Proceedings of the 2nd Java^{TM} Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

A Just-In-Time compiler for memory constrained low-power devices

Nik Shaylor
Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303
USA
nik.shaylor@sun.com

Abstract

Typical just-in-time compilers for the Java™ platform are often too large and too slow to be used in small computing devices such as cell phones or PDAs. In contrast, the JIT described here was targeted for such devices, and was built on Sun Microsystem's KVM product. Key to making the JIT effective were: a pre-compilation transformation of the bytecodes to make compilation easier; compilation of only a subset of the bytecodes to make the production of the system simpler; quick and simple management of the JIT code buffer; and an effective heap comparison technique that greatly aided debugging of generated code. The JIT speeded up execution by a factor of between 5.7 and 10.7. Its implementation required only 60KB of ARM machine code.

1. Introduction

Slow and space-constrained computing devices have tended not to include virtual computing technology. The advent of Java 2 Platform, Micro Edition Connected Limited Device Configuration (CLDC) [1] has changed this because its small size, and natural portability, has attracted a number of cell phone manufactures to standardize on the Java language as the programming language for third party software.

The Java programs being written for these small systems are often not compute intensive, and so can be satisfactorily executed using a simple interpreter loop written in C or assembly language. Nonetheless, a growing number of these programs are games that require better execution performance than can be provided by interpretation alone. Dynamically compiling bytecodes into native machine code addresses this problem. However, different techniques to those used in larger system are needed to make this feasible.

2. Background

2.1 Traditional JIT compilation strategies

JIT compilation has been used in many programming environments [1][2]. This technique has mostly been used in desktop or server class systems, albeit in different ways. Servers typically benefit from high quality code generation, whereas desktop systems tend to be optimized for reduced program startup latency and better interactive response. To avoid the overhead of compiling and optimizing all an application's classes at runtime, a number of incremental compilation strategies have evolved. Many use an interpreter and a compiler. Others use a number of compilers with different levels of optimization. The general strategy of only compiling the "hot" parts of an application will often result in only a small percentage of an application being compiled, thus saving considerable compilation time.

2.2 Typical J2ME programs

In sharp contrast to larger Java systems the number of classes included in the average J2ME program is far lower, typically less than 50. Compute intensive games are also typically small, which lessens the importance of heuristics that select only the performance critical parts of an application for compilation.

3. Design

The system described here used as its foundation the KVM, a small footprint JVM designed specifically for memory constrained devices.

3.1 Interaction with compiled code

Because the JIT was implemented on top of an existing virtual machine, it was easy to build a compiler that produced native code for only some bytecodes, with the interpreter handling the rest. The compiler was thus straightforwardly targeted to the bytecode subset that produced the greatest increase in performance. There were four reasons why a JIT for the complete bytecode set was not implemented:

1, Thread context switching would have had to be performed whilst executing generated native code. This would have added complexity to code generation, runtime support, and the base KVM code. By only performing context switching in the interpreter no changes were made to the way the thread scheduling was done in KVM.

2, The generated machine code would have needed to be more rigorous in the way it dealt with error conditions and other exceptional conditions. As it is, the machine code only needs to check for error conditions. When they occur the error handling bytecodes can be then executed by the interpreter, which then can deal with the details of how the error should be processed.

3, A complete JIT would have required more complicated interactions between the generated machine code and the virtual machine as a whole. For example, the generated machine code could cause the compiler, class loader, garbage collector, or native code to run. In retrospect some of these restrictions were not strictly necessary, but the system probably has fewer undiscovered bugs, and it does not seem to have limited the performance of the type of compute-intensive software that is the target of the design.

4, A debugging technique (discussed below) was used which could not have been employed so easily with a complete JIT.

Therefore the system was designed to allow execution to pass from the compiled code to the interpreter at any time, and also for the interpreter to be able return to generated code in a timely fashion. Additionally, to keep the interpreter from getting trapped in a long loop of bytecodes it was necessary to be able to return to compiled code in the middle of a method as well as at the start.

The basic interpreter loop is as follows:

```
Start:
        Try to enter compiled code.
        Interpret the next bytecode.
        goto Start.
```

The attempt to enter compiled code has several parts. First, if the current method has not been compiled then checks are performed to determine if it can be. Compilation may not be possible for one of the following reasons:

1, The method is a *native* method. (A function written in C or assembler).

2, The method has more than a certain number of parameters or local variables, is unusually large, or has some other condition that basically only occurs in test code and is too troublesome to deal with.

3, There is no available memory for more compiled code.

If a method can be compiled then a jump table is also produced that contains the machine code addresses of a number of entry points into the compiled code, and their corresponding bytecodes addresses. When attempting to re-enter compiled code a search is then made of this table to see if the address of the current bytecode is present. If it is then the corresponding machine code address is found and the compiled code is entered at this place. The jump table contains the addresses of all backward branch targets so it is therefore not possible to be stuck in a loop in the interpreter.

If compiled code is entered, then some amount of computation will result. The return to the interpreter is simply done by having the compiled code update the interpreter's state (instruction pointer, local variables, etc.), and then exiting back to the interpreter loop. The interpreter will then start executing the method where the compiled code left off.

There are several conditions that will cause control to be returned to the interpreter. Some of these conditions exist because the JIT lets the interpreter deal with complex situations (such as exceptions, synchronization, or garbage collection):

1, A native function was called.

2, The thread was blocked because of a monitor enter operation.

3, An object could not be created without running the garbage collector.

4, A method was called but a monitor or an activation record could not be created without running the garbage collector.

5, An operation was attempted that required a class to be initialized.

6, The start of an exception handler was reached.

7, An exception or error occurred. The interpreter always processes these.

8, The part of a method was reached for which no corresponding machine code could be generated.

9, A function was called for which there was no compiled code.

10, A method return was executed but there was no record of the where in compiled code to return to.

11, A method return was executed but there was no compiled code to return to because the code buffer had been flushed.

3.2 Register allocation

It is important that compiled code can be entered at places other than the start of a method. Without this a long running method might be prevented from entering compiled code for a long time. To support this, a table of entry points is generated that contains, at least, an entry for the start of the method and all the backward branch targets.

There is, however, complexity associated with entering compiled code at an arbitrary branch target, because the correct machine state must be established before this can be done. This is complex because a Java interpreter is, most naturally, a stack based computing machine, whereas the generated machine code for most computer hardware will, most profitably, use a register-oriented model.

This problem is neatly avoided in this system by pre-processing the bytecodes into a form where there is never anything on the interpreter's evaluation stack at a place where control might be transferred between the interpreter and compiled code. This essentially means that there is no evaluation stack, and to compensate, additional, local variables are used instead to hold intermediate expression results. This creates a simple one-to-one correspondence between the registers used in compiled code for expression evaluation and the local variables on the Java stack.

In the ARM implementation of the JIT there are 12 registers used by the compiler to represent local variables. Three of these are regarded as general temporary registers. The other nine are used to shadow the first nine local variables of the activation record for the method being executed. The three general registers are used in the cases where a local variable is required that is not one of the first nine. There are three because of the three-address nature of the ARM instruction set.

This simple form of register allocation worked surprisingly well in the tests made upon the system because they rarely used more than nine local variables. Nevertheless, this register allocation is not, in itself, a complete solution.

3.3 Pre-compilation by bytecode transformation

A key part of the JIT design was to split the compilation process into two passes. The first pass transforms the standard, stack-based bytecodes into a simple 3-address intermediate representation in which all temporary expression results are placed into new local variables instead of entries on an evaluation stack. The second pass converts this three-address form into native machine code.

Because of the relatively small number of methods used by programs in small devices, the system simply converts all methods in all classes as they are loaded. Although some care has been taken to make this process fairly fast, it has not been optimized, as its efficiency does not appear to be an important factor. In testing, there have typically only been between 200-400 methods (including the methods of system classes). What *is* an important factor is the amount of temporary memory needed to do the conversion. Consequently, the system only holds one basic block's worth of IR at a time, discarding it after the block is converted.

The basic process being performed in the first pass will be familiar to the author of the simplest compiler. The instructions that use values from the stack are linked to the instructions defining these values; local variables are then assigned the temporary values that connect the instructions together. A number of standard optimizations are then employed.

The resulting 3-address intermediate representation is then converted back into standard bytecodes. Figure 1 shows the bytecode sequence for the expression *a = 1 – (b * c)* before transformation. Figure 2 shows the corresponding bytecode sequence after transformation. It can be seen that a new local variable (number 4) has been added to hold the intermediate result. This type of transformation typically causes the code length to increase by about 30%.

```
iconst_1
iload_1
iload_2
imul
isub
istore_3
```

Figure 1. The bytecodes for the expression *a = 1 – (b * c).*

```
iload_1
iload_2
imul
istore_4
iconst_1
iload_4
isub
istore_3
```

Figure 2. The bytecodes after transformation

It is important to note here is that transformation essentially eliminates the evaluation stack, even though the output of transformation is bytecodes. The instruction granularity of the IR is preserved in the bytecodes, so that the evaluation stack is only used *within* the bytecode implementation of a single IR instruction. Switching between the interpreter and native code is only done at the boundary of an IR instruction, so the use of the stack is never a factor because it is always empty at these points.

The transformation process is currently not very sophisticated, but it is clearly feasible to use simple static analysis to order the local variables so that the most used ones would be assigned to machine registers by the compiler.

An attractive benefit of this transformation process is that it is possible to perform it ahead of time. However, one problem that arises when it is done ahead of time is that the transformer needs to know how many local variables can be mapped into registers. Since not all target architectures are the same, the transformed result cannot run optimally on all of them. A solution would be to do the transformation on the target device as a part of application installation, perhaps done as a background activity (only while the devices is having its battery charged for example), or as a one-time operation by the virtual machine with the results being saved for future execution.

3.4 Code Generation

A basic hypothesis of the design of the JIT was that code generation can be done very quickly. This means that a relatively simple strategy can be used for managing the memory buffer for the generated code, since, if code is discarded non-optimally, it can easily be regenerated. Therefore a single fixed-sized buffer is used for the generated machine code. When it becomes full, the entire contents are discarded. This strategy makes simple a number of standard optimizations. For example, in common with the HotSpot virtual machine[4], monomorphic method invocations are generated for calls to methods where there is only one known receiver type at code generation time. Simply discarding the entire code buffer when the relevant method is subsequently subclassed makes this simple strategy completely safe.

As mentioned above, the current implementation does not retain the IR created by the bytecode transformer. The code generator works by parsing the bytecodes back into three-address form, then emitting the corresponding machine code. On a StrongARM PDA running at 206MHz the process of compiling 250 methods takes less than 100 ms. An experiment using a version of the JIT that generates Pentium instructions reveals the compilation cost to be 75 Pentium cycles for each byte of the input bytecode stream. This experiment has not yet been done on the ARM system; one would expect it to be faster because the Pentium code generator contains a more sophisticated register allocation algorithm than the ARM implementation. Nevertheless, even at 75 cycles per byte it is more than an order of magnitude faster then most other JIT systems. Obviously this comparison is not really fair because it does not take into account the time taken by the bytecode transformer. However, by accepting a small pause when starting up a program (typically only 10% extra startup time) the code generation time is made virtually insignificant, and by saving the transformed bytecodes for future invocations the transformation cost would only paid once. Seventy-five cycles is roughly the time needed to interpret two bytecodes on KVM, so the cost of code generation is quickly amortized during execution. When the size of the code buffer is reduced to about 60% of the code working set of a program the compiler has to run very frequently to regenerate code, because the buffer is constantly being filled and then flushed. Nevertheless, even in this case the execution time is typically two to three times faster than running the bytecodes using the interpreter alone.

3.5 The format of activation records

The KVM implementation is a very literal encoding of the Java virtual machine specification. Five significant virtual machine registers are maintained, and the processes of method invocation and method return are quite lengthy. It was found that replicating this very literal behavior in compiled code slowed down method invocation significantly, so a different calling convention is used in compiled code. This calling convention was designed so that return to the interpreter can occur at almost any place in a method. This led to the implementation of a stack frame *converter* that can change the activation records from the form created by compiled code back to the form used by the interpreter. It also necessitated that local variables held in machine registers be saved and loaded into the activation record by the caller instead of by the callee (which would be more efficient because registers unused by the callee do not require saving and restoring). This issue seems not to have hurt performance too badly, probably because all the registers can be saved using a single ARM instruction. The stack frame converter is not particularly complex, but various performance tradeoffs were encountered during its implementation. At one point the normal calling procedure was made faster at the cost of more work being done by the stack frame converter. This considerably lengthened the time it took to switch back to the interpreter. This in turn caused the frequency of this operation to be more of a factor in overall performance. It was no longer possible to 'quickly' switch back to the interpreter to execute an unimplemented bytecode. In early versions of the implementation, execution would switch hundreds of thousands times per second. It thus became necessary to compile many more types of bytecodes to get the performance up. In the end the benchmark programs only switch about every 10 milliseconds.

3.6 Threading issues

Since the KVM is relatively simple and portable, it has its own internal “green” thread implementation. This made possible a considerably simpler JIT design because there are no native thread preemption or SMP issues to be concerned with.

However, thread switching is an issue. In the current JIT implementation, if a compiled thread goes into an infinite loop the virtual machine will hang forever. Although this does not appear to contradict the Java language specification it is not an acceptable situation. A solution to this would be to dedicate a machine register to be used as a counter decremented at each backward branch. When the counter reaches zero execution would pass to the interpreter so that a context switch could be performed.

3.7 Native methods

Native methods are primitive functions that are usually written in C or assembler. In KVM they naturally fall into two categories, those that take a "long" period of time because they are (typically) waiting for I/O to complete, and those that do not. The former are normally coded in such a way as to cause KVM to context switch, if possible, to another thread of execution. For this reason, compiled code cannot, generally, call native methods directly. However, it was found to be very important for compiled code to be able to call certain non-blocking functions for performance reasons. The compiler was modified to generate direct calls to the code for `System.arraycopy()`and to the graphic drawing functions (although the latter were not called when running the benchmark programs).

3.8 Debugging

The implementation was quite easy to debug due to a new dual execution feature, which enabled straightforward comparison of the results of interpretation and compiled execution. Before the execution of each basic block of machine code the heap was saved. A single basic block's worth of machine code was then executed (this was achieved by the code generator inserting special code to return to the interpreter at the end of each basic block). The heap was then saved a second time, and then the original heap was restored. The same basic block's worth of machine code was then executed on the interpreter using the bytecodes that were used to generate the machine code. Comparing the heaps after two such executions then verified (or not) that the interpreter and the compiled code had done the same work. When a word in object memory was found to differ, the address was noted, and then the garbage collector was run in a special mode that just looked for the given address. If the structure containing it was found the structure was dumped out. Having special annotations for stack structures made identifying corrupt local variables very easy. The vast majority of code generation bugs showed themselves this way.

Although the target platform was the Compaq Pocket PC, most of the debugging was done using the GNU gdb debugger on a workstation. The debugger can be built (with considerable difficulty!) so that it executes code using a machine code simulator. The ARM simulator included with gdb proved very reliable and a great deal faster then the tools available for the Pocket PC, which exhibited a painful delay of 15 to 20 seconds when stepping from one assembly instruction to another.

The combination of using the gdb ARM emulator and running the compiled code one basic block at a time often proved too slow to be practical. In these cases another strategy was employed. Internally, functions with names like Asm_MoveRR() were used to encapsulate the generation of code (this example would generate a register-to-register copy). It was very easy to have these routines generate equivalent machine code for a workstation so the program could be tested without the ARM simulator. Debugging the resulting workstation machine code was not easy, but the much increased speed of execution made it worthwhile.

4. Performance evaluation

The performance of the JIT was measured and compared to the performance of the KVM interpreter. All the optional performance features of the KVM were enabled in order to make the comparison as meaningful as possible. The JIT implementation was based on version 1.0.2 of the KVM. A number of enhancements were subsequently made to this code to incorporate all the significant interpreter performance improvements present in KVM version 1.0.4.

In addition to determining performance improvements due to compilation, measurements were made to investigate the effects of code buffer size on performance. These effects are clearly heavily dependent on the size of the application, but the experiment of particular interest was to see how the system performed when the buffer size was reduced to below that needed to contain the whole application.

4.1 Benchmark tests

In order to evaluate the effectiveness of the design three real-world Java programs were used to test the system.

1, A graphical program specifically written to demonstrate the effect of the JIT. This program renders a number of rotating cubes. (In normal operation, with graphics enabled, it was possible to navigate through this virtual world.)

2, The DeltaBlue constraint solver.

3, An MPEG-1 video decoder.

All three of these programs were run in a special mode in which graphical output was disabled in order that only VM execution time would be measured.

4.2 Results

The three benchmark programs were run on the interpreter-only version of the system and on the JIT-enabled version using a number of different code buffer sizes. Table 1 shows the execution times for the benchmarks in seconds (not every test was performed with the smallest buffer sizes). The same information is presented in figures 3, 4 and 5 in graphical form.

Table 1. Benchmark execution times in seconds

	Cubes	DeltaBlue	MPEG
Interpreter	25.8	6.04	47.5
JIT w/128KB	4.3	1.05	4.4
JIT w/64KB	4.3	1.09	4.8
JIT w/48KB	4.3	1.03	5.8
JIT w/32KB	4.3	3.3	12.5
JIT w/24KB	58	5.3	39
JIT w/20KB	89	83	49
JIT w/18KB	510	89	90
JIT w/16KB	567	163	132
JIT w/14KB		188	168
JIT w/12KB		173	206
JIT w/10KB		170	351
JIT w/8KB		318	

The first observation is that with a large JIT buffer of 128KB the performance of the JIT-enabled version was faster by a factor of between 5.7 and 10.7.

The second observation is that the performance of all the benchmarks remained roughly the same until the buffer size was reduced to 32KB, after which execution times rose significantly due to the code generator running periodically. Code generation has two cost components. One is the basic execution time of the code generator. The other is the cost of converting the stack from the format used by compiled code to the format used by the interpreter, which is done when the code generator is invoked when compiled code is running.

Note how performance degrades differently with the three benchmark programs. The performance of the cubes program drops off sharply below 20KB, the DeltaBlue program demonstrates two distinct plateaus, and the MPEG program shows a more linear decline.

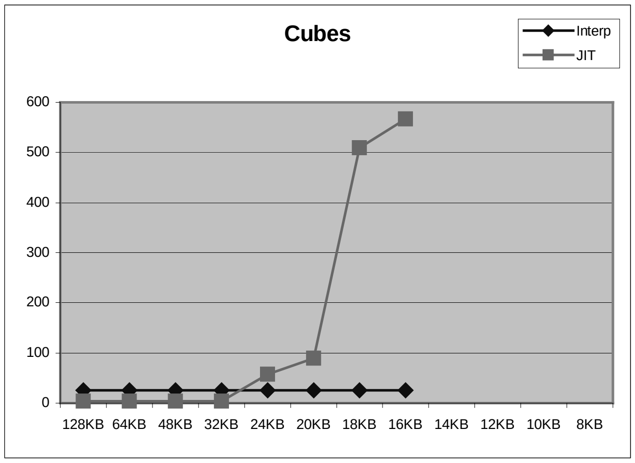

Figure 3. Cubes Execution time.

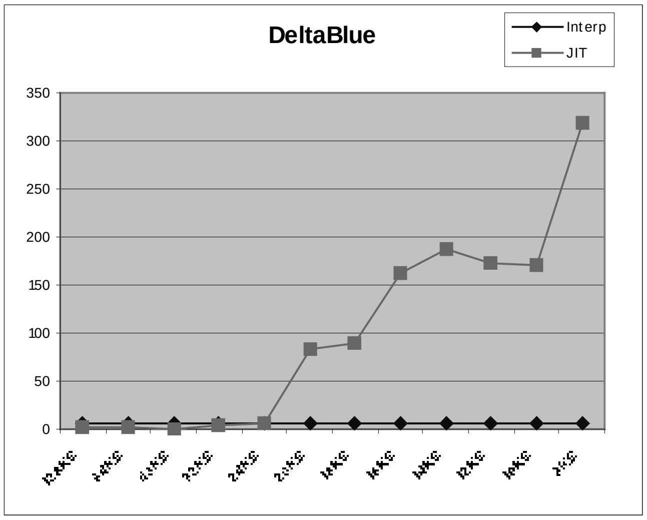

Figure 4. DeltaBlue Execution time.

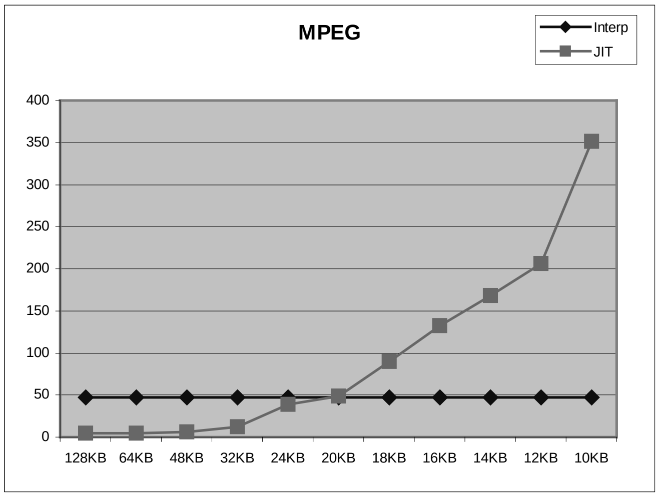

Figure 5. MPEG Execution time.

5. Conclusion

This paper describes a small JIT specifically designed to run Java games on a handheld device. The implementation size is only 60KB of ARM machine code. Although the quality of code produced by the JIT compiler is relatively low, the system nevertheless runs programs five to ten times faster than the KVM interpreter. The noteworthy features of the design include: transforming the input bytecodes into a form that does not use the evaluation stack for temporary results, making it easy to switch between interpreted and compiled forms of execution; switching to the interpreter for the handling of complicated bytecodes, such as those involving exception handling and synchronization; and favoring fast compilation speed over sophisticated code buffer management. The ultimate result of these features has been a simple overall design.

6. Acknowledgements

Thanks go to Kay Neuenhofen who wrote the Cubes programs and the dynamic assembler used in the project, to Kinsley Wong who greatly assisted in getting the ARM port to work, and to Doug Simon, Mario Wolczko, and William Bush in preparation of this paper.

7. References

[1] JSR-000030 J2ME Connected, Limited Device Configuration. http://jcp.org/aboutJava/communityprocess/fnal/jsr030/index.html

[2] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Symposium on principles of Programming Language,* January 1984.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Conference on Programming Language Design and Implementation*, July 1989.

[4] The Java HotSpot Virtual Machine Architecture, March 1998. See whitepaper at http://java.sun.com/products/hotspot/