Exploiting FPGA Concurrency to Enhance JVM Performance

James Parnis **Gareth Lee**

School of Computer Science & Software Engineering
The University of Western Australia,
35 Stirling Highway, Crawley, W.A. 6009, Australia.
Email: {parnij01,gel}@csse.uwa.edu.au

Abstract

The Java Programming Language has been praised for its platform independence and portability, but because of its slow execution speed on a software Java Virtual Machine (JVM), some people decide to use faster languages such as C. Building a JVM in hardware is an obvious solution to this problem. Several approaches have been taken to try to achieve the best solution. One approach is by reducing the number of Java instructions a program has to execute along with directly executing instructions in hardware, for example on a Field Programmable Gate Array (FPGA), to increase the execution speed. Another approach is the translation of Java Byte Codes into native code by a FPGA and then executing the native code on a conventional CPU. Others have developed a multi-threaded JVM and exploited the parallelism offered by a FPGA and have specifically designed the JVM for real-time systems. This paper compares and contrasts all these approaches and then argues that the parallelism of a FPGA should be exploited in the most general way possible by not restricting the threads of execution to a specific task. It gives a method for building such a JVM and also some results from a JVM that was built using this method. The paper concludes that this approach should be taken to build a system that is capable of running threads of a Java program in parallel.

Keywords: Field programmable logic, FPGA, Java virtual machine.

1 Introduction

Field Programmable Gate Arrays (FPGAs) are a relatively new technology. Due to the fact that they have not been around too long, their development is not as advanced as general purpose processors. Leaving this aside, FPGAs have the potential to make a big impact on the computer market. FPGAs are already being used in applications today such as communication satellites, portable electronic devices, and devices that require their circuitry to be upgraded on a regular basis. FPGAs are ideal for devices that run on batteries due to the low power consumption to work ratio.

There is an abundance of research concerned with FPGAs — one of these research areas is building a conventional Central Processor Unit (CPU) on a FPGA. Some CPUs have been specifically designed to act as a Java Virtual Machine (JVM). Researchers have built JVMs that are very similar to software JVMs, but have added a few optimisations such as compressing multiple program instructions into one instruction and building the JVMs from hardware (Schoeberl 2003) instead of using the conventional software approach. Alliance Core (Digital Communication Technologies n.d.) adopt a very similar approach, but also add in extra functionality such as support for other programming languages such as C. There has also been research into building a JVM that can run four threads of execution in parallel, but with the limitation that these threads are assigned to pre-determined tasks, such as to interrupt handling (Kreuzinger, Brinkschulte, Pfeffer, Uhrig & Ungerer 2003). Radhakrishnan et al. (Radhakrishnan, Bhargava & John 2001) build a system similar to a software Just-In-Time (JIT) Compiler (Cramer, Friedman, Miller, Seherger, Wilson & Wolczko 1997), but on a FPGA. Ha et al. (Ha, Vanmeerbeeck, Schaumont, Vernalde, Engels, Lauwereins & De Man 2001) build a JVM on a FPGA to speed up the execution of web based Java Applications.

This paper discusses how people have taken advantage of FPGAs to build JVMs. It will first discuss the advantages of a FPGA and how these advantages have been used. It will then go on to discuss the limitations and how they have been dealt with. The applications of using a FPGA for a JVM is also looked at. It will then argue that the parallelism of a FPGA should be used to create multiple threads of execution and that they should not be specialised for a specific task. In the Method section the process of how to build a JVM that can run threads in parallel using the programming language Handel-C is described. The paper then moves on to show some results that have been obtained from a JVM that has been built using Handel-C. It will then conclude that building a JVM for a FPGA using tools such as Handel-C does work and that it should be investigated further.

2 Field Programmable Gate Arrays

FPGAs are composed of a large matrix of configurable logic blocks (CLBs), as shown in Fig. 1. Each CLB is composed of three distinct components:

- One or more functional units which can be programmed to implement arbitrary boolean functions;
- One or more latches which allow state to persist over time; and
- A programmable crossbar router which allows each CLB to connect its inputs and outputs to its neighbours (or via its neighbours to CLBs further afield).

In addition the FPGA has I/O blocks (IOBs) around the border of the matrix to direct input and output signals to the external world.

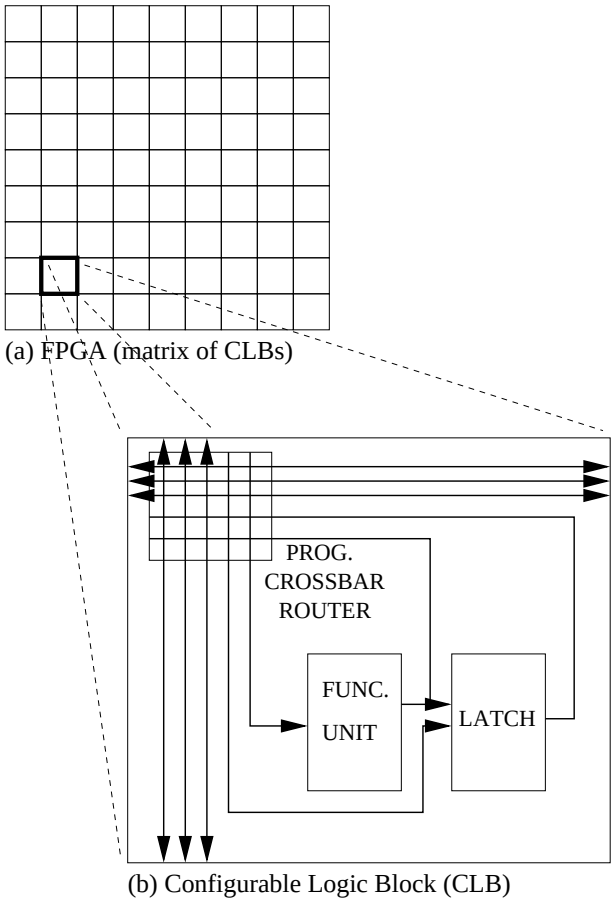

Figure 1: (a) A Simplified Field Programmable Gate Array; (b) One Configurable Logic Block within the FPGA.

These are all the elements needed to build a universal computer. For example, grouping together a number of functional units and configuring them to behave as full-adders allows an ALU to be constructed; similarly grouping together latches allows arbitrary registers to be allocated. Each of the CLBs operates concurrently, as do all the communication signals between them. Reconfigurable computers can thus be programmed to execute massive amounts of computation per clock cycle; they can be configured as SIMD systolic arrays or vector processors, but equally well as MIMD heterogeneous systems.

The behaviour of the functional units and the connections between CLBs can be configured by setting bits in a configuration random access memory (RAM), hence the configuration of any CLB can be changed under the control of a host PC. This memory is volatile, so the FPGA must be programmed when it is first powered up and may be reprogrammed at any time during operation.

3 Java Virtual Machine advantages with FPGAs

A JVM is a complex machine with many security features, and its complex nature makes mapping it to a conventional CPU a hard task. A FPGA is a reconfigurable device such that you can build your own electronic circuit on it with great ease and also change it after it has been built. This makes a FPGA ideal to test and re-test multiple designs. Because of its flexible nature it is possible to build a very efficient JVM on a FPGA.

Ha et al. (Ha et al. 2001) gain an advantage by executing expensive Java instructions in hardware, on the FPGA, to speed up Internet applications that use the JVM. This way the advantage of having a platform independent language is still achieved, and since the instructions are executed in hardware the speed of execution is increased. Schoeberl (Schoeberl 2003) takes advantage of a FPGA in his system where he makes use of the multiple ports of the block RAM devices on a FPGA to compress four instructions into one, thus increasing the efficiency of the machine fourfold. In addition to this he also optimises the instructions that are to be executed, further increasing the speed.

Implementing a JVM on a FPGA has advantages such as being able to build parallel circuits. Kreuzinger et al. (Kreuzinger et al. 2003) take advantage of the parallelism of a FPGA when building their JVM, in that they have designed a system capable of running multiple threads in parallel, but with the limitation that these threads are assigned to specific tasks and are left idle when that task is not active, for example a task could be to accept input from a microphone. Radhakrishnan et al. (Radhakrishnan et al. 2001) take on an approach that is similar to a Just-In-Time (JIT) compiler (Cramer et al. 1997), but takes advantage of using a FPGA to do the translation of Java byte-codes into native code instead of the translation being done in software. They achieve better performance overall, than a regular software JIT compiler.

The fact that the JVM executes in hardware instead of software along with the flexibility of a FPGA device allows the JVM to do more work in fewer clock cycles. These advantages add strength to the argument that one should build a JVM on a FPGA, but along with these advantages come certain limitations that also must be looked at and will be discussed in the next section.

4 Limitations of using a FPGA for building JVMs

FPGAs are in their element when flexibility of design is a major issue, which is the case when developing a complex system. A system is never perfect once built, it always needs to be refined — this makes FPGAs excellent development tools. Yet, FPGAs also have limitations, as any hardware device has, due to the predefined space and limited memory, although the latest FPGAs are overcoming this with increased space and memory. A JVM is a very complex abstract machine, and because of this it is extremely hard to fit a fully-fledged JVM on a FPGA. To overcome this problem, previous researchers have limited the amount of the JVM specification (Lindholm & Yellin 1999) they have implemented.

Schoeberl (Schoeberl 2003) and Kreuzinger et al. (Kreuzinger et al. 2003) only implement certain subsets of the JVM specification, but even with this limitation the devices are well suited to certain applications. For example, Schoeberl (Schoeberl 2003) has used his JOP system for a commercial project where it is used to control up to 15 asynchronous motors for loading and unloading goods wagons. Schoeberl (Schoeberl 2003) overcomes the space limitation of a FPGA by having a supporting host machine that the FPGA communicates with and that does certain preprocessing tasks to take away some of the higher level activities that are required by a JVM. Alliance Core (Digital Communication Technologies n.d.) in their

CPU boast of an optimised circuit that only takes up to 3% of a top-of-the-line FPGA. Although building a JVM for a FPGA has certain limitations due to the nature of the device, there are also many applications that they can be used for. These applications will be discussed in the next section.

5 Applications of JVMs on FPGAs

We are seeing a rapid increase in the number of portable electronic devices available, such as mobile phones and MP3 players. These electronic devices have circuitry that is designed to perform tasks such as producing sound, manipulating sound, transferring data, and performing algorithms. Because portable devices run on batteries, one of the main design issues is building a circuit that consumes the least amount of power in order to gain the most running time on a device. FPGAs potentially excel in this low power consumption field due to their ability to be designed in such a way that they perform multiple tasks concurrently, and thus use less power.

Schoeberl (Schoeberl 2003) takes advantage of this when designing his JVM in that he optimises instructions that would usually require multiple clock cycles to one clock cycle: for example, by using the multiple ports on the block RAM to do an addition instruction in one clock cycle instead of four. This aids the goal of using less power due to the relationship between clock cycles and power usage.

Alliance Core (Digital Communication Technologies n.d.) say that their device is aimed at the following applications: Internet appliances, multimedia controllers, automotive devices, network/switch processors, set-top boxes, networked embedded controllers, terminals, PDAs, TVs, printers, copiers, and smart cards. Ha et al. (Ha et al. 2001) have aimed their system at Java applications that are web based, taking the place of a regular software JVM in order to increase the speed.

A software JVM is relatively slow compared with executing native code directly on a general purpose CPU. JIT compilers try to overcome this by using software to convert the Java Byte Codes into native code just before execution. Radhakrishnan et al. (Radhakrishnan et al. 2001) use a FPGA to perform this task. Because of the conversion being done in hardware compared to software there is a major increase in speed. This is an example of how a FPGA can be used in a conventional computer to aid in increasing the speed of a JVM.

The next section argues that a more abstract approach to building a JVM should be used.

6 Aim

As previously discussed, FPGAs have been used to build JVMs and explore the advantages of building a JVM in hardware. An approach of using multiple read and write ports on the FPGA block RAM and optimising the Java byte codes has been explored by several researchers (Schoeberl 2003) (Digital Communication Technologies n.d.) (Radhakrishnan et al. 2001). Ha et al. (Ha et al. 2001) have investigated the use of FPGAs for accelerating Java web applications. FPGAs have a great deal of flexibility when it comes to creating circuits in parallel. Kreuzinger et al. (Kreuzinger et al. 2003) use this flexibility in designing a four threaded JVM aimed specifically at real-time systems. This paper is concerned with constructing a multi-threaded JVM similar to Kreuzinger et al. (Kreuzinger et al. 2003), but instead of having threads dedicated to a particular task and aiming the device for real-time systems, it is approached on a more abstract level by both increasing the number of threads and having the threads identical in structure to each other and not specialised for specific tasks. The next section describes a method for building such a system.

7 Method

A FPGA is programmed to form an electronic circuit. This circuit can be designed manually using hardware description languages such as VHDL (IEEE 1994). Hardware description languages allow a system to be described as a composition of gates — a circuit design exercise. In this work we adopted a more unusual approach by designing the JVM in the imperative programming language Handel-C (Celoxica n.d.). Handel-C is in the style of ANSI-C, but also contains elements of the earlier language Occam (Jones 1985) and Hoare's Communicating Sequential Processes (CSP), from which Occam was derived (Hoare 1985). Handel-C allows systems to be designed using the familiar imperative programming paradigm. It takes the code you have written and converts it to a design of an electronic circuit that can be transferred to a FPGA.

Handel-C is well suited for designing a JVM. A JVM has to be able to support execution of instructions that are defined in the JVM specification. Each instructions has a unique op-code associated with it. For example, the JVM can execute the instruction `iadd`, which corresponds to op-code 96. A way to handle these instructions is to build a switch statement that handles these op-codes and executes the correct instruction. A loop must also be used to go through the instructions to be executed, as shown in Figure 2.

```
void CPU(unsigned 8 index) {
  ...
  while(moreInstructions) {
    switch (instruction) {
      case 96:
        iadd();
        break;
      ...
    }
    ...
  }
}
```

Figure 2: Overview of the CPU top-level function.

The code written is very similar to that of a simple software JVM. There is no point in using more sophisticated techniques such as employed by a JIT compiler. This is because the Java instructions are not being mapped to native code for a hardware device, instead they are executed directly.

A Java program compiles to a list of instructions that are to be executed on a JVM. These instructions are commonly called Java byte-codes. The JVM specification (Lindholm & Yellin 1999) contains a description of all 204 instructions. Each instruction can be represented by 1 byte and is followed by 0 or more bytes that are arguments to the byte-code. The number of bytes that follow the instruction are (naturally) dependent upon the instruction. This is categorically specified in the JVM specification for each byte-code.

In deciding what byte-codes to implement for the JVM, a few Java programs containing thread usage, loops, arrays, and integer arithmetic were created. These programs where compiled and "jclasslib" (ej-technologies n.d.), a Java class file inspector, was used

to find out the byte-codes these programs used. An implementation was created for 65 out of the 204 instructions. The 65 instructions are listed in Table 2 which contains their name and corresponding op code. These instructions include integer operations, loop instructions, conditional statements, and array operations. For further information on these instructions please refer to the description provided in the JVM specification (Lindholm & Yellin 1999). To give an idea of the comparison of a system that can execute certain statements in parallel to one that cannot, Table 1 has been provided. This table lists some of the implemented instructions along with a brief description and also the number of clock cycles they take on the JVM on the FPGA. It also includes the minimum number of clock cycles the same instruction would take on a machine only capable of executing statements sequentially. The table may contain entries with more than one instruction in cases where the instructions are similar. The following list describes the fields used in Table 1:

- Op Code: The integer used to identify the instruction;
- Name: The name of the instruction;
- Description: A description of the instruction;
- CC: The number of clock cycles required by a FPGA-based JVM to perform the instruction;
- SCC: The minimum number of clock cycles the same instruction would take on a sequential JVM.

The most common procedures that the instructions share are the ones that manipulate the stack. Two things are needed here.

1. Somewhere to store the stack and
2. Functions to push and pop the stack.

Because the stack is frequently used it makes sense to place it somewhere that can be accessed quickly. A FPGA has internal storage devices called Block RAMs. Each Block RAM has multiple Read/Write ports and is placed on the chip directly near to where the circuit is built. This allows multiple accesses to the stack per clock cycle. Handel-C provides us with syntax to declare and access Block RAM as a variable much as any other array in C. Here is a definition of a stack that uses the Block RAM:

```
ram signed 32 stack[50] with {block = 1};
```

There are also ways to declare variables that use Block RAMs so that the programmer can make use of the multiple ports it has. Now that we have made space for the stack on the Block RAMs, we can now write functions to push and pop the stack.

A variable named stack pointer is used to keep track of the top of the stack. Pushing an item on the top of the stack involves placing the item in the array at the element with an index of the current stack pointer, and then incrementing the stack pointer by 1. Popping an item off the stack involves decrementing the stack pointer, and then returning the item in the array at the element with an index of the stack pointer.

Since Java instructions frequently use push and pop it is important to have these functions as efficient as possible. Some Java instructions, such as `iadd` pop two items off the stack. As previously mentioned, the Block RAM has two ports. We can use these ports in parallel to read/write two values in one clock cycle with the use of the Handel-C function `par`. Figure 3 shows an implementation of pop, Figure 4 shows an implementation of push, and Figure 5 shows an implementation of pop that can be used to concurrently pop two items off the stack as required by instructions like `iadd`. All these functions take a single clock cycle to complete. It is worth noting that within the par block when a variable is assigned a new value, this value is only valid after the par block. It is the previous value of the variable that is used as an r-value by other statements within the par block. For example, if before calling pop the value of the variable `sp` is 2, then it is this value that is used within the par block. It is only after the par block that the variable sp will now be read with a value of 1.

```
pop(signed 32 *item) {
  par {
    (*item) = stack[sp-1];
    sp--;
  }
}
```

Figure 3: Implementation for pop.

```
push(signed 32 item) {
  par {
    stack[sp] = item;
    sp++;
  }
}
```

Figure 4: Implementation for push.

```
poptwo(signed 32 *item1, signed 32 *item2) {
  par {
    (*item1) = stack[sp-1];
    (*item2) = stack[sp-2];
    sp = sp-2;
  }
}
```

Figure 5: Implementation for dual concurrent pop. For reasons of clarity this is a simplified form of the actual code.

The JVM have several arithmetic instructions that pop two items off the stack, and push an item back on the stack. These instructions include integer addition, integer subtraction, integer multiplication, and integer division. Their corresponding instruction names and decimal representation of their byte-codes are iadd (96), isub (100), imul (104), and idiv (108). These instructions require us to perform an operation on the top two items of the stack and place the result back on the stack. This requires two reads and one write to the Block RAM. This means we require a minimum of two clock-cycles because we can only perform two read/writes to the Block RAM in parallel. Figure 6 shows the implementation of the instruction `iadd` that takes a total of two clock-cycles.

We have now defined ways to handle how to interpret what instruction is to be executed and how to execute this instruction. This idea can be taken further with the aid of Handel-C to build a JVM that is capable of running instructions from different threads in parallel. A JVM that does not have the ability to run instructions in parallel would handle threads in a sequential manner. It gives a certain amount of processing time to one thread and then uses context switching (Tanenbaum 2001) to give another thread

Op Code	Name	Description	CC	SCC
2...8	iconst_m1, iconst_n. where n = 0...5	Pushes constants -1...5 on the stack.	1	5
96, 100	iadd, isub	Pops two integers off the top of the stack, performs addition/ subtraction on them and pushes the result on the stack .	2	7
42...45 26...29	aload_n iload_n where n = 0...3	Push the reference/ integer stored in local variable n on the stack.	1	3
75...78 59...62	astore_n istore_n where n = 0...3	Pops the top item off the stack and place it in local variable n.	1	3
25, 21 58, 54	aload, iload astore, istore	Similar to previous, but takes a 8-bit argument as to what local variable index is to be used.	1	3

Table 1: Some common Java instructions and the minimum number of clock cycles they require to execute on a FPGA-based JVM (CC) and a sequential JVM (SCC).

Code	Name	Code	Name	Code	Name	Code	Name
2	iconst_m1	43	aload_1	89	dup	163	if_icmpgt
3	iconst_0	44	aload_2	96	iadd	164	if_icmple
4	iconst_1	45	aload_3	100	isub	167	goto
5	iconst_2	46	iaload	104	imul	172	ireturn
6	iconst_3	54	istore	108	idiv	177	return
7	iconst_4	58	astore	116	ineg	178	getstatic
8	iconst_5	59	istore_0	132	iinc	179	putstatic
16	bipush	60	istore_1	153	ifeq	182	invokevirtual
17	sipush	61	istore_2	154	ifne	183	invokespecial
18	ldc	62	istore_3	155	iflt	184	invokestatic
21	iload	75	astore_0	156	ifge	187	new
25	aload	76	astore_1	157	ifgt	188	newarray
26	iload_0	77	astore_2	158	ifle	194	monitorenter
27	iload_1	78	astore_3	159	if_icmpeq	195	monitorexit
28	iload_2	79	iastore	160	if_icmpne		
29	iload_3	87	pop	161	if_icmplt		
42	aload_0	88	pop2	162	if_icmpge		

Table 2: The subset of Java instructions implemented in this project.

```
iadd() {
  signed 32 temp1, temp2;
  poptwo(&temp1, &temp2);
  push(temp1+temp2);
}
```

Figure 6: Implementation for iadd.

some processing time. Having a JVM that is capable of running instructions in parallel lets multiple threads execute their instructions at the same time.

The novelty of this system is in the way it achieves its parallelism. A function was created with a name of `CPU`. This function is capable of executing Java instructions. It also contains the storage devices used for running the virtual machine, such as the stacks and local variables. A `CPU` has entities that are its own, and access to shared entities. For example, a `CPU` has its own storage location for Operand Stacks, Local Variables, Java instructions, etc... A CPU shares some resources such as the Constant Pool. In the context of only one `CPU` function existing, having shared as well as individual entities does not make much sense. It is when duplicating the `CPU` function, in order to achieve parallelism, when benefits are achieved. With the aid of the Handel-C function `par`, we can create multiple copies of `CPU` and have them run in parallel. With Handel-C it is possible for the programmer to select what to duplicate and what to share by a simple keyword `inline`. If a function is declared to be `inline`, then its corresponding circuitry is duplicated and not shared. This is illustrated in Figure 7.

```
inline test() {
  ...
}
test2() {
  ...
}
CPU() {
  test();
  test2();
}
main() {
  par {
    CPU();
    CPU();
  }
}
```

Figure 7: Illustrating the use of inline functions.

In this example the circuitry that corresponds to the `CPU` function is duplicated and executed in parallel because the `CPU` function is called twice in the `par` block. The function `test` has its circuitry duplicated because it is declared to be `inline`. This means each `CPU` function has its own copy of the function `test`, but the function `test2` has its circuitry shared by the two `CPU` functions because it is not declared to be `inline`. In this manner it is possible to create methods where each `CPU` has its own copy, or where there is one shared copy among the `CPU` functions.

Sharing is implemented for functions that produce a very complex circuit and do not get used very often. Some memory areas are also shared between threads for global access. Semaphores (Tanenbaum 2001) are used to control access to shared resources. The next section describes some results from a JVM built using this method.

8 Results

Building the parallel JVM on a FPGA is still a work-in-progress, but so far it can run multiple threads in parallel and run basic Java numeric programs. It has support for arrays that can be declared for private use for a particular thread, or can be shared among the threads. There is also support for Java semaphores which can be used to control access to shared memory. Each CPU has its own stack, local variables, and instruction executor. The CPUs run concurrently, since each thread has its own piece of circuitry to execute on.

8.1 Increasing the number of CPUs

As discussed earlier, the CPU structure of the JVM on a FPGA is responsible for executing a thread. The number of CPUs is the same as the number of threads that are able to be executed in parallel. So by increasing the number of CPUs, we increase the number of threads able to be executed in parallel. It is important to find out the benefits gained from increasing the number of CPUs and also the shortcomings.

The more CPUs we have, the more space we are using on the FPGA. This means we have more circuitry to power. With current technology, this is not much of a drawback due to the minimal amount of power that this requires (Ghosh, Devadas, Keutzer & White 1992). Most power is used when a clock cycle occurs, and thus by increasing the clock speed we also increase the power consumption. This has been the trend for general purpose processors, which increase their clock speed to increase the speed of execution. By executing multiple instructions in parallel we can achieve more work per clock cycle, thus gaining a power consumption to work output advantage.

To demonstrate the decrease of clock cycles for the same amount of work by increasing the number of CPUs a test program was constructed. It calculates Euler's Phi (Anderson & Bell 1997) of a range of numbers in a "for" loop. Euler's Phi is used in Number Theory and calculates the number of positive integers less than a certain number which are relatively prime to that number. The test program was executed on a JVM with 1 CPU, 2 CPUs, and 3 CPUs. The number of clock cycles this test program took to run was counted and the results are shown in Table 3.

CPUs	Clock Cycles	Relative	Overhead
1	33657822	100%	0%
2	17507261	52%	2%
3	11311064	33.6%	0.3%

Table 3: Speedup calculating Euler's Phi resulting from additional CPUs.

This speedup occurs because the work load is distributed across the CPUs. Each CPU executes a subset of the "for" loop. As can be seen from the results, it is not a linear relationship. One of the reasons for this is the uneven distribution of the work load. In this example, the statement within the loop (Euler's Phi) takes a different amount of time for each iteration — the larger the number the longer the statement takes. The other reason is the time it takes to copy the Java instructions to the local cache of each CPU.

Using 2 CPUs requires less clock cycles than 1 CPU to achieve the same amount of work, and 3 CPUs requires less than 2 CPUs. This means we achieve the same amount of work for a less amount of energy. Energy (E) is defined by $E = P * T$, where P is for power and T is for time. The power used by

the JVM on the FPGA with 1 CPU, 2 CPUs, and 3 CPUs is approximately the same, but the time taken decreases as we add more CPUs which is responsible for the saving of energy. With a top-of-the-range FPGA it would be possible to run this experiment on up to 19 CPUs.

The performance gained by increasing the number of CPUs depends on the program being executed. For example, if multiple threads access a shared resource at the same time then there may be some waiting for the resource. The benefit is achieved when multiple CPUs are doing work at the same time, the more work they do at the same time the better. The next subsection details an experiment that was used to compare a JVM on a Pentium machine to the JVM on the FPGA. Due to the simplistic nature of the test program it was possible to fit 6 CPUs on the FPGA.

8.2 Multi-threading on JVMs

A multi-threaded Java program was created and executed on the JVM on the FPGA and a Pentium machine running Microsoft Windows. The Pentium had a clock speed of 133 MHz and it was using Java version 1.4. This version of Java uses a procedure called "Hotspot" that involves compiling the most frequently used parts of the Java instructions into native code for quicker execution. The test program was executed on the Pentium with "Hotspot" enabled and it was also executed with "Hotspot" disabled which then uses the standard interpretation method. The standard interpretation method is a more accurate comparison with the JVM on the FPGA because it more accurately follows the same method to execute a Java program.

The Pentium with a clock speed of 133 MHz was chosen for a few reasons. One of these reasons is that the clock speed is similar to that of a portable device, and also because with the FPGA used in this project it was only possible to clock it at a rate of 50 MHz. The latest FPGAs are able to be clocked at speeds in excess of 500 MHz.

The test program was executed with 1, 2, 3, 4, 5, and 6 threads. For the Pentium machine the time taken was calculated by using the Java method `System.currentTimeMillis()`. It was taken 5 times and the average was calculated. On the FPGA a counter that counted the number of clock cycles was implemented and the number of clock cycles the FPGA took was divided by the clock speed of the Pentium machine to compare the results (and hence the approximate amount of energy used). In other words, the results were compared as if the FPGA was clocked at the same speed as the Pentium.

As mentioned in the previous subsection, energy is the power rate times the amount of time, and also that the clock speed is the largest factor that influences the power rate, so by normalising the results this way we allow for a more realistic comparison of the amount of energy used. The following equation depicts this relationship in which `S` is the clock speed of the Pentium, `NC` is the number of clock cycles taken on the FPGA, and `T` represents the amount of time taken. $T = NC/S$. Table 4 shows the results of the experiment in a tabular format. Figure 8 displays a graph of the results where the time taken to complete the task has been divided by the number of threads. This division of time gives the time taken to complete one unit of work.

As can been seen from Table 4 executing more threads requires more time for the Pentium, but for the FPGA it is done in roughly the same time. This is because the FPGA is able to execute the multiple threads in parallel whereas the Pentium computer cannot. It is seen by the graph in Figure 8 that the time taken to complete one unit of work varies on the Pentium as the number of threads increase. It was predicted that the time taken would increase due to the extra cost associated with context switching, but this is not what was recorded. For 1 and 2 threads the time taken per unit of work stays constant, but decreased for 3 threads, which then increases for 4, 5, and 6 threads. This could be due to the fact that with this experiment and the fast speed of the processor that context switching is not as much of a high cost as expected and that some other factors are influencing the results, which is hard to calculate because of limited proprietary knowledge of the implementation of the Pentium technology and Microsoft Windows. It is also interesting to note that the Pentium follows this pattern for the JVM with "Hotspot" and without "Hotspot". As expected, the time taken per unit on the FPGA decreases as the number of threads increase. One might like to note that the number of clock cycles taken on the FPGA varies slightly which is mainly due to the copying of instructions into the local cache of each CPU and semaphore access to certain methods such as the method used to output results.

8.3 JVM on FPGA resource usage

To gain an estimate as to how many JVM CPUs can fit on a FPGA some results were calculated. The JVM that was tested was one which contained support for semaphores, global variables, integer instructions, and loop constructs. The Xilinx ISE tools were used to calculate statistics on JVMs that were compiled with between 1 and 10 CPUs. The statistics recorded contain information on how much space is needed to add an extra CPU to the system. It is important to note that this is not a fully-fledged JVM, but has a limited instruction set implemented as described in the method section. The metrics that were recorded include the number of slices, number of slice flip-flops, number of look up tables (LUTs), and an equivalent gate count. They will now be discussed.

As mentioned previously, a FPGA is made up of thousands of CLBs. Each CLB on the FPGA used in this project, the Xilinx XCV1000, contains 2 slices; a slice contains a flip-flop, also known as a latch, and a LUT. Latches are used to store state, and in the context of the JVM they are used to store temporary variables. LUTs are used to implement the circuitry that is to be created, such as arithmetic functions.

A CPU requires a certain number of latches and LUTs, so it makes sense that as we increase the number of CPUs the JVM has, the number of latches and LUTs grow linearly. There is also a number of latches and LUTs that are required for other elements of the JVM such as the initialisation subsystem. This is represented by the initial value of latches and LUTs and can be calculated by working out the difference between 2 and 1 CPUs and then taking this difference away from the cost of 1 CPU. For example, 2 CPUs require 4821 latches and 1 CPU requires 3052. This means it cost $4821 - 3052 = 1769$ latches to add an extra CPU. If we take this away from the number of latches for 1 CPU, $3052 - 1769 = 1283$, we now have the number of latches required for shared JVM subsystems such as the initialisation of the constant pool which is a system task and not a CPU task. It is important to note that this result is not exact, but an estimate. It can vary by about 100 latches in either direction due to logic needed to control sharing.

The graph in Figure 9 displays 6 different results. The first in the legend is the "Number of Slices". This is the number of slices that are being used on the FPGA. The next is the number of slices that contains unrelated logic. To explain this a bit of back-

N	FPGA Clock Cycles	FPGA (Normalised)	Pentium Hotspot	Pentium No-Hotspot
1	432203373	3.25 secs	0.93 secs	7.85 secs
2	432213563	3.25 secs	1.86 secs	15.75 secs
3	432212827	3.25 secs	2.74 secs	22.87 secs
4	432210538	3.25 secs	3.67 secs	31.25 secs
5	432253516	3.25 secs	4.53 secs	38.98 secs
6	432216609	3.25 secs	5.42 secs	46.87 secs

Table 4: Execution time required to execute N units of work in N threads. Execution times have been scaled so that all results are at an effective clock speed of 133MHz.

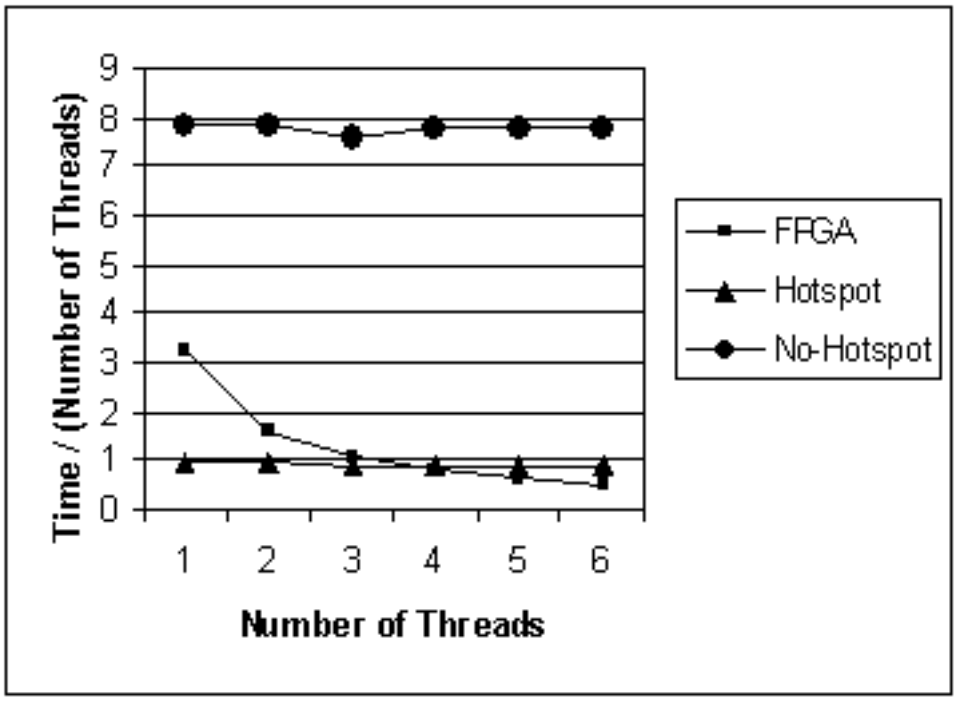

Figure 8: A graphical representation of the results in Table 4 but normalised by N. This can therefore be interpreted as the amount of time taken to complete one unit of work.

ground knowledge is firstly required. The Xilinx ISE tool requires the details of the FPGA it is compiling for, which allows the tool to configure certain parameters. The FPGA used in this project, the Xilinx XCV1000, was entered in the Xilinx ISE tool. This FPGA contains a total of 12288 slices. The Xilinx ISE tool creates an optimised circuitry if it can nicely fit the design within these slices. If this is successfully the Xilinx ISE tool does not need to create any unrelated logic just to make the design fit. Although, if a design cannot be optimally fit within the space, then the Xilinx ISE tool tries to fit it in another way which requires creating unrelated logic. This unrelated logic is used to "squeeze" the design within the FPGA. As can be seen from the graph, the unrelated logic line only starts to grow when the limit of 12288 slices is reached. If a larger FPGA was used, then unrelated logic would start getting created at its total number of slices. In essence, these first two lines only pertain importance to the individual FPGA used in this project, and the following lines hold more important information. The next line in the legend is the number of slice flip-flops, which is the number of latches used. Following this is the total number of LUTs which consists of LUTs that are used to create circuitry specific to the JVM and LUTs unrelated to the JVM, but used for connection between different elements.

By inspection of the graph in Figure 9 we can see that a CPU requires more LUTs than latches per CPU. We detect this by seeing that the gradient for LUTs is steeper than the gradient for latches. We can also see that the initial number of latches and LUTs is very similar which means that the shared circuitry uses roughly the same number of latches and LUTs. As mentioned previously, each slice contains a latch and a LUT, so if we want to make complete use of the FPGA then we should try to have the growth of latches and LUTs similar. In relation to the JVM this means that we should make more use of temporary variables as to increase the number of latches used per CPU, and/or decrease the amount of circuitry required per CPU. This is because we have an equal number of latches and LUTs available to us and we want to run out of them at the same time to minimise waste.

To gain an estimate of how many CPUs can fit on a top-of-the-range FPGA we can use the number of LUTs as a guide. This is because we will run out of LUTs first because their number increases faster than that of latches. A leading FPGA offered by Xilinx is the Virtex II XC2V8000. This FPGA contains 46,592 slices which means we can have up to 46,592 LUTs. Table 5 displays the recorded results of how many LUTs are required for different numbers of CPUs. The "Difference" column is the number of LUTs required to add the extra CPU and is calculated by taking away the previous number of LUTs. Using this table, the average number of LUTs required to add a CPU is 2921.222. So we now can estimate that $46592/2921.222 = 15.95$ CPUs can fit on the XC2V8000 FPGA.

Most of the integer and loop instructions in the Java specification have been implemented. Instructions are frequently accessed by the CPU, so to increase performance each CPU has a local instruction cache. Block RAM is used for this cache because of its speed of access and multiple ports. By the end of this project it is expected that the JVM will be more efficient by adding in sharing of certain functions between the CPUs and also by optimising certain functions that are used frequently. This will increase the speed of execution and also decrease the number of CLBs used.

CPUs	LUTs Used	Difference
1	3889	-
2	6953	3064
3	9947	2994
4	12876	2929
5	15617	2741
6	18580	2963
7	21483	2903
8	24255	2772
9	27237	2982
10	30180	2943

Table 5: Number of LUTs required within the FPGA to implement differing numbers of independent CPUs.

9 Conclusion

Java has an advantage of being platform independent, but because of the conventional software JVM, it has a slower execution speed than a program that does not have to be interpreted before being executed on a conventional CPU. Building a JVM in hardware overcomes the speed problem, but keeps the advantage of having a platform independent language. Adding in a feature so that the system is able to execute multiple instructions in parallel gives a speed advantage to a multi-threaded program and also produces more work per power unit, and as such is ideal for embedded systems.

Handel-C can be used to build such a system with relative ease due to its easy syntax. It takes care of the complex task of converting the code into an electronic circuit that can be transferred to a FPGA. It also provides a debugging environment that can be used to debug a system, which is a lot easier than debugging the hardware directly.

10 Acknowledgements

We would like to thank Xilinx Inc. and Celoxica Ltd. for the donation of devices and software in aid of this project.

References

Anderson, J. A. & Bell, J. M. (1997), *Number Theory with Applications*, Prentice Hall.

Celoxica (n.d.), 'Handel-C overview'. (Retrieved May 8, 2003 from the World Wide Web: http://www.celoxica.com/tech/handel-c/default.asp).

Cramer, T., Friedman, R., Miller, T., Seherger, D., Wilson, R. & Wolczko, M. (1997), 'Compiling Java just in time: Using runtime compilation to improve Java program performance', *IEEE Micro* **17**(3), 36–43.

Digital Communication Technologies (n.d.), 'Xilinx alliance core'. (Retrieved May 10, 2003 from the World Wide Web: http://www.xilinx.com/products/logicore/alliance/digital_comm_tech/dct_lightfoot_32bit_processor.pdf).

ej-technologies (n.d.), 'jclasslib'. (Retrieved September 22, 2003 from the World Wide Web: http://www.ej-technologies.com/products/jclasslib/overview.html).

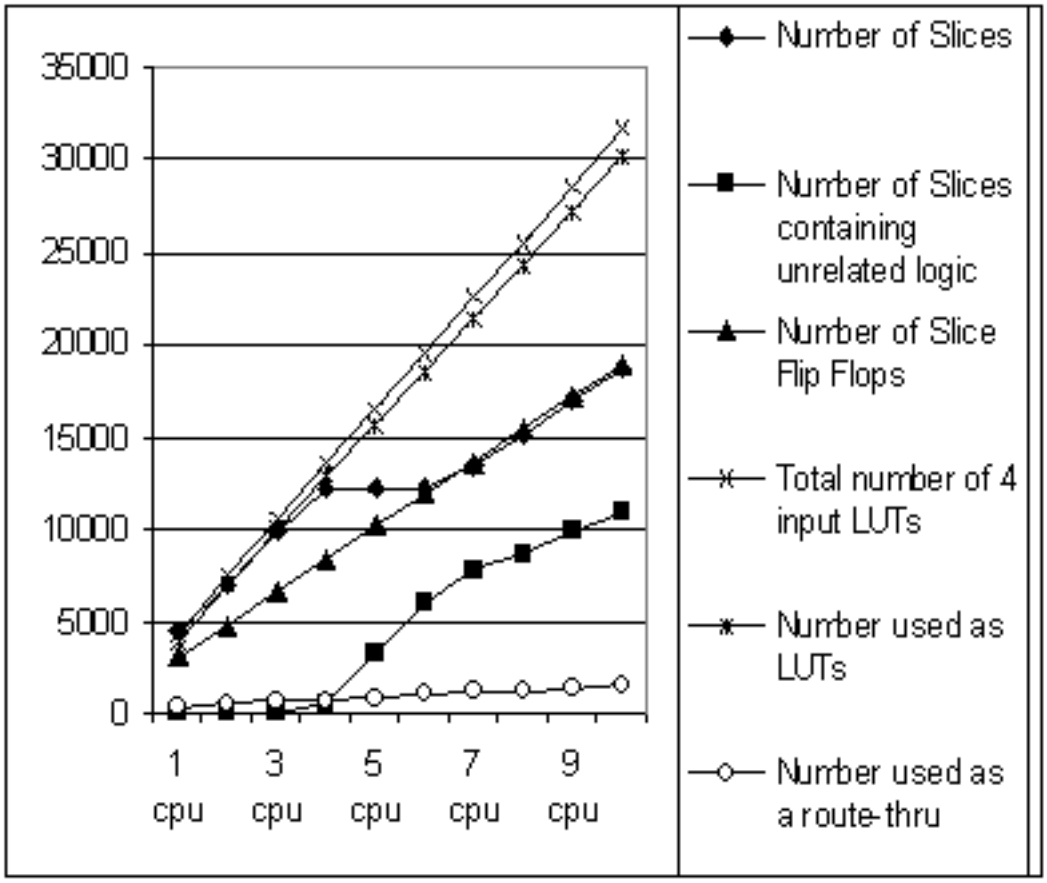

Figure 9: More detailed graph of resource usage within the FPGA with increasing numbers of CPUs. The categories are, as reported by Xilinx's ISE synthesis tools.

Ghosh, A., Devadas, S., Keutzer, K. & White, J. (1992), Estimation of average switching activity in combinational and sequential circuits, *in* 'Design Automation Conference', IEEE Computer Society Press, pp. 253–259.

Ha, Y., Vanmeerbeeck, G., Schaumont, P., Vernalde, S., Engels, M., Lauwereins, R. & De Man, H. (2001), Virtual Java/FPGA interface for networked reconfiguration, *in* 'Proceedings of the conference on Asia South Pacific Design Automation Conference', IEEE, pp. 427–439.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ. ISBN 0-13-153289-8.

IEEE (1994), *IEEE Standard VHDL Language Reference Manual: ANSI/IEEE Standard 1076-1993*, IEEE Press.

Jones, G. (1985), Programming in 'occam', Technical Report Technical Monograph PRG-43, Oxford University Computing Laboratory Programming Research, Group.

Kreuzinger, J., Brinkschulte, U., Pfeffer, M., Uhrig, S. & Ungerer, T. (2003), 'Real-time event-handling and scheduling on a multithreaded Java microcontroller', *Microprocessors and Microsystems* **27**(1), 19–31.

Lindholm, T. & Yellin, F. (1999), *The Java Virtual Machine Specification*, Addison-Wesley.

Radhakrishnan, R., Bhargava, R. & John, L. K. (2001), Improving Java performance using hardware translation, *in* 'Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)', ACM Press, New York, pp. 427–439.

Schoeberl, M. (2003), Using JOP at an early design stage in a real world application, *in* 'Proceedings of the workshop on Intelligent Solutions in Embedded Systems'. *http://www.vmars.tuwien.ac.at/wises/

Tanenbaum, A. S. (2001), *Modern Operating Systems, 2nd edition*, Prentice Hall, Upper Saddle River, N.J.