers required for maintaining call stack frames and the JVM stack's temporary variables were ignored. The results are shown in figure 3.

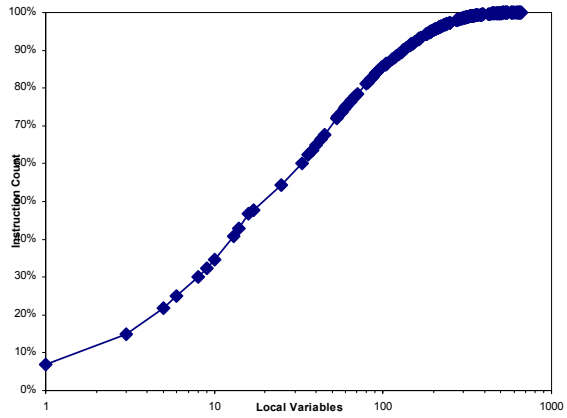

Figure 3 The approximate percentage of javac execution that will fit with in a group block with out spill code

The graph shows that pay off of having more registers available to the register allocation algorithm. The algorithm is able to perform well with few registers, it gets 30% of the total instruction count in 8 registers. This is good for instruction sets with a small number of visible registers such as the IA32. With an instruction set with a more reasonable 25 registers, 54% of the total number of instructions are covered.

The performance of the code translation with in the Dynamite JVM was tested. A bubble-sort kernel was written and run on the Sun JDK version 1.17 [Blackdown, 1999] as well as the Dynamite JVM and the Dynamite JVM with hot-region optimisation (-O). The Sun JDK was with out a JIT compiler and the Dynamite JVM was running on an old revision of the IA32 backend. All start up overheads (i.e. to load system libraries and perform translation) are removed. The results are shown in figure 4.

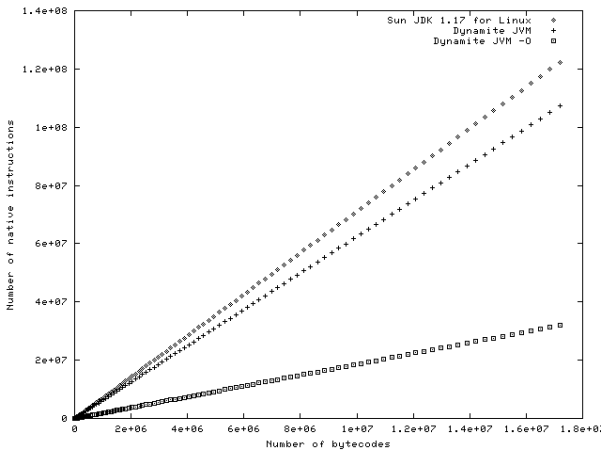

Figure 4 Performance of Dynamite JVM on bubble-sort

The figure of merit is calculated by dividing the number of native machine instructions executed per subject machine instruction. The Sun JDK

interpreter has a figure of merit of 7.11 Pentium II instructions per Java bytecode. The Dynamite JVM without hot-region optimisations has a figure of merit of 6.23, where the bulk of this is due to translator overhead in moving between basic blocks. The Dynamite JVM with hot-region optimisations (-O) has a figure of merit of 1.86. With the latest code optimisations in the Dynamite kernel and IA32 backend it is reckoned this figure will be four times faster.

Finally, the performance of a recursive benchmark is tested as recursion is the worst case for our abstract register pool optimisation scheme. A Java conversion of the Takeuchi benchmark was made. The results are shown below.

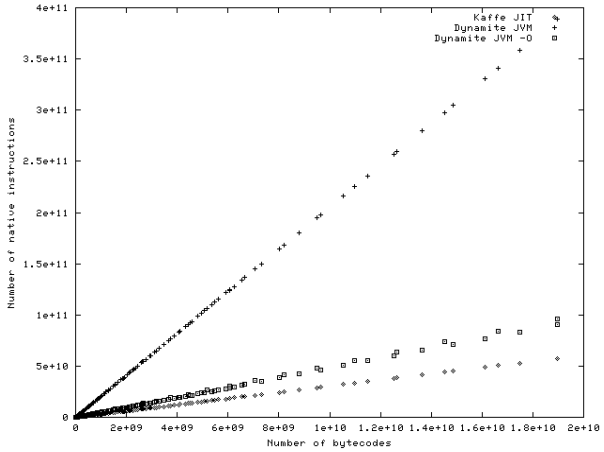

Figure 5 Performance of Dynamite JVM on Takeuchi

For comparison the Kaffe JIT compiler version 3 [Wilkinson, 2001] was timed. The results show a figure of merit of 3.03 for the Kaffe JIT compiler. The Dynamite JVM with out hot-region optimisations has a figure of merit of 20.49 and a figure of merit of 5.07 with hot-region optimisations. These figures of merit are high due to the overhead of method calling. The effect of scheduling the basic blocks with in the group block can be seen on the Dynamite JVM –O line (the line isn't straight but fluctuates up and down). The reason for this is the input parameters have caused a different ordering of the basic blocks run. With the latest kernel and backend for Dynamite, the Dynamite JVM with optimisations is expected to out perform the Kaffe JIT compiler. An example optimisation yet to be performed (in these results) is removing dead code of the form store to location x the value loaded from location x. By removing this dead code by hand, a figure of merit of around 3 is achieved.

6 Conclusion

This paper has introduced the Dynamite JVM a complete Java virtual machine built on a dynamic binary translator. By allowing finer granularity optimisations it is expected to out perform current JIT compilers. It is also different from current dynamic Java compilers in the way it standardises its intermediate representation over all levels of optimisation. This means different register allocation schemes, for example, won't be needed when performing optimisations such as method inlining. In fact the abstract register pool eliminates method boundaries and doesn't require separate optimisation and de-optimisation passes to remove method call overhead. With advancements in optimisations in the Dynamite kernel and backend, the Dynamite JVM is expected to be amongst the fastest Java virtual machines.

The use Sun's Java environment has made virtual machines popular. Recently new virtual machines have been invented, notably the Microsoft .NET virtual machine [Microsoft, 2001a]. The techniques described in this paper have applications to these and other virtual machine environments.

References

[Aho, Sethi, Ullman, 1986] Alfred Aho, Ravi Sethi, Jeffrey Ullman, "Compilers: Principles, techniques, and tools", Addison-Wesley, 1986.

[Bala, Duesterwald, Banerjia, 2000] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, "Dynamo: A transparent dynamic optimization system", Hewlett Packard labs, in the proceedings of PLDI 2000.

[Blackdown, 1999] Blackdown, "Java linux", http://www.blackdown.org/, 1999.

[Chow, Hennessy, 1990] Fred C. Chow, John L. Hennessy, "The priority-based coloring approach to register allocation", ACM transactions on programming languages and systems, volume 12, issue 4, 1990.

[Computer, 2000] Computer Weekly, "ITers must become e-people", November, 2000.

[Cormie, 2000] David Cormie, "Jazelle ™ - ARM ® architecture extensions for Java applications", ARM, November 2000.

[Free, 2001] Free Software Foundation, "GCJ: the GNU compiler for Java", http://www.gnu.org/software/gcc/java/, Free Software Foundation, April 2001.

[Gosling, McGilton, 1995] James Gosling, Henry McGilton, "The Java language environment", technical report, Sun Microsystems, May 1995.

[Hennessy, Patterson, 1996] J. L. Hennessy, D. A. Patterson, "Computer architecture: a quantitative approach", Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[Howson, 1999] Miles Howson, "PlayDoh backend for Dynamite", final year undergraduate project report, the University of Manchester, 1999.

[Krall, 1998] Andreas Krall, "Efficient Java VM Just-in-Time compilation", in the proceedings of PACT '98, Paris, France, October 1998.

[Lindholm, Yellin, 1999] Tim Lindholm, Frank Yellin, "The Java Virtual Machine Specification", Addison-Wesley, Reading, MA, USA, second edition, 1999.

[Linfoot, 1999] Mike Linfoot, "An ARM backend for Dynamite", final year undergraduate project report, the University of Manchester, 1999.

[Microsoft, 2001] Microsoft Corporation, “Microsoft technologies for Java”, http://www.microsoft.com/java/, Microsoft Corporation, 2001.
[Microsoft, 2001a] Microsoft Corporation, “Microsoft .NET”, http://www.microsoft.com/net/, Microsoft Corporation, 2001.
[O’Connor, Tremblay, 1997] J. Michael O’Connor, Marc Tremblay, “picoJava-1: The Java virtual machine in hardware”, IEEE Micro, 17(2), March 1997.
[Rogers, 1998] Ian Rogers, “A LARD frontend for Dynamite”, final year undergraduate project report, the University of Manchester, 1998.
[Rogers, Rawsthorne, Souloglou, 1999] Ian Rogers, Alasdair Rawsthorne, Jason Souloglou, “Exploiting hardware resources: register assignment across method boundaries”, ICCD workshop on hardware support for objects and microarchitectures for Java, Austin, Texas, USA, October 10, 1999.
[Sandham, 1998] John Sandham, “Dynamite frontend and backend for x86”, final year undergraduate project report, the University of Manchester, 1998.
[Shriver, Smith, 1998] Bruce Shriver, Bennett Smith, “The anatomy of a high-performance microprocessor: a systems perspective”, IEEE Computer Society, 1998.
[Souloglou, 1996] Jason Souloglou, “A Framework for Dynamic Binary Translation”, M.Phil. Thesis, The University of Manchester, 1996.
[SPEC, 1998] Standard Performance Evaluation Corporation, “SPEC JVM98”, http://www.spec.org/osg/jvm98/, OSG, 1998.
[Sun, 1999] Sun Microsystems Inc, “The Java HotSpot Performance Engine Architecture”, Sun White Paper, April 1999.
[Symantec, 1998] Symantec Corporation, “Symantec’s Just-in-Time (JIT) Java compiler runs Java applets and applications 50% faster”, Symantec Corporation, January 1998.
[Transitive, 2001] Transitive Technologies Ltd, “Transitive Technologies Ltd.”, http://www.transitives.com/, 2001.
[Transmeta, 2000] Transmeta Corporation, “A New World of mobility from Transmeta”, http://www.transmeta.com/, 2000.
[Wilkinson, 2001] Tom Wilkinson, “KAFFE: A virtual machine to run Java code”, http://www.kaffe.org/, 2001.