Efficient Implementation of Java on the ST20

Stuart Menefy
stuart@bristol.st.com

SGS-THOMSON Microelectronics Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
United Kingdom

Abstract

Recently the World Wide Web has become far more interactive through the use of Java, Sun Microsystems programming language, which allows programs to be executed by a WWW browser. To make these programs portable across a wide range of architectures, they are downloaded in the form of ‘byte-codes’, the instruction set for a virtual machine.

Current implementations interpret these bytes-codes, however we believe that SGS-THOMSON’s ST20 CPU will be able to translate the byte-codes into native instructions as the program is downloaded, and then execute them at full speed.

The paper will discuss some of the issues involved in performing the translation from byte-codes into native ST20 instructions, and how this could be used to form the heart of an Internet terminal or Network Computer.

1 Introduction

Much of the recent growth in the Internet has been brought about through the introduction of the World Wide Web. This has proved to be a great way of distributing information which consists of text or simple graphics, however for more advanced content which requires real-time updates, or interaction with the user, it is severely limited. One way to overcome this is to download a program to the user's computer, which is then responsible for displaying the data. This gives the content author great flexibility. However to do this effectively, issues of portability, security and performance must be addressed, all areas which were considered in the design of the language Java, and its execution environment.

Once the issues of downloading code have been addressed, it becomes possible to envisage the downloading of whole applications. These could provide simple facilities such as word processing or e-mail, on a machine which only has to run a simple browser and thus does not need the complexity or cost of a personal computer.

SGS-THOMSON Microelectronics provide a family of microprocessors collectively called the ST20, which are based around a CPU core and various peripherals. What makes the ST20 interesting is that the CPU architecture shares many features with the Java virtual machine, making it possible to implement Java efficiently.

This paper is divided into six sections. Following this introduction, the next two sections are overviews of Java and the ST20. The main part of the paper describes the compilation process which is used to convert a Java application into native ST20 instructions. This is followed by a discussion of some other features of the ST20 which can be used to support a Java execution environment, and finally the conclusion.

2 Java

Java is a programming language, similar to C++, but with many features changed to make it easier to write correct, reliable programs. What makes Java interesting is not only the language itself, but that it was designed to execute in a particular environment, in which security and portability were major issues. This is exactly the situation found on the World Wide Web, and why Java has been proposed for this area[1].

2.1 Java and the World Wide Web

Java is designed to allow users to download arbitrary programs from anywhere on the Internet and run them on their local machine, without having to be concerned about compatibility, portability or security. It achieves this in several ways:

- Java has been designed to make writing reliable programs easier. There is no support for pointers, array accesses are fully checked, and dynamic memory is garbage collected.
- Java is compiled into an architecture neutral object file format, which can be run on any system which supports a Java execution environment.
- The execution environment is strictly defined, with no implementation dependant features. Every system should provide the full run-time library.
- Every program is verified when it is downloaded, to ensure that it only accesses resources which are explicitly allocated to it, and at run time, file and network access is strictly controlled[4].

For more details on the Java language, see [2].

2.2 The Java Virtual Machine

The Java Virtual Machine (VM)[3] is a mythical CPU which executes Java byte codes as its native instruction set. This machine is the target for the Java language compiler (and could be for any other language, within the constraints of the VM definition).

This machine has several interesting features:

- A 32 bit architecture, with instructions defined to operate on bytes, 16, 32 and 64 bit integers, and 32 and 64 bit IEEE floating values. 64 bit numbers occupy two adjacent locations (MSB/LSB ordering is undefined, and should be impossible to determine).
- A zero operand instruction set; all expressions operate on a stack, the offset into which is implicit in the instruction. The compiler generates code for an arbitrarily deep stack, the maximum depth of which is recorded in the object file.
- Each function has a local variable area, which is addressed using integer offsets from its base.
- All inter-module calls are implemented as methods of an object. Parameters for the call are loaded into the stack, and stored into the called method's local variable area.
- Exception handling is defined as part of the instruction set and run-time.
- Multi-threaded applications are supported through methods in the run time library and special synchronisation instructions.

When combined with the core run-time library, the VM provides a portable execution environment for Java applications.

3 ST20

The ST20[5],[6] family is a range of 32 bit microprocessors designed for use in embedded applications. Each is based around a micro-core which provides a 32 bit CPU, and a collection of on-chip peripherals which typically include an external memory interface, interrupt controller, fast SRAM or cache, and application specific blocks such as UART's, IEEE 1284 interface or DVB descrambler.

Features of interest in the context of this paper include:

- A small general purpose register set organised as a three deep stack (**Areg**, **Breg** and **Creg**).
- A very compact instruction set in which most instructions are one byte long. Expressions take no arguments and implicitly operate on the register stack.
- A local workspace[1] addressed as offsets from a dedicated workspace pointer register. Later versions of the ST20 core cache the bottom (active) portion of the workspace providing 0 cycle loads in many cases.

1. The workspace takes the place of a stack on other processors. The term workspace is used to avoid confusion with the register stack (and the Java stack).

- A microcoded scheduler which provides a fast context switch (500nS) and support for extending this with a software OS by trapping certain scheduling operations.

As can be seen, there are many areas in which the ST20 is close to the Java virtual machine. The remainder of this paper discusses how these features can be exploited to give efficient execution of Java programs.

4 Compiling Java Byte Codes

The first generation of Java execution environments interpreted the Java byte codes. This allowed the interpreter to be portable, but execution speed was a problem. In order to efficiently execute Java bytes codes, they must be translated into the processor's native instruction set.

Java's object oriented structure lends itself to loading the program in blocks, one class at a time. This is ideal when a program is being loaded over a slow network connection, because the program can start executing with just a minimum amount of code loaded, and pull in the remainder on demand. This can be extended by allowing the execution environment to delay translating the byte codes into native instructions until just before a classes method is called. This has been termed just-in-time (JIT) compilation, and requires a rather different compiler from traditional ones.

A JIT compiler must execute quickly, while still generating reasonable quality code. The compilation process must be invisible to the user, if the system pauses on entry to a function while the compiler is called, then it will be useless. However, many of the time consuming activities which a compiler normally has to perform such as parsing the input language, building a symbol table and syntax checking have been performed by the Java compiler[1], which allows the JIT compiler to perform many of its optimisations in advance, so the compiler simply has to glue together many precompiled instruction sequences. However for optimal code things get slightly more complicated.

4.1 Development

Development of the Java execution environment for the ST20 will start from the current interpreter produced by Sun Microsystems, which is written in C. This should allow a phased approach to be used, first porting the interpreter, which will then allow development of the run-time libraries to be carried out at the same time as work on improving the performance of the VM. The final system will consist of a mixture of pre-compiled C and JIT compiled Java. These two environments can call each other freely, to support Java's `native` methods, and some of the complicated Java byte codes will be compiled into calls to C functions, rather then implemented in-line.

4.2 Memory layout

Deciding how to store the Java stack and local variables in memory is crucial to the implementation of the virtual machine. In particular, the representation of the stack in memory must be simple, as most instructions push or pop the stack.

1. The byte code verifier will also guarantee that the program is still correctly formed when it is loaded.

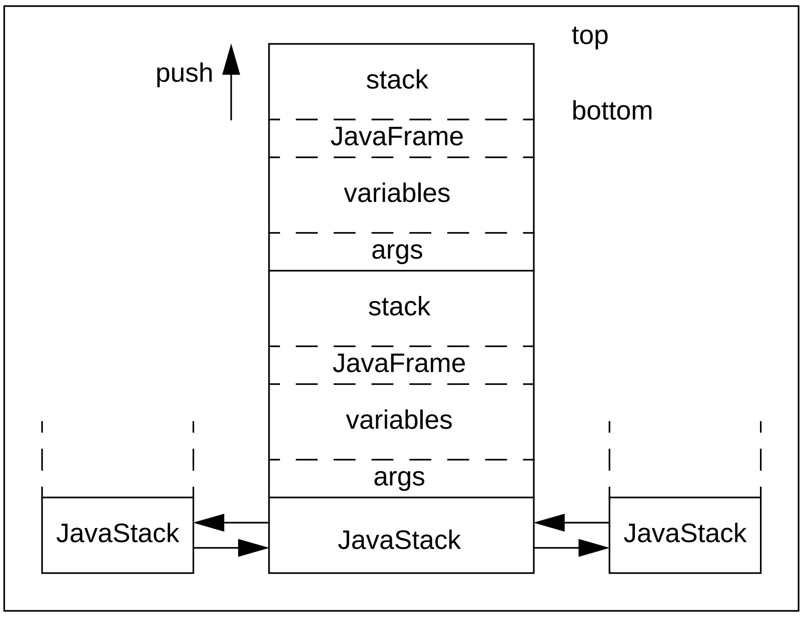

Figure 1 Interpreter stack and variable layout

As shown in figure 1, the interpreter stores the local variables and stack in a frame, one per method. Several frames are stored together in a data structure called a JavaStack, and these are linked together to allow stack extension, without requiring hardware support.

JavaStack and JavaFrame are the C structures which contain administrative information. Storing JavaFrame between the variables and stack which it refers to allows method parameters to be passed without having to copy the data, except when moving between stacks. Both the variables and stacks are accessed as C arrays, so effectively the stack grows up, with the top of the stack being at the higher address. The stack pointer is maintained in the JavaFrame.

For development purposes it is desirable for the compiler to be compatible with the interpreter. However following this exactly would restrict the opportunity for optimisations, so, in several stages, a more ST20 sympathetic implementation should be brought in.

To minimise the overhead of accessing the stack, the stack and local variables must be mapped into the ST20's workspace. This makes it very simple to access both stack and variables, short, fast instructions such as `ldl` (load local) can be used, which operate implicitly on the local workspace. In a first version this will use exactly the same data structures as the interpreter, simply switching from the normal C stack to the Java stack when executing compiled Java code. This will simply involve setting the workspace pointer register to point to the base of the current frame as part of the preamble to the compiled code, and swapping it back before executing C code.

There is no need to maintain an explicit stack pointer while executing compiled code. The compiler can simply track the current stack depth as instructions are compiled, and adjust the address used to access a given stack location accordingly. Thus the stack pointer maintained in the JavaFrame structure only needs to be updated when returning from compiled code to C.

In a later version the need to switch stacks may be removed altogether, using a single stack, which will contain a mixture of both C and Java stack frames. This should give a slight performance gain, and reduce memory consumption.

One problem with following the interpreter memory layout is that because the ST20 architecture uses a falling stack, the workspace cache, which caches the 'bottom' 16 words of the workspace, will hold the arguments and local variables. It would be much more useful if they could hold the active portion of the Java stack. To overcome this three changes are required:

- The layout of a frame must be reversed, so that the stack is at the bottom of the memory and not the top.
- The stack, and therefore the stack frames within the JavaStack structure, must fall rather then rise.
- The workspace pointer must track changes to the Java stack pointer so that the currently active portion of the stack remains in the cache. Changing the workspace pointer involves generating an `ajw` instruction, so workspace should be allocated and freed in blocks, with some hysteresis, to avoid repeatedly expanding and contracting the stack.

This means that given the size of workspace allocated for the stack (`stack_size`) and the current depth of the stack (`stack_depth`), the offset of the bottom (the active part) of stack is:

```
stack_base = stack_size - stack_depth
```

and the offset of a stack item (numbered from 0 for the bottom of the stack) can be determined by:

```
stack_offset(item) = (stack_size - stack_depth) + item
```

Similarly a variable is located at offset:

```
variable_offset(item) = stack_size + item
```

For example, Table 1 shows the effect of the instruction sequence:

```
bipush 4
iadd
```

when the stack initially holds three values (top to bottom: 8, 10, 20, indicated by stack[2] down to stack[0]) and five words have been allocated for the stack. Initially stack[0] (holding

Table 1: Stack usage example

WS	Initial value	After `bipush 4`	After `iadd`
4	stack[2]: 8	stack[3]: 8	stack[2]: 8
3	stack[1]: 10	stack[2]: 10	stack[1]: 10
2	stack[0]: 20	stack[1]: 20	stack[0]: 24
1		stack[0]: 4	
0			

20) is at workspace 2. The effect of pushing value 4 causes the stack depth to increase to four, and the value 20 is now at stack[1], but remains at address 2. Similarly the addition pops two values and pushes one, so the stack returns to a depth of three, and bottom of the stack is at location 2.

4.3 Registers

It is intended that the ST20's register stack will in effect act as a cache for the currently active portion of the Java stack. Usually this will be the bottom three elements of the stack, so **Areg** will hold stack[0], **Breg** stack[1] and **Creg** stack[2]. This means that many Java instructions can map directly onto their ST20 equivalents, and many sequences will butt end to end, without any register shuffling required.

However, because the Java stack is of arbitrary depth, and the ST20 stack is only three deep, the generated code must also ensure that instructions which push new values onto the stack do not result in valid register contents being lost, and similarly popping values off the stack may mean that stack entries further down the stack need to be reloaded.

For example, the code sequence:

```
static int test(int a, int b, int c, int d)
{
      return a+(b+(c+d));
}
```

results in the byte codes:

```
iload_0
iload_1
iload_2
iload_3
iadd
iadd
iadd
ireturn
```

If this was called as `test(5, 6, 7, 8)`, this would result in the stack usage shown in table 2. For the initial three loads, the registers are used to hold the stack contents directly,

Table 2: Register and Stack usage example

WS / Regs	After initial 3 `iload`'s	Before `iload_3`	After `iload_3`	After first `iadd`	After second `iadd`	Before third `iadd`	After third `iadd`
3		stack[2]: 5	stack[3]: 5	stack[2]: 5	stack[1]: 5	stack[1]: 5	
2							
1							
0							
C	stack[2]: 5	stack[2]: 5	stack[2]: 6				
B	stack[1]: 6	stack[1]: 6	stack[1]: 7	stack[1]: 6		stack[1]: 5	
A	stack[0]: 7	stack[0]: 7	stack[0]: 8	stack[0]: 15	stack[0]: 21	stack[0]: 21	stack[0]: 26

and nothing is written to memory. However the fourth load would overflow the stack, so **Creg** is written into memory, prior to the load. The first two additions can be performed entirely using the stack, and so it is not until just before the third addition that the value has to be loaded back from memory.

A secondary, but similar problem, is that some Java instructions will translate into multiple ST20 instructions, which will require additional registers for temporary storage. In this case registers may have to be written back to memory when it does not appear necessary. Both of these problems are controlled by monitoring register usage during the compilation process, as described in section 4.4.

4.4 Compilation process

The objective of the compilation process is to reduce every byte code to a canned sequence of instructions, which require a minimal amount of additional 'glue' instructions to combine them. These additional instructions are required to compensate for the ST20's register stack being smaller then the VM stack, or registers getting corrupted while executing compiled instructions.

The compiler will be driven by the byte code input stream. This will perform table look up, using the byte code as a key, and determine four pieces of information:

1 Which Java stack entries need to be in each register before the call or if the register will be preserved across the call.

2 What will be in each of the registers after the call. This can be either a Java stack entry (which may or may not have been modified), the unmodified contents of a register prior to the call, or an undefined value.

3 The effect on the Java stack (a push or pop, and by how much).

4 The ST20 instructions to implement the byte code.

This table is generated by running a script over a text file which contains the above information. For example, the Java byte code `iadd` is described by:

```
iadd                            /* integer add */
      PRE:    Areg=stack[0]     Breg=stack[1]    Creg=any
      POST:   Areg=stack[0]*    Breg=Creg        Creg=undefined
      STACK:  -1

      add
```

This specifies that before the instruction, the top two elements of the stack must be in **Areg** and **Breg**, and after the instruction, **Areg** will hold the top element of the stack, which has been modified (hence the "`*`"), **Breg** will hold what was in **Creg** before the call, and **Creg** is now undefined. Only one ST20 instruction is required, `add`.

The glue instructions are generated in a similar way. What is required is a set of instructions which transform the current state of the registers (which includes not only which Java stack location it currently holds, if any, but also if the value has been modified, or could still be obtained from memory) into the required state before the byte code instructions.

A trivial solution to this is to write all modified registers to memory, and load the new registers. However this would generate too much code, and so a two step process has been devised:

1 Generate code to save any registers to memory which need saving.

   Registers only need saving when they will be pushed out by the loading of other arguments or because they will be corrupted by the instructions used to implement the byte code. Any registers which hold useful values and which will not be corrupted should remain in case they will be useful in subsequent instructions.

   Thus, having determined how many registers need to be saved, the compiler first of all discards any registers whose contents are unmodified (and so can be simply reloaded from memory), and if more registers need to be saved, generates the code needed to save them to memory.

2 Generate code which will convert from the current register contents to the desired ones.

   This can be a simple look up operation, because for each of the three registers, there are five possible sources of data: data from one of the three registers, data from memory, or don't care. This gives a maximum of 125 possible code sequences, although a very large number will be identical.

So for example, if the compiler was about to generate code for the `iadd` instruction described above, and the current register contents were:

```
Areg=stack[0]   Breg=stack[2]*  Creg=stack[3]
```

then one register needs to be saved to allow the loading of stack[1]. **Creg** would be chosen because it can simply be discarded, not having been modified, and so no code needs to be generated. This changes the register contents to:

```
Areg=stack[0]   Breg=stack[2]*  Creg=undefined
```

and so the required function needs to perform:

```
Areg=Areg       Breg=stack[1]   Creg=Breg
```

which is simply:

```
ldl   stack[1]
rev
```

4.5 Jumps

The ST20 jump instructions (both conditional and unconditional) are such that it will be difficult to maintain any values in the ST20 registers over a jump. For `cj` this is because it leaves 0 in **Areg**, which is difficult to include in the instruction description table, and `j` is a timeslice point, which if taken will corrupt the registers anyway.

This is in addition to the problem of reconciling the contents of registers which will result from jumping to a location, or reaching it through executing the previous instruction.

Thus the target of a jump must be flagged (possibly by the insertion of a dummy instruction which is marked as ‘corrupts all registers’ but which requires no code) so that after the previous instruction all valid registers are flushed to memory, and the next instruction has to reload them from memory.

4.6 Instruction counts

Initial results indicate that very few ST20 instructions are required to implement Java byte codes. The two tables described in section 4.4 have been generated for the ST20C2, and the frequency with which instruction sequences of a particular length appear are shown in tables 3 and 4.

Table 3: Number of ST20 instructions required to implement a Java byte code

Number of ST20 instructions	Frequency
0	3
1	64
2	37
3	9
4	7
5	2
7	1
8	3
9	6
10-13	11
Function call	55

Table 4: Number of ST20 instructions required to convert the registers between byte codes

Number of ST20 instructions	Frequency
0	8
1	21
2	35
3	31
4	22
5	7
6	1

Table 3 shows that for most byte codes (72%), at most two ST20 instructions are required to implement the byte code. It is important to note that these figures also include all error and array bound checking.

There are currently a number of byte codes which are not implemented in-line (indicated by the final “Function call” figure). These are predominantly floating point operations, which, because the ST20 has no hardware floating point unit, are implemented as calls to library routines.

Table 4 shows how many ST20 instructions are required for all of the 125 possible transformations required to convert the stack from its contents following one byte code prior to the next. Again the majority (51%) require at most two ST20 instructions, which should be the most common sequences.

4.7 Security

The Java security mechanisms involve checks at both and load time and run time:

- When a Java class is loaded it is verified to ensure that the code is valid. This includes checks that the class is not corrupted or invalid, that there are no stack under or overflows, that methods are called with the correct arguments, that all instructions operate on operands of the correct type and that there are no invalid type conversions.
- At run time low level checks are performed by the interpreter which could not be performed at load time, for example checks for array bound violations or null object dereferences. In addition the run time library performs higher level checks such as attempts to access files or network connections.

Translating Java byte codes into native instructions will not make any difference to these checks. The translated code must perform the same run time checking as the interpreter, and will attempt to use explicit checking instructions where possible (see section 5.2).

However the possibility of introducing a new security vulnerability through a bug in the translator must be considered. By keeping the code generator as simple as possible, and using pre-computed code sequences which can be fully tested, this risk is greatly reduced.

5 Other features

There are a number of other features which make the ST20 an interesting target for Java.

5.1 Micro-code scheduler

The ST20 has a simple scheduler built into the processor’s instruction set. This provides timeslicing of processes in hardware, together with simple interprocess communication and synchronisation primitives. In addition a trap mechanism allows scheduling events to be selectively intercepted in software to allow the scheduler to be extended. In this context the small register set is an advantage because very little state has to be saved on a context switch.

Using the hardware scheduler as a basis, a light-weight operating system has been written with the Java execution environment in mind. This extends the hardware scheduler with multiple priorities, and provides the synchronisation primitives required by Java, exploiting the hardware scheduler wherever possible.

5.2 Exception handling

All the ST20 CPUs provide standard error checked arithmetic, and traps for dealing with error conditions. These can be used as the basis for the Java runtime exceptions, which in many cases will mean that no additional code has to be generated for error checking.

For those areas which require explicit error checking, the ST20C2 core provides a couple of special error checking instructions:

- `ccnt1` was designed to check the length of messages, which must be greater then zero, and less than the specified maximum length. By specifying that the messages can be of any length, this acts as an effective check for zero, which is the nul object reference.
- To provide checked array accesses, the `csub0` instruction will check that an array index is less than the length of the array.

Both of these instructions will generate a trap when the condition fails, otherwise they leave the value being tested in **Areg**, so it can be used by the next instruction.

When a trap occurs the runtime system will receive the trap and needs to determine which byte code the erroneous instruction corresponds to, and thus which exception handler should deal with it. This means that while compiling the Java byte codes, the compiler must also record the addresses of code. This simply involves updating the existing exception handling table with the addresses of compiled instructions.

5.3 Debugging

Current Java debugging technology is fairly rudimentary. It is hoped that while generating the addresses of instructions for the exception handling mechanism, it will also be possible to generate symbolic debugging information. This will involve combining the addresses of the compiled code from the compiler with the debugging information already present in the Java class file (the Java equivalent of an object file). This information could then be shipped back to the debugger, which would allow symbolic debugging of Java programs.

6 Conclusion

The similarity of the ST20 architecture to the Java Virtual Machine allows some functionality to be implemented very efficiently. In many cases it is possible to keep stack values in registers across multiple byte codes. However the need to generate additional instructions in the compiled code to reload the stack from memory means that in these cases some efficiency is lost.

Most of the other work which has been done on Java JIT compilers target the Intel x86 family. Currently these report that the generated code is around three times larger than the byte codes and runs between 5 and 17 and times faster depending on the mix of Java and native code [7]. Comparing the code generated for the ST20 with that which would be required on a conventional register based machine, such as the x86, it is likely that the ST20 will pay a penalty due to the small number of values which can be kept in registers. However generated code is likely to be smaller for the ST20 compared to that required for a conventional processor, because so many instructions operate on the stack and so do not require arguments, which is exactly how Java programs obtain their compact encoding. This is especially important for embedded systems with limited memory spaces, and also improves performance because less code has to be fetched from memory.

The ST20 family has been designed for use in embedded systems, and provides a low cost solution for many markets, including set top boxes, network terminals and mobile phones. It is in exactly these areas that the use of Java is being examined, and the ability to run it efficiently on the ST20 makes an ideal partnership.

References

[1] The Java Language Environment White Paper, James Gosling and Henry McGilton, Sun Microsystems Computer Corporation, October 1995 (`ftp://ftp.javasoft.com/docs/whitepaper.*`)

[2] The Java Language Specification 1.0 Beta, Sun Microsystems Computer Corporation, 30 October 1995 (`ftp://ftp.javasoft.com/docs/javaspec.*`)

[3] The Java Virtual Machine Specification, Release 1.0 Beta DRAFT, Sun Microsystems Computer Corporation, 21 August 1995 (`ftp://www.javasoft.com/docs/vmspec.*`)

[4] HotJava: The Security Story (`ftp://www.javasoft.com/docs/security.ps.Z`)

[5] ST20-TP2 Datasheet, SGS-THOMSON Microelectronics, January 1996 (42 1674 01)

[6] ST20C2/C4 Core Instruction Set Reference Manual, SGS-THOMSON Microelectronics, January 1996 (72-TRN-273-01)

[7] Interview with Régis Crelier, Borland International, Inc., 10 April 1996 (`http://www.borland.com/internet/java/interviews/regis2.html`)