# HARDWARE SUPPORT FOR EMBEDDED JAVA

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

January 2007

By
Paul Capewell
School of Computer Science

# Contents

# List of Tables

# List of Figures

# Abstract

Java is a modern, general purpose object orientated programming language originally designed for embedded systems. Java first saw wide spread adoption in the area of web-based distributed applications because of its portable binaries, security features and convenient programming interfaces. These features also apply to modern mass market embedded and mobile systems. An increase in the power of such devices has led to wide spread adoption of Java in this domain.

This thesis investigates the issues which arise when attempting to execute portable Java binaries on embedded processor architectures. The central theme is acceleration of Java binary translation through the extension of embedded processor pipelines. This is an established method of efficiently reducing power consumption, memory requirements and system cost while increasing Java execution speed.

Existing techniques for hardware assisted binary translation are investigated, and novel approaches to the problem are suggested and evaluated. Current commercial products in this space use simple translation techniques, this ensures low power requirements and system cost. More elaborate translation mechanisms are proposed that can increase execution rate while aiming to have low implementation costs.

Simulations of a simple asynchronous Java decoder design at silicon level calibrate and contrast to the results of higher level simulations of the new translation mechanisms employing asynchronous pipelines. Results show performance increases when employing more advanced translation mechanisms and provide quantified trade-off options which can aid designers of such systems in the future.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

# Copyright

# Acknowledgements

I would like to thank Amy, my family, the APT group and the staff at Silistix for their individual and invaluable support during the years spent working in Manchester.

# Chapter 1

# Introduction

## 1.1 The Java Language and Runtime Environment

The advent of the transistor meant that much larger, more reliable, systems became feasible and design methods advanced to meet the needs to handle greater complexity.

Java [45] is a general purpose object orientated programming language introduced by Sun Microsystems. One of the main aims of the language and runtime system was to allow for binary compatibility across multiple platforms. To facilitate this feature an intermediate binary format was designed. To allow for compatibility with Java binaries on any platform, a *Java virtual machine* (JVM) [32, 33] must be implemented. The JVM interprets Java binaries (Java class files) and issues appropriate instructions to the host processor(s). It is usual to code a JVM in software, however this is not mandatory, this choice is left to the implementer.

The JVM specifies an 8 bit instruction format capable of operating on various data-types from 8 bit integers through to double precision floating-point numbers. The JVM is stack based and hence has relatively high density binary files [35, 15]. High code density is a desirable quality in many of the markets which Java targets. Java has achieved widespread use as a language for the Internet mainly due to binary compatibility between different systems, but also due to its compact binary distribution format, resulting in reduced bandwidth requirements for program distribution. Interestingly, Java was originally designed for embedded systems where code density is paramount. Compact Java byte-code facilitates reduced storage requirements and therefore cheap, low power, feature rich systems. The advantages of binary compatibility also apply to embedded systems which will have an extended

life-time and reduced support cost through shared binary software updates. The compactness of Java binary files also comes into play when delivering *over-air* applications in mobile phones and other wireless connected devices which are becoming increasingly common.

Aside from the characteristics of the JVM, Java itself is also a very practical programming language with support for object orientated programming and large scale, manageable projects. It has a well specified set of application programming interfaces (API's) suitable for a large variety of target product areas, ranging from smart cards to distributed server environments. These qualities are another important factor in the widespread adoption of Java in embedded systems where, previously, use of C and assembler have often resulted in poor code re-use and very platform dependent, bespoke code. The main reason for this is that code must be optimised to minimise code density and to interface with the target platform. The advantage with such languages is that runtime performance targets may be easier to meet than with a virtual machine based system such as Java, which will usually run slower than native code.

A further important feature of Java is the inbuilt security of the language and runtime environment. Firstly, as programs run in a virtual machine environment, they never have direct access to the host machine, every instruction going through the JVM. Java has several levels of security and allows code to be run with different security levels. Untrusted binaries can be run in a sandbox environment, without access to restricted library calls. Even trusted code does not run in a totally unprotected environment like typical native code may, for example arrays are cleared on allocation, and pointers to memory do not exist as such. On small embedded systems without memory protection, achieving a reasonable level of security requires little effort when adopting a Java implementation.

The simplest form of Java virtual machine is one which takes each incoming Java byte-code, decodes its meaning and then issues appropriate native code (by calling a subroutine or something similar) to carry out the task specified by that byte-code. This is how the first generation of JVM's typically worked, and this can be made quite efficient, and can be optimised in many different ways, for example by the inclusion of SUN's `_quick` byte-codes [32] which can be substituted in the place of some slow to interpret byte-codes at run time to speed up repeated interpretation. Recent JVM implementations have taken a different approach, where blocks of Java byte-codes are compiled into native code blocks which are then called directly when

needed. This process can be time consuming but once complete this compiled code will execute very quickly, as JVM intervention is only needed as new parts of the program are reached. This approach is referred to as *just in time* compilation, or *JIT* compilation.

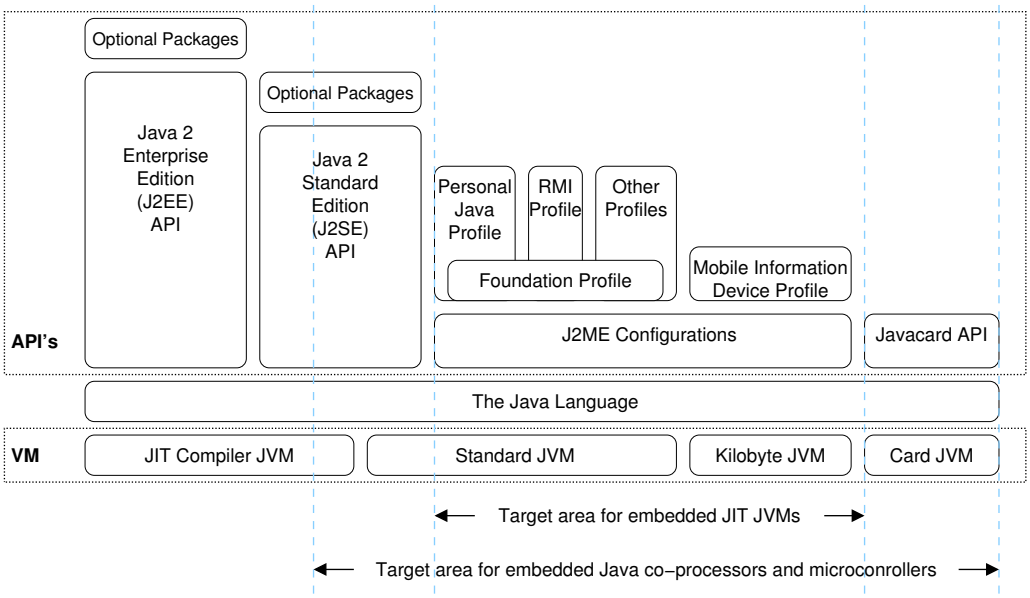


Figure 1.1: Java Product Spectrum.

Figure 1.1 shows a matrix of the different flavours of Java currently available in 2005. The top portion of the diagram shows the API sets available to programmers, each targeting different classes of systems. All of the API sets are shown to run on the same Java language, with the different virtual machine types listed at the bottom under the appropriate API. The API set is usually used to differentiate between the different releases of Java which Sun distributes, but it is important to note the suggested JVM as these are related to the type of system the API is targeted at:

**J2EE - Java 2 enterprise edition:** for servers and distributed software architectures. Systems running this environment usually have a server orientated JIT JVM which creates native code for all (or a large percentage of) the Java binaries it executes. This will aggressively optimise frequently used portions of the code speculatively as soon as possible, as it is anticipated that in a server environment the code will be run very repetitively and performance over a longer period of time is more important than startup times or memory usage.

**J2SE - Java 2 standard edition:** for workstations, PC's and Internet enabled devices. This configuration usually runs with a JIT compiler virtual machine. The current Sun implementation uses a technology called Hotspot$^{TM}$ which starts by interpreting byte-codes, then compiling basic blocks to native code and applying progressively sophisticated optimisation to frequently executed blocks. This results in fast startup times, important for interactive (usually web based) applications and also eventually good performance, after profiling and gradual optimisation have taken place. This method also has a reduced memory footprint over some other JIT strategies, as much rarely used code will never be compiled to native code and therefore not require further storage space. Simpler interpreting JVM's are also an option in this space, for reduced memory devices such as small Internet appliances or hand-held computers where J2SE functionality may be all that can be afforded within given memory limitations.

**J2ME - Java 2 micro edition:** for embedded devices. This is one of the newer Java editions, and is possibly becoming one of the broadest areas of Java use. The specification of J2ME allows for many different API sets to be specified to suit a particular product area. Standards exist for many types of devices ranging from TV set-top boxes to mobile phones. The JVM used can vary from a standard desktop JVM to smaller virtual machines suiting PDA's and mobile phones. This type of compact JVM is often called a kilobyte virtual machine (KVM), meaning that it requires a memory footprint measurable in kilobytes rather than megabytes when operational (normally around a few hundred kilobytes).

**Java Card:** for smart-cards. The JVM specification for this version of Java does not include floating point or long integer data-types, so is limited to integer arithmetic, allowing for a minute virtual machine, which will run on small system-on-chip smart-card processor cores with a small program ROM and even less RAM, but still facilitating the desirable benefits Java can offer.

## 1.2 Accelerating the Java Virtual Machine

Accelerating Java implementations is important, just as accelerating computers in general is important because people consistently expect more from their computer

systems in terms of performance and price per unit performance. In the case of Java in particular, increasing performance relative to native software is important in order for Java to compete and be a viable alternative to native programming languages, enabling people to exploit the advantages Java has to offer.

Many options are available for improvements over a simple JVM implementation. Ideally it would be nice to be able to increase performance of Java applications wholly in software, in all application areas, but this is not always possible. A JVM must always come at some cost, as it must emulate a machine architecture capable of running Java byte-codes, a native application does not have to emulate anything. So why not convert the whole Java program to native code? Firstly one key advantage of Java is the portable binary format, so any translation would have to be done from the binary byte-codes. After this realisation a fast virtual machine can be built, if a good binary translator can be designed. It is now necessary to store two sets of binaries at run time resulting in possibly excessive memory requirements. Fast software only JVM environments will always be fighting memory requirements, although this is only a big problem in the embedded application space, where memory is often very limited.

The only case where there is no apparent cost to the JVM is when the underlying machine architecture implements the JVM specification, or most of it. Taking away the virtual from the JVM is the essence of all hardware approaches to Java acceleration. However the costs here are in the implementation, it is expensive to design new processors. The JVM is also designed as a good virtual machine architecture with compact binaries, not as a good instruction set architecture to actually build hardware for. This approach also conflicts with the fact that there are already established processors in almost all Java application areas, which would be very difficult to compete against in terms of performance. A clever compromise is often needed in the hardware acceleration space.

## 1.3 Thesis Overview

### Chapter 1: Introduction

Chapter 1 contains a review of the Java language, application areas, virtual machine technologies and acceleration techniques. Rationale for the work carried out

is presented, justifying the need for improved efficiency Java virtual machine environments, particularly for embedded systems with small memory footprints. The introduction is concluded with a summary of each chapter in the thesis.

### Chapter 2: Improving Java Virtual Machines

Chapter 2 takes an in depth look at techniques employed for improving Java virtual machine performance. Firstly the chapter looks at existing software-only systems running on general purpose processor architectures, mainly aimed at the workstation market. Dedicated Java processor and acceleration co-processor hardware is then evaluated. Comparison of implementation strategies, decoding techniques, complexity and architectural partitioning is undertaken.

### Chapter 3: Architectural Techniques

Chapter 3 looks at implementation options for processor hardware. This chapter evaluates the benefits and drawbacks of using asynchronous design styles, different decoding strategies and design partitioning in the construction of processor architectures. This chapter focuses on developing an appropriate set of ideas for use in the design and implementation of an embedded system suitable for efficiently running Java code.

### Chapter 4: JASPA

Chapter 4 gives an overview of the first prototype Java enabled architecture designed to evaluate currently known translation techniques and asynchronous design approaches. The design was realised into a gate level netlist and simulated as a chip layout in 180 nanometre CMOS silicon. This chapter shows the feasibility of design, advantages of asynchronous decoding approaches as-well as problems with the implementation and naive decoding techniques.

### Chapter 5: Architectural Simulation

Chapter 5 describes and justifies the simulation system constructed to evaluate novel Java enabled processor architectures. The simulator architecture is described in detail showing how it allows for rapid simulation at the architectural level (pipeline

stage is lowest level entity). Asynchronous communication between units is supported along with separate easily configurable timing models for a given design. Implementation choices, system performance and features are evaluated.

## Chapter 6: Instruction Folding

Chapter 6 introduces instruction folding as a technique for enhancing the performance of Java translation suitable for rapid execution on a RISC processor core's ALU. Folding uses the registers available to cache local variables enabling operand stack operations to be skipped, allowing multiple Java byte-codes to execute in a single RISC processor cycle. Implementation strategies are explored and results from simulation of a folding Java processor architecture are presented. This chapter also describes the byte-code profiler implemented as a component of the simulated processor.

## Chapter 7: Improved Instruction Folding

Chapter 7 presents further techniques for instruction folding, allowing more efficient use of registers in order to reduce hardware costs while reducing the memory access overhead of hardware Java translation. Novel register allocation techniques are presented along with an improved stack cache methodology permitting instruction folding. Different configurations and numbers of processor registers are investigated to compare the efficiency of different folding techniques along with varying memory latency. The self-timed qualities of the folding decoder are examined here also.

## Chapter 8: Conclusion

Chapter 8 concludes the thesis, providing summary and details of all techniques investigated and introduced through this research. A discussion of problems, possible resolutions and future work inspired by this investigation of embedded architectures for Java closes this chapter.

# Chapter 2

# Improving Java Virtual Machines

This chapter details the varied approaches taken to achieve improved JVM performance introduced in both commercial products and research projects. The vast majority, if not all, commercial desktop and server virtual machines are implemented entirely in software. This is where most innovation has taken place in terms of gaining high performance, mainly at the expense of memory usage and start-up times. Making changes to general purpose hardware used in such workstation and server systems where Java is not the dominant software implementation environment would defy economics. Performance similar to pre-compiled code can already be achieved, typically plenty of memory is available to the JVM.

Embedded environments, such as mobile phones have provided a niche where Java is a standard, causing the establishment of many commercial hardware based Java solutions. This is still an area of great activity, with products available for 5 or more years continually evolving to suit the changing market place and performance increases in processors aimed mainly at high end mobile consumer devices.

The Java virtual machine is defined by a set of stack based instructions, byte-codes, operating on a small selection of data types. Data values can be stored in local and global variables as single values, arrays or in compound object structures. Access to all of these data storage containers is supported by dedicated byte-codes. Integer and floating point types have a large set of arithmetic and logic byte-codes all of which operate exclusively on an operand stack. Values must be loaded to this stack as constants or from local or global variable spaces before they can be processed. All instructions are 8 bits in length, hence the term byte-code. Some instructions have further immediate data fields, although many single byte instructions have been added for common values in order to improve code density.

## 2.1 Software Acceleration Approaches

The majority of Java code is currently run on software Java virtual machines such as those readily available from Sun, Microsoft and under GNU licences, all of which run on standard processors. Improving execution speed of Java in software only implementations is therefore an important area of research.

The main cost of a simple interpreting JVM is the time spent in the interpretation loop. Typically each Java byte-code will be fetched from memory in turn and decoded, an appropriate handler routine is then found and called, hence there is a decoding cost of several native machine instructions per Java instruction. This results in a speed penalty of perhaps 5 to 50 times slower than equivalent native code, dependent on the machine's architecture. In order to increase the performance of software JVM's the time spent decoding byte-codes relative to the time spent in handler routines must be decreased. The following Java implementations, which are currently available, attempt to address this issue using a combination of smart interpretation techniques, dynamic compilation and possibly architecture specific optimisations.

### 2.1.1 Sun 1.x JVM

The first JVM's made by Sun were purely interpreters and did not perform any just in time compilation by default. The main feature added to increase interpretation speed was a set of extra *quick* byte-codes. These byte-codes are not part of the JVM specification, but are substituted at interpretation time to replace byte-codes where constants must be looked up through the constant pool. A quick version of any byte-code has a direct reference to the constant, known only at run-time, saving the look-up process on future passes through processed code. There are 25 such quick byte-codes specified [32].

### 2.1.2 Hotspot™

Sun's Hotspot™ [47] technology combines an optimised interpreter and JIT compiler in an attempt to offer fast start-up times and ultimately very high sustained performance of Java code. Hotspot starts off by interpreting byte-codes, while performance profiling information is gathered. When it is determined which sections of code are being executed most frequently, these are gradually compiled, with a range

of increasing optimisations to achieve high performance but without large delays as code first starts running. Profiling and the majority of compilation is done at the method level, while other optimisation features can take place at boundaries of basic blocks. Incremental garbage collection is a feature added in the Hotspot JVM which spreads garbage collections over time resulting in reduced jitter in performance, which is also important in many applications.

### 2.1.3 Jikes

Jikes [2] is a virtual machine in development at IBM. Jikes (previously called Jalepeño) is a just in time Java compiler and unusually is written entirely in Java. The Jikes JVM is aimed at high performance server systems and sets out to exploit modern microprocessor architectures and SMP multiprocessing while providing server level reliability and performance characteristics. As Jikes is aimed at servers, it does not pay much attention to reducing memory usage. The virtual machine does not do any interpretation but instead relies upon a set of three inter-operable compilers, in this way execution performance can be traded off against compile time characteristics of each compiler, response time of new code segments can therefore be controlled.

### 2.1.4 Monty

Monty [48] is a relatively new virtual machine for Java, and is aimed squarely at the embedded systems market segment. Monty is made by Sun Microsystems and is a commercially licenseable product. Monty currently targets only the ARM series of microprocessors and is extensively optimised for this platform. The aim of project Monty is to provide an order of magnitude performance gain over their previous J2ME kilobyte virtual machine (KVM).

Sun realised that there is a new breed of embedded systems, such as smart phones and palm-top computers, with increasing amounts of memory, where over-air application deployment is important for software updates and distribution, Java is often the chosen platform to facilitate this. Previously the KVM virtual machine was being used, which is purely interpretive. Surveys showed that typically only 1 to 4 times performance could be achieved using interpretive optimisations, but 10 to 20 times speed up was possible with some form of dynamic compilation. Project Monty was born, as an attempt to implement a customisable Hotspot style compiler for embedded applications.

Monty is targeted at ARM systems only and takes advantage of the instruction set, cache architecture as well as optimising the execution engine and resource allocation specifically toward maintaining a small memory footprint. The main aims of Monty are simplicity and compactness. A highly optimised interpreting engine, a simple one pass compiler with relatively few optimising features and a highly efficient garbage collector help to realise these goals.

### 2.1.5 Other J2ME systems

Other Java virtual machines are available which are hand optimised for specific processor targets and achieve notable performance increases and size reduction in comparison to more portable solutions. Some systems employ forms of advanced dynamic compilation taking advantage of optimisations only possible at run-time, giving further performance boosts, at the cost of a larger virtual machine and application footprint. Another solution is needed in very low memory environments, as software systems always trade off memory usage against performance.

## 2.2 Java Acceleration Hardware

Dedicated Hardware for the acceleration of Java becomes an option when software approaches are too slow, and possibly require too much memory, both in terms of runtime costs and the static size of the virtual machine. The idea of hardware approaches is to implement parts, or all of the JVM functionality in fast dedicated hardware, either attached to or integrated into a standard processor or as a Java specific architecture.

Decoding of byte-codes and generation of small native code sequences can be done in very little time and hardware for the majority of frequently used byte-codes, this has a big impact on the performance of a JVM. This is the simplest approach and allows a simple interpreter to decode the remaining byte-codes in software with a reduced memory footprint over even a simple interpreting JVM. Adding more functionality is a trade-off between memory usage, hardware complexity, hardware compatibility and execution speed and has been explored in the following hardware Java accelerators.

### 2.2.1 PicoJava II

#### 2.2.1.1 Instruction Set Architecture

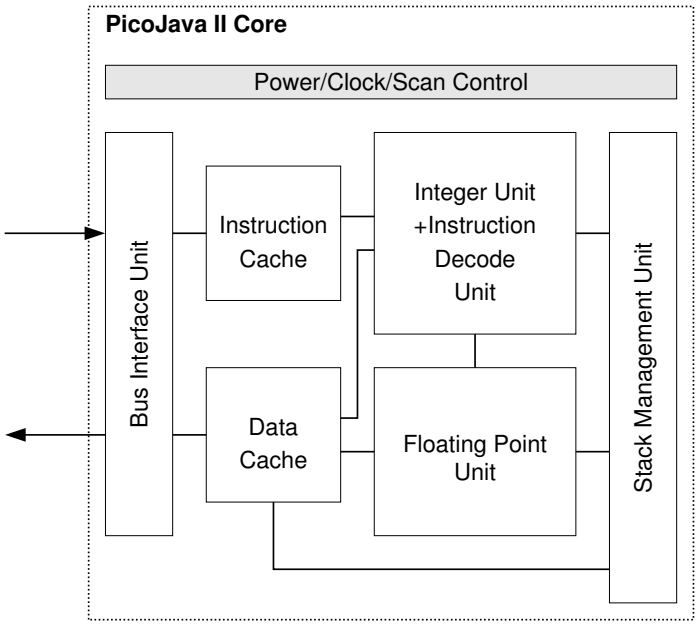


Figure 2.1: The PicoJava II Core.

PicoJava [35] is a micro-processor designed by Sun Microsystems which executes Java binaries natively. PicoJava was the first hardware platform dedicated to executing Java, and is now publicly available, as Synthesisable verilog RTL code, under Sun's community source licencing program. PicoJava is configurable and can has a gate count ranging from around 100,000 to 400,000 gates depending on internal cache configurations and the target circuit technology chosen. The core [46] is shown at block level in figure 2.1. An overview of the pipeline structure is shown in figure 2.2.

| Fetch | Decode | Register | Execute | Cache | Writeback |
|---|---|---|---|---|---|
| Fetch Cache line for decoding | Decode insructions + perform folding | Get operands from stack cache/register bank | Execute instruction/ do branch calculations (may be micro–coded and take >1 cycle) | Access Data Cache | Write results to the stack cache/ register bank |

Figure 2.2: The PicoJava II Pipeline.

PicoJava executes all 226 Java byte-codes as specified in the JVM specification

[32] (although 30 are actually handled by software traps) and also executes extra instructions needed to control the processor at a hardware level. Low level functionality is not specified or needed in a *virtual* machine specification, but is required to construct an operating system and support the Java run-time environment. In total 115 extra instructions are added in the PicoJava instruction set architecture to directly control memory and internal register accesses, support the construction of system software, support the use of other programming languages and allow for system diagnosis. These extra instructions allow for the kind of hardware access prevented in the Java security model, but are needed to facilitate a functional Java implementation. For instance, it must be possible to control memory access explicitly in order to allocate space for objects at run time.

PicoJava partitions the Java byte-code space into sections depending on complexity of implementation. Most byte-codes are executed in a single cycle, these being the core arithmetic, stack and data access instructions. Around 30 moderately complex and reasonably common byte-codes are executed in multiple cycles by micro-coded instructions. The remaining 30 or so more complex byte-codes are handled in software, which may take tens or hundreds of cycles to execute, but these byte-codes are infrequent so little penalty is paid relative to the hardware saving. These complex instructions are also open to different implementation strategies (when memory allocation and operating system issues come into play) making software implementation the most sensible, flexible option.

#### 2.2.1.2 Instruction Folding

A key feature of PicoJava is a technique employed at the instruction decode stage called *instruction folding*. It is well known that although stack based instruction sets like Java byte-code have a high code density due to their small instruction lengths. Operand storage is implied by the current state of the stack, rather than explicitly specified in each instruction, as addresses in memory or as indexes to registers. Stack based instruction sets unfortunately compensate for this reduced code density with the need for extra execution cycles to setup the stack as required by the arithmetic instructions. Fortunately this problem is reduced as results from previous computations are placed on the top of the stack. PicoJava attempts to combat this problem by caching up to 64 of the top stack entries making them randomly accessible as operands for the ALU. PicoJava takes advantage of the stack cache, which is effectively a register file, with two read ports and a single write port,

to circumvent redundant stack manipulation. In Java much data processing occurs on local variables, which are frequently loaded from their location around the start of a stack frame, to the top of the operand stack, then operated on and stored back to a local variable. PicoJava can spot this type of byte-code sequence and execute the operation in a single cycle, so long as the local variables are cached within the top 64 entries of the stack. Up to two cached local variables can be taken as input to the ALU on the two read ports and the result written back on the write port to either a variable or the top of stack if necessary. Sun claim that this instruction folding system eliminates almost all of the overhead of a stack based instruction set architecture, with 23 to 37 percent of all instruction executions being folded into other instructions [35]. An example of successful instruction folding is shown diagrammatically in Figures 2.3, 2.4 and 2.5.

| Java Language | Java Byte–Code | RISC Machine–Code |
|---|---|---|
| a = a - b | ILOAD_2 | SUB R1, R1, R2 |
| | ILOAD_1 | |
| | ISUB | |
| | ISTORE_1 | |

Figure 2.3: Example of Java Byte-code Versus RISC Code.

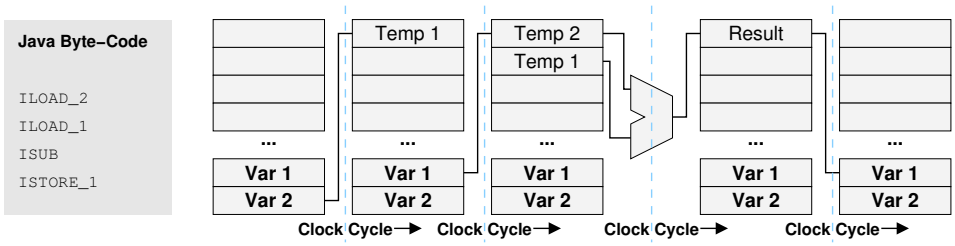


Figure 2.4: Execution of the Code Sequence: No Folding.

The problem with this type of instruction folding is that this mechanism implies complications over a simple processor design increasing the size and power consumption requirements. Design concerns include a relatively large register file, with stack spill and fill mechanisms. Instruction fetch and decode logic is also made somewhat more complicated and must be capable of looking ahead in the instruction stream and determining folding patterns. PicoJava looks at a window of 7 of 16 pre-fetched bytes. Even if patterns are detected, they can only be executed in a single cycle

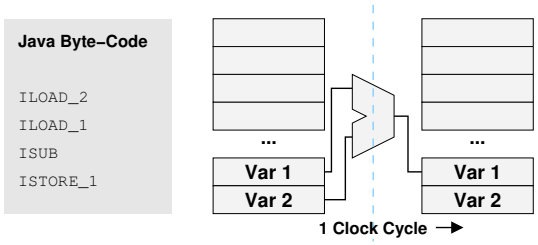


Figure 2.5: Execution of the Code Sequence: With Folding.

if indeed all needed operands are stored in the stack cache. Branch latency performance may also be affected by this need for lookahead or pre-fetching, although it is noted that Java byte-codes are typically shorter than RISC instructions, for example, reducing buffering requirements.

Possibly as a result of the complexity of PicoJava, it has found fairly few application areas and has not thus far been a big commercial success. For instance it is not really suited to low-power embedded systems, where Java acceleration is highly desirable. The system designer also has the problem of using Java exclusively to code the system software, which is a big change to make if other established embedded processors will have been used in the past. It does not make sense to throw away existing system software, which may be written in assembly language to meet performance and size requirements; such software may be difficult and time consuming to recode in Java.

### 2.2.2 JEMCore™

JEMCore [49, 23] is a Java micro-processor by Ajile Systems which executes Java byte-codes natively. It has a native implementation of threads, activated by an extended set of byte-codes, simplifying the implementation of embedded operating systems and also results in rapid context switching times, claimed to be under a microsecond. JEMCore is designed to support J2ME applications where some level of real-time performance is required. The real-time Java specification is directly supported. JEMCore is an entire embedded processing system, in product form it supports many interfaces allowing for a very compact solution to many embedded system designs. The drawback of this type of product is that although it allows for the construction of an entire Java enabled embedded system, with only one CPU core, it requires a complete shift to Java, which could mean re-writing lots of well

tested and trusted code, this may often be too expensive to consider.

JEMCore is a simple microcoded processor architecture and claims to be implemented in 25 to 35 thousand gates. Optimisation for speed is facilitated by programmable custom instructions which can be micro-coded by a system designer to replace invocations of library functions. Ajile claim 5 to 50 times speed up is possible, over the KVM, depending on how complex the algorithm being optimised. In a system with a secondary or host microprocessor, this type of optimisation would be achievable using Java's native interface. This method of accelerating Java is very labour intensive, and cannot really be seen as an architectural feature. This processor is relatively small, by gate count, so clearly a decision has been made not to over-complicate the implementation, and also to keep power consumption low. JEMCore also does not have a cache, but instead on chip ROM and RAM, with predictable latencies, to help support the Java real-time specification [9].

### 2.2.3 Moon2

Vulcan Machines' Moon2 [34] is a stack based processor intended for direct execution of Java byte-codes. It is available as an IP block, for fabrication or inclusion as a soft-core targeting FPGA's. The processor revolves around a microcoded control unit, and integrated 256 entry embedded SRAM stack. The implementation cost is 27,000 gates, 3K microcode ROM and 1K single ported RAM.

It is reported [34] that by caching the stack, simple byte-code folding optimisations can be carried out. Pushes to the top of stack from cached entries can be factored out, resulting in a reduction in memory accesses. This is similar to the approach taken in the PicoJava processor.

### 2.2.4 Lightfoot

The Lightfoot processor series [42, 50] is a stack based processor which claims 8 times acceleration over a RISC based processor interpreting Java. There is a one to one mapping from selected byte-codes to Lightfoot instructions, in order to obtain optimal performance.

The processor has instruction and data memory interfaces, and is based around a three stage pipeline. Interestingly, the processor supports C and C# through the Microsoft common run-time environment along with Java byte-codes, and is aimed at FPGA implementation as well as inclusion as IP on customer's silicon.

The Lightfoot has a 256 word register space, and has support for soft byte-codes, which branch directly to handler code in memory. The register bank has four parameter cache registers that can hold the first four parameters to a method. The data stack is cached in 8 on chip registers. There is a return stack, which can be used as an index register to access memory, auxiliary stack space or hold subroutine return addresses.

### 2.2.5 JOP

JOP [40] is a Java byte-code direct execution core designed for minimal implementation cost. It has four pipeline stages: byte-code fetch, microcode fetch, decode and execute. Each byte-code is implemented by a micro-code sequence, fetched in a single pipeline cycle. Microcode is designed specifically to implement the Java byte-code set, and has direct access to 16 local scratch variables to hold state. ALU operations are performed only on the stack through two top of stack (TOS) registers. On chip memory acts as a stack cache of 128 entries, and reduces memory stalls. The memory architecture takes advantage of the fact that results need not be necessarily stored back to memory in the same cycle as being read, reducing the hardware cost of the stack cache.

JOP was implemented on an Altera FPGA, and used only from only 1077 logic cells, with a memory requirement of 3.25K. Speeds of 101MHz are reported in the thesis, which is double the clock speed of the commercial Lightfoot, with roughly one third of the implementation area. This is roughly 10 percent of the gate count of the Sun PicoJava processor core. JOP is claimed to be the smallest hardware JVM implementation to date, with the highest performance per gate, per cycle. Absolute performance at 101MHz is still over 10 times slower than the Sun JIT compiler JVM on a 266MHz Pentium MMX processor, running a set of arithmetic and networking benchmarks.

### 2.2.6 JStar

JStar [11] is a Java processor system designed by Nazomi Communications. In contrast to the previously described Java hardware JStar is not a stand-alone processor.

The architecture of JStar is shown in Figure 2.6. JStar is shown at the interface of a general purpose CPU and some level of memory hierarchy. JStar acts as a translator between Java byte-codes stored in the memory or cache of a system and

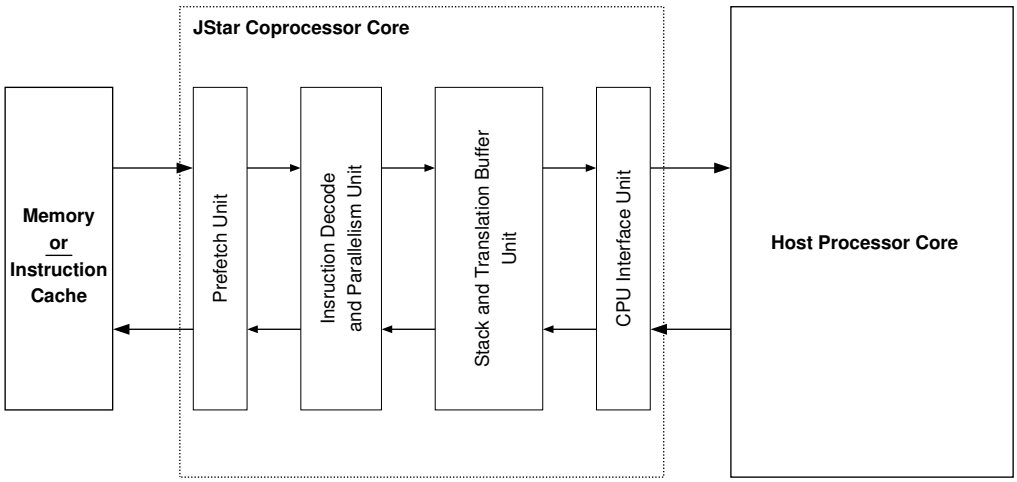


Figure 2.6: JStar Co-processor Architecture.

the system's processor, which is expecting some other native instruction format. JStar translates 160 of the 226 JVM byte-codes to native instruction sequences for ARM, MIPS and other processors, depending on the particular product. This has the benefit of speeding up execution of commonly used instructions, but also reduces the memory footprint of the software part of the JVM as these byte-codes no longer have to be trapped. Further memory reduction takes place as even unhandled byte-codes are now at least detected in hardware, taking further work away from the slow software, reducing the memory footprint and increasing system speed. JStar also performs some level of byte-code folding.

The currently available incarnation, JA108, connects via a standard SRAM memory bus, as a bridge to memory. The chip runs at up to 104MHz and includes an integrated 4K, 2-way set associative byte-code cache and 2K, 2-way set associative data cache. It is implemented in 180nm fully static CMOS, to reduce power consumption.

JStar is an efficient solution to speeding up Java, as it is almost transparent to the system. Software JVM components will need to be significantly changed to enable the hardware acceleration, the benefit being a reduction in the number of byte-code handlers necessary leading to reduced memory requirements. Power consumption of the system is possibly altered with the introduction of extra logic. JStar is available as a single chip package, or as IP for inclusion on custom silicon. Power consumption will be far lower if the logic is integrated into the processor chip, as no extra chip to chip communication will occur. One problem with JStar is

that it includes a relatively complex host interface and RAM interface at its inputs and outputs; hardware which would be unnecessary if the translation stage was integrated more tightly into the processor hardware. Accomplishing this would only really be possible if the host processor core IP could be modified or re-designed.

### 2.2.7 Jazelle

#### 2.2.7.1 Jazelle DBX

Jazelle [3] is a Java hardware product made by ARM ltd. This accelerator is different to others in that it is very tightly coupled with a host ARM processor, in fact it is a pipeline stage in the design. Jazelle is essentially a Java byte-code decoder and simply translates byte-codes to native ARM instructions to be sent to the processor's execute unit. Thus Jazelle is an extra decode stage in the processor. Initial descriptions of Jazelle suggest that the Java decode pipeline stage is only connected into the pipeline when the processor enters Java mode, and this stage sits before the standard ARM decode stage outputting sequences of ARM instructions for each byte-code it encounters. An alternative strategy which may be used in future versions is to add a parallel Java decoder which takes over from the ARM decoder in Java mode and interfaces directly with the processor's execute stage, leaving the pipeline latency the same as when not in Java mode.

Jazelle has the advantage of being a relatively compact piece of hardware in comparison to the others described here. It is simple, as it has very modest interface logic and only performs simple translations of individual byte-codes. ARM quote 12,000 gates are required for a Jazelle implementation, much smaller than other Java hardware. Much hardware is shared with the host processor, reducing costs over even similar units like JStar. 134 byte-codes are directly handled in hardware (on the ARM926EJ) and others are handled by ARM's highly optimised JVM support software, VMZ. ARM claims that software handled byte-codes only occur 5% of the time, resulting in good performance figures and reduced power consumption over a software JVM. Despite the extra hardware fewer cycles are needed to execute the same Java code. Jazelle hardware also removes the need for an interpreter loop as such, like JStar, implying further reductions in memory cost and decode time and power consumption.

The main drawback with a Jazelle style system is the need to make changes to the architecture of a processor core to support efficient integration with the Java

decoder. This is impossible for many system designers due to licensing issues, or due to the cost of developing their own processor core. The processor core also has to be one which is widely adopted in the target market, like the ARM architecture is, otherwise there is little advantage over a stand alone Java core.

#### 2.2.7.2 Jazelle RCT

While Jazelle DBX is short for *Direct Byte-code Execution*, Jazelle RCT [22] supports *Run-time Compilation Target* code. The second generation Java acceleration solution from ARM provides support for JIT compilers in their new range of Cortex-A and Cortex-R processors aimed at high performance embedded systems. The extensions provided are minimal and are aimed at reducing start-up times and improving code density and performance.

Essentially Jazelle RCT adds an alternative set of 16 bit Thumb instructions which can be enabled by changing the processor mode. These instructions replace some existing Thumb instructions with Java specific alternatives. Although primarily aimed at Java, the extensions can be used by other operand stack based binary translators such as those implementing Microsoft's C# with Microsoft Intermediate language, *MSIL*. Only 12 new instructions are added: two to change processor mode, one for array bounds checking, two to branch to handler routines, and seven memory load and stores, enabling easy access to local variables and array indexing.

It is claimed that statically compiled Java code can be from 7 to 44 percent larger than Java binaries. This compares with a figure of around 300 percent for 32 bit ARM code. This mode of Java extension has a very small hardware cost, but makes JIT compilation more attractive, reducing memory requirements significantly. It must be noted that the above code sizes are compiled ahead of time, a JIT compilation environment along with original binaries is still a significant memory overhead compared to a direct execution or interpreter based solution.

## 2.3 Summary

A series of approaches to Java acceleration have been reviewed, looking in detail at implementation costs and specifically their suitability for embedded applications. While software approaches have improved a great deal over the first generation interpreter JVM environments, their memory requirements are large. This does not make them very applicable to small embedded systems. Hardware approaches show

promise, the most efficient being processor decoder extensions which minimal extra hardware but retain compatibility with existing embedded code and devices.

# Chapter 3

# Architectural Techniques

In order to improve on existing hardware support for Java run-time environments, a decision was taken about what approach to take. The remainder of this thesis introduces an acceleration system targeted at integration with embedded processor pipelines. This chapter justifies the optimisation approaches taken, through evaluation of the problem and proposed solutions.

## 3.1 Translation Techniques

Among the surveyed architectures in Chapter 2 were three broad categories of approach to Java acceleration. The key difference between these approaches is the position of acceleration hardware in the system, discussed below in Section 3.2. The potential design choices are described below.

### 3.1.1 Direct Execution

Direct execution of Java byte-codes is perhaps the most obvious to providing acceleration for Java applications. Providing support for the decoding and execution of each instruction defined in the specification of the JVM is not however a trivial matter. The reason why it is difficult to implement a minimal embedded processor supporting Java, is that the byte-code set was designed for interpretation in software. Many byte-codes operate on quite abstract entities such as: objects, arrays and memory allocation. There are also additional requirements, such as bounds checking on array accesses, complicating matters further. The result is that, in order to implement such a processor, a micro-coded architecture allowing for complex

multi-cycle instructions must be implemented. This conflicts with the typical profile of low-power RISC processors which tend to have simple, efficient hardwired control structures.

Along with the complexity of implementing the whole set of byte-codes, which may additionally require operating support, there are further complications. The byte-code set used in the JVM is not really suited to implementing low level code, for device drivers and operating system functionality. The solution in direct Java execution solutions is to add extra instructions for this purpose. The end result in effect is an assisted interpretation solution, but with all of the disadvantages of the bespoke instruction set for system code.

Cores such as JOP have shown that small implementations can be achieved, overcoming some of the problems with cores such as PicoJava. In practice such cores would be required in addition to other embedded processors in many applications, making them less attractive than a more minimal set of extensions to existing processors. An extension solution could handle the more complex byte-codes in software, and would be more efficient in terms of total silicon area.

### 3.1.2 Assisted Interpretation

Hardware support for an interpreter can be provided in many different ways. The central idea being to add extra instructions, or provide some direct execution features, to a processor in support of the JVM. This approach has been taken most successfully with the Jazelle [3] extensions to ARM architectures.

Jazelle (DBX) supports direct execution of a subset of the Java byte-code set, while allowing for branches direct to handler code for complex instructions. This removes the standard interpreter loop, processing simple byte-codes very rapidly without software intervention.

An alternative approach would be to provide extra instructions capable of implementing interpreter loops very efficiently, and also instructions for complex operations such as array access with bounds checking, or method resolution.

The drawback with this type of extension is that the host processor may not have the best pipeline for executing even simple byte-codes, meaning some performance would be lost to more dedicated processors. Key to this is the stack based nature of the JVM, ideally suited to architectures with many registers. Changes in Jazelle are restricted to the decode stages of the processor in order to minimise implementation cost and aid portability between cores. In future systems, if Java

is a very important requirement, changes to the execution pipeline could be made aiding the Java decoder, while remaining cheaper than adding a dedicated Java core to a system.

### 3.1.3 Assisted Compilation

Assisted compilation is about supporting ahead of time and just in time compilation. The only example of this documented here is the Jazelle RCT. The aim being to reduce the size of compiled Java byte-code by providing a more compact representation of commonly used operations, mainly indirect memory access to local variables and arrays. The second aim is to support the efficient compilation and reduce start-up times of Java applications. In the case of Jazelle RCT this involves support for interpreting byte-codes through quick lookup and branching to software handlers.

## 3.2 System Partitioning

Hardware supporting Java can be positioned at many points in the architecture of a system. The following options are available, each implementation choice providing a range of benefits and problems. Important issues are: ease of integration, performance, available bandwidth, latency of mode change and branches, power consumption, complexity of implementation and manufacturing cost.

### 3.2.1 Dedicated Processor

A dedicated Java processor core could be a separate chip or integrated with other system components on a single die. Integration would be relatively trivial, as no changes are needed in the internals of other processor cores, and the core would not need to be specific to a particular platform. The main issue would be access to memory, and handover of control to the Java core. This would mainly be a software integration issue, as the core could share the memory bus with the host processor. Arbitration schemes could be an issue depending on how the processors would operate in parallel and share memory.

The possibility for high performance, is greatest with a dedicated Java processor, as it will be tailored in every possible way to implement the JVM. Memory

bandwidth and latency are not an issue due to direct connection to the bus, also there are few dependencies and communications with the host processor.

Power consumption is an issue for a stand alone core, as other approaches try to minimise the gate count by sharing hardware with the host processor, this is more of an issue if the trend of increasing static power consumption continues. A separate chip solution will also incur extra power costs. All of this is offset against the potential efficiency of a Java specific architecture.

Design effort and implementation cost, will likely both be greatest when taking this approach. A complete processor pipeline, system interface and possibly caches must be implemented. These features will require a large silicon area compared to other types of acceleration hardware. The costs are also likely to be greater, as a whole processor core will need to be licensed. If implemented as a separate chip, implementation costs could become prohibitive.

### 3.2.2 Memory Bridge

A memory bridge is an elegant solution to the acceleration problem, targeting code for the host processor from fetched Java binaries. The bridge is placed between the processor and its instruction memory. Java byte-codes are intercepted as they are fetched from memory by the host processor, the bridge then issues instructions to the host processor implementing the functionality of the fetched byte-code. The host processor need not be aware that the code at the fetch address is Java byte-code as the bridge is translating to native instructions. The interface can be a standard memory type, allowing for easy on chip and off-chip integration with a wide variety of host cores. There is however the problem of managing caching at the host core, which will be addressing bytes, and possibly receiving multiple words in response.

The main issue with this type of acceleration hardware is the extra latencies incurred, in performing all operations. It is possible that the normal operation of the host processor will be degraded by the extra memory interface logic. The main performance hit would be taken when executing Java code, as latencies for branches could be larger than in a more integrated approach, due to the Java decode stage being behind both cache and fetch stages in the processing pipeline.

It is possible that some of the latency problem could be solved with an extra layer of cache, and branch prediction in the bridge. Overcoming such problems takes away much of the elegance of this design approach, which in a minimal form would be small, easy to integrate and low power in comparison to a dedicated Java core.

### 3.2.3 Cache Bridge

A Cache bridge would bring many of the same advantages as the memory bridge. Although restricted to an on-chip implementation, the latency between generation and execution would be reduced for the translated code, improving branch latencies slightly.

This approach provides some of the advantages of adding a Java decode stage to the processor pipeline; in this case before rather than after the fetch unit. The advantage being that it may not be necessary to alter the processor core at all to design such an accelerator. This is advantageous to anybody does not have access to the internals of the host processor design, which is typically almost everybody outside of the manufacturer.

### 3.2.4 Co-processor

Some processors, such as ARM offer a coprocessor interface allowing a set of instructions to be implemented outside the regular pipeline. Normally this involves adding some extra instructions, or an offload engine for processing blocks of data. This approach would not typically suit a Java accelerator, as it is not usually possible to add support for a whole new instruction set. There may also be problems with fetching the required instructions with reasonable latency and at the required rate.

An example would be the ARM 7 co-processor interface [4], requiring a matching three stage pipeline. This would not allow for the type of integration required for an efficient accelerator, where sharing of the host processor's resources is desirable. The resulting hardware may also be larger than for a single stage solution. Extra support instructions for compilers or interpreters could potentially be added using this method, without needing to modify the host core.

### 3.2.5 Java Decode Stage

Adding a Java decoder stage to an existing processor, is a very neat solution to improving embedded Java performance. The main obstacle to this being the necessity to modify the host processor core. This only really leaves this avenue open to processor vendors. However embedded processor cores often need customisation to suit different markets, making this a very sensible option to take for companies supplying the mobile, televisual and smart-card space.

A Java decode stage is tightly integrated into a processor pipeline, allowing for minimal latencies and takes advantage of existing branch prediction hardware, without imposing additional restrictions on bandwidth or interfering with normal processor operation. Selected byte-codes can be executed directly by generating instructions for the host processor, or directly controlling the execution pipeline. A big advantage here is the ability to manage state and registers without having to store it in the general purpose processor registers, allowing better utilisation of the host core. The second advantage is that unhandled byte-codes can trigger branches to handler code, without the need for an explicit interpreter loop, making further gains over software approaches.

The benefits of adding a Java decode stage to a processor over alternative solutions are mainly about efficiency. Efficiency in terms of sharing the host processor's execution resources and minimising additional logic, which leads to savings in power and implementation cost. Secondly efficiency in terms of performance, with the possibility of approaching the performance of a dedicated Java processor core, as this is essentially what the modified core becomes. The advantage being the availability of the host instruction set for implementing complex operations and low level operating system code. The disadvantage being the execution unit and register architecture not being specifically aimed at executing Java byte-codes.

### 3.2.6 Other Processor Extensions

It is possible to add extra instructions to a processor core to aid with implementing features of the JVM. Such instructions are present in the ARM Jazelle RCT extension set aimed at improving the performance and density of object code in a JIT compilation environment. Such extensions could also be used to improve the efficiency of interpreter loops and byte-code handlers, by supporting natively the kind of indirect memory accesses and run-time checks necessary. Such an approach certainly requires less implementation effort, as issues revolving around state and interrupts generated by new processor modes are avoided. The resulting performance and memory savings may not however be as compelling as in more direct approaches.

## 3.3 Implementation Strategies

One of the central inspirations for this investigation of hardware support for embedded Java, was the work at The University of Manchester on low power asynchronous microprocessors [17, 18, 19, 20, 38]. In such self-timed processors aimed at embedded applications, as the pipeline is free of a centralised clock it can operate with timings natural to the current instruction stream rather than being locked to the worst case period. This quality was deemed to suit the nature of binary translation well, and was a key factor in the design and exploration of Java acceleration hardware presented here. These issues are discussed at length in Chapter 4, when describing the initial Java support hardware.

Below follows a brief overview of asynchronous circuit styles, used to implement such designs.

### 3.3.1 Asynchronous Logic

The majority of computing systems and digital circuits are currently designed using a synchronous design methodology. This means that the whole circuit or system is synchronised to a single clock signal. This clock signal is used as a reference for parts of a circuit which must communicate with each other. This involves some form of latches at each end of the communication, which assume the incoming signal is ready to be sampled when the clock signal transitions. Other logic circuitry used to process the data in some way is placed between latches. The clock period must be long enough to allow the signal to propagate through the logic between communicating stages and remain steady long enough to be sampled by the receiving latch. This fundamental timing constraint must hold true for *all* communicating stages in a clocked system.

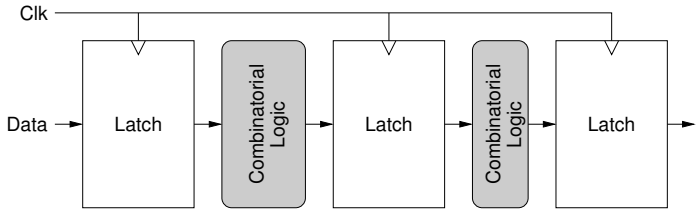


Figure 3.1: A Clocked Pipeline Design.

A clocked pipeline is shown in figure 3.1, showing latches which sample data

on clock edges, connected via *combinatorial logic*, which processes data as it moves between stages. Once data has been sampled, the newly sampled data is presented on the output of the latch, which proceeds through the logic or to the output of the circuit to be sampled on the subsequent clock edge. Although all clocked systems are not built using edge triggered latches, this structure is common to all clocked systems and serves here to demonstrate the advantages and problems with using a synchronous design methodology.

The reason for using synchronous design methodologies is simplicity. The only conditions which must be obeyed are the timing constraints at the latch, this means making sure signals are correct and steady for the setup time and hold time of the latch around the time of the clock edge. Logic is allowed to glitch so long as output signals are steady in time for the next clock edge. Synchronous circuits can be validated by finding the slowest path through all logic gates between each pair of latches, and making sure that the clock frequency is low enough to meet this constraint for the worst such path in the system. Once this has been verified, the circuit will work, assuming the logic performs the correct function, and is fabricated without error.

Serious problems only really start to arise in synchronous designs when they become large. With increasingly small circuit features in silicon, clock distribution becomes difficult as wire delays become larger relative to gate propagation delays. This phenomenon is called clock skew, as it becomes almost impossible to distribute a clock signal to all parts of a chip without intolerable phase differences between the clock signal at different sections of the design. Communication between distant parts of the chip, where the clock may be skewed must be carefully controlled. For example, in the DEC Alpha [14] a signal would take 4 clock cycles to cross the entire chip die. Distributing a fast clock, with low skew also requires lots of careful buffering, custom layout, and more importantly power. The DEC Alpha also used 1/3 of its huge power budget on distributing the clock signal across the chip. More recently the Intel Itanium 2 [41, 5] used a novel balanced H-tree for clocking, still consuming 33% of the 130W system power. This is claimed to be an improvement over a full grid approach, requiring fewer latching elements.

A further problem in large designs is that of IP reuse and integrating many large sub-systems on a single chip. Synchronous designs are defined by their clock speed; once a speed has been defined the design progresses around that, such that the timing constraints can be met. This leads to partitioning of a problem into blocks

which can complete processing within a clock period. What happens if you plan to solve a design problem you previously solved, but this time within a system clocked at a different speed for some reason? If the new clock speed is too low, you may not achieve necessary throughput with the old design. If your new clock speed is too high, you can not even use the old design as timing constraints will not be met. Either you must redesign the sub-system, generate a local clock for each module and find a *sensible* way of communicating or use a different design methodology altogether.

#### 3.3.1.1 Introduction

Asynchronous or *self-timed* circuit design [8, 44] is any way of building circuits without global synchronisation of components, such as a clock. In order for components to communicate in a circuit, some level of synchronisation is necessary. Synchronisation only really has to happen at the time communication happens, and only has to involve the units involved in any particular communication. This means that the synchronous design methodology involves a lot of potentially unnecessary communication, leading to power being wasted distributing clock signals to parts of the chip which may be inactive.

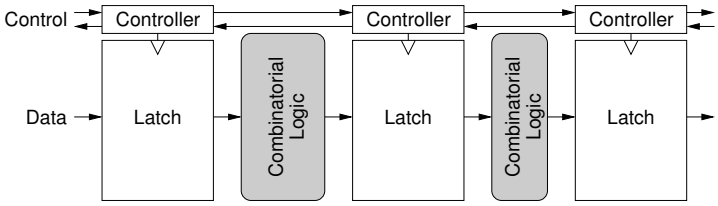


Figure 3.2: A Self-Timed Pipeline Design.

Figure 3.2 shows a typical self-timed pipeline. Instead of a clock synchronising all latches, each latch has a latch controller, which communicates locally with data inputs and outputs. If data is being sent between stages, then a transaction called a handshake occurs on the control signals, this determines when the data is valid, so allows correct sampling of data and therefore successful communication without a global clock. There are different approaches to generating this control information, the critical point is that each control path is closely related to the data path, allowing for fast communication to occur when logic between stages is fast and for stalling when a slow path is encountered. In a synchronous design, the whole system would

be clocked at the speed of the worst case slowest transaction, in a self-timed equivalent the system will run as fast as the data and logic will allow. The main overhead of self-timed design styles, however, is in the need to generate control signals at each communicating block.

Asynchronous circuits can possibly benefit from the following advantages:

**Low Power** Firstly there is no huge clock signal and distribution network. Also there is automatic *gating* of redundant parts of a self-timed system, only logic blocks where communication is taking place cause logic to switch. Logic which is not switching does not consume power, at least in CMOS technology.

**Low Peak EMI** Synchronous systems produce bursts of electro-magnetic radiation when logic switches, this is always on a clock edge, so large peaks are generated at multiples of the clocks frequency. In self-timed systems, this is not so as communication happens when it needs to, and when logic is ready, resulting in a much lower peak EM output. This is useful where interference may be an issue.

**No Clock Distribution** There is not the problem of distributing a global clock signal the whole way across the chip with low skew. However asynchronous control circuity is needed which will take up extra area.

**High Throughput** A self-timed circuit will run as fast as the blocks within it can communicate and process the data it is given. This means the performance will tend to that of the *average-case*. In synchronous designs the whole system is clocked at the worst case processing time, meaning performance will always be that of the *worst-case* behaviour.

**Robustness** Self-timed circuits will run in the presence of changing voltages, temperatures and fabrication parameters. Performance will however vary accordingly.

**Composability** Self-timed circuits present a standard interface and are designed without clocking assumptions, so will integrate with other circuits easily enabling simple design re-use.

**Security** Self-timed circuits can possibly be difficult to attack by analysing power signatures [54], as there is no clock reference signal to help the attacker. Also

delay-insensitive data encoding can hide data values as the same power will be consumed transmitting a 0 or a 1.

### 3.3.2 Self-Timed Communications Protocols

Protocols for communications between logic stages are needed in asynchronous design, in order to replace the clock as a source of synchronisation. The two main latch control styles are described below.

#### 3.3.2.1 Bundled-data

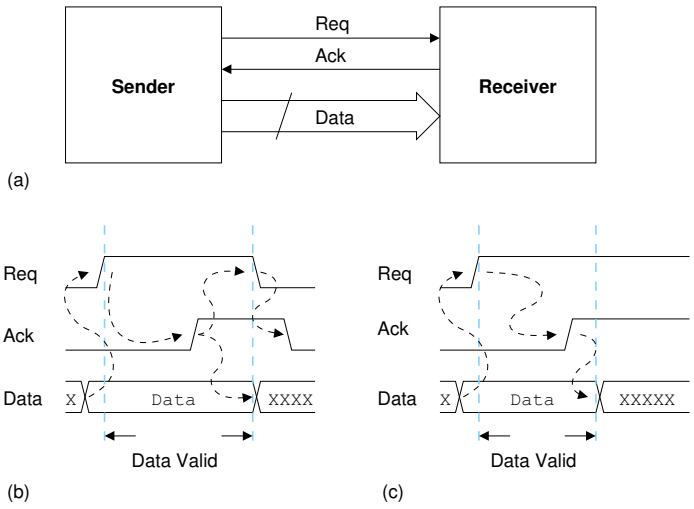


Figure 3.3: Bundled Data (Push) Protocol. (a) Latches (b) 4-phase (c) 2-phase

Figure 3.3 (a) depicts an example *bundled-data* communication channel. The protocol name refers to the fact that the data is encoded in a standard boolean fashion and synchronisation information covers the whole of the data bundle. This means that when a request is detected at the receiver, it is assumed that all of the data bits are now valid. It is possible to vary the protocols depending on when the data is valid, and also if the sender or receiver requests a transaction, but these examples show the main properties of such protocols.

Figure 3.3 (b) shows a 4-phase protocol, where the request ( *Req*) and acknowledge ( *Ack*) signals are level sensitive, so both must return to 0 before the next handshake can take place. In this handshake the sender issues a request to send data by asserting the data and raising the *Req* wire when it will be valid after propagating through the logic between the stages. Once the receiver is ready to receive

some data it waits for *Req* to be 1 and then samples the data, asserting *Ack* to acknowledge this to the sender, who can now de-assert *Req* and start working on the next data item. *Ack* is then de-asserted by the receiver and another transaction is now free to take place.

Figure 3.3 (c) shows a 2-phase protocol, which follows the same principles as the 4-phase protocol, except that there is no need to return the control signals to zero between transactions as the logic at each end is now sensitive to *transitions* rather than *levels*. The transaction is complete when the receiver sends a transition on *Ack*. Although fewer transitions are needed to communicate, transition signalling tends to be slow and expensive to implement in technologies such as CMOS.

An issue with bundled-data protocols is that the timing of the request signal and data is crucial, such that the order of events at the sender must be preserved at the receiver. This is done by matching delays between wires by inserting buffers to delay control with the data, care must be taken if data dependent timing (average case performance) is to be preserved. In order to be more robust and not dependent on a design's layout for correctness, the data signals can be used to convey validity, this is done with delay insensitive codes.

#### 3.3.2.2 Delay Insensitive Codes

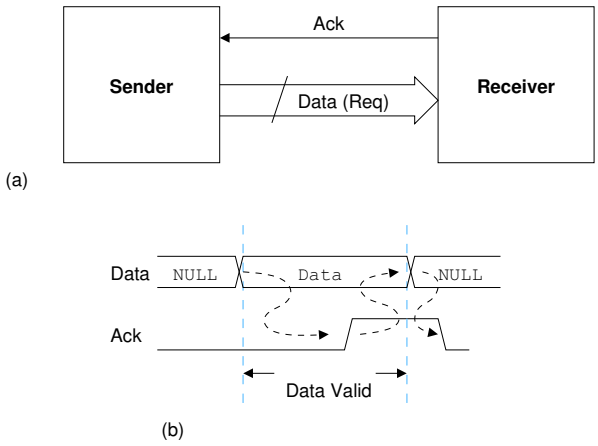


Figure 3.4: Dual Rail Protocol. (a) The communicating blocks. (b) 4 phase protocol.

Figure 3.4 shows a 4-phase delay-insensitive protocol, where the values on the data wires can be tested to check for validity, hence a request can be detected from the data. Once acknowledged the data returns to a null state, and then the

next transaction can take place. The implementation of such a protocol requires redundancy in the data, so that data values and also validity can be detected for each bit. Delay insensitive (DI) codes are such encodings. The most common DI codes are 1-of-2 (dual rail) and 1-of-4. Dual rail encodes each data bit on two wires, raising one of the wires signifies a 1, raising the other signifies a 0 and setting both to zero is the null state, setting both to 1 is illegal. 1-of-4 signalling uses 4 wires for two bits of data, raising one of these for each of the four binary states, and all zeros for null. The problem with such DI circuits is the need for extra decoding logic to detect if data is complete on the input to a latch, and also there is the overhead of having the extra wires. 1-of-4 logic potentially consumes less power than dual rail as only one transition is needed to transmit two bits of data, although both usually use return to zero coding doubling the number of transitions.

## 3.4 Summary

To support Java acceleration in an embedded system requires minimal implementation cost and power consumption, while providing maximum possible acceleration; the goal being efficiency. Reducing the number of processor cycles taken to execute Java code provides the required benefit to the user at the same time as potentially reducing power consumption. In order to reduce power consumption, the best solution is one with a minimum impact upon the existing embedded system in terms of unnecessarily replicating functionality.

### 3.4.1 Candidate Architectures

From the architectures surveyed at the beginning of this chapter, the most effective, are the memory/cache bridge and instruction decoder processor extensions. The reason being that less additional hardware is required for such solutions, as they take advantage of the functionality of an existing embedded processor core. A general purpose processor core will usually be required in such systems regardless of the Java execution hardware. Integration of such designs on the same die as the processor will further reduce power and implementation costs for large scale production runs.

Adding a byte-code decoder stage to an embedded processor pipeline is the most attractive option, as it provides the tightest integration. Communications latencies with the rest of the core, mainly concerning branches, are smallest at this point, giving the best potential performance with minimal hardware costs. Memory bridge

designs are complicated by further integration issues, concerning efficient signalling of state and management of latencies over the memory interface. Dealing with such issues not only affects performance, but requires extra logic, further reducing power efficiency.

The most efficient method of adding hardware support for Java in a wide class of embedded systems seems to be the extension of existing cores with a native Java instruction decoder. This approach satisfies the following requirements:

- Lowest additional hardware cost, while retaining existing advantages of host core, due to optimal integration point.
- Potential for low power consumption, second only to a dedicated Java optimised processor core.
- Potential for high performance, second only to a dedicated Java optimised processor core.
- Simplicity of integration, no interface complications, no need to develop extensions to JVM to support low level code and retains compatibility with libraries of existing code for host processor.

The only real drawback of this option is the requirement of modifying the processor pipeline. Changes are however almost entirely localised to the additional decode pipeline stage, allowing for relatively simple integration with existing cores. This was therefore the approach investigated in the remainder of this thesis.

### 3.4.2 Improving Byte-code Translation

Current embedded processor extension schemes such as Jazelle, are kept simple, decoding individual byte-codes in sequence, to conserve power and silicon area. A central aim of the research presented here, is to investigate the operation of processor extensions for the direct execution of Java byte-code. To the author's knowledge, there are no existing publications of such an investigation. It is predicted that further performance improvements can be demonstrated by applying techniques such as instruction folding, without overly increasing complexity, all within the framework of existing embedded processor cores.

### 3.4.3 Application of Self-Timed Design

Further to the investigation, it is thought that self-timed pipelines provide a perfect implementation environment for such a processor. The key goal being the exploitation of average case performance, for both processing time and power consumption. In the simple case, the nature of Java execution will require the generation of many simple stack management operations, without the need for instruction fetches. In more elaborate approaches, such as when instruction folding, the potentially time-consuming nature of applying optimisations can be accommodated without globally modifying the timing of the pipeline, or complicating the implementation with further stages. While stalls in the pipeline could possibly prevent any performance advantage being seen on this front, the advantage gained by retaining a simple pipeline structure will potentially be valuable in terms of both design cost and importantly for embedded systems, power consumption.

### 3.4.4 Conclusions

The following chapters of this thesis present an investigation into the implementation and improvement of Java processor extensions through the addition of direct decode and execution ability to existing RISC processors. This is deemed to be the most efficient avenue for the acceleration of Java in the memory constrained embedded application space. While work is based on existing commercial solutions, a full investigation into the performance of such designs has not been carried out before. Secondly there seem to be many areas where the execution of byte-codes can be improved, both through better management of the stack based byte-code operations, and by taking advantage of self-timed pipelining.

# Chapter 4

# JASPA

This chapter describes the design of the JASPA the Java Aware Synthesisable Portable AMULET. JASPA is a modified self-timed ARM processor capable of accelerated Java binary execution. The SPA [38] processor in development within the AMULET group at the University of Manchester has been used as part of this project and provides all the processor's features aside from the Java specific extensions presented here. Work on this Java enabled architecture was inspired by preliminary work by Ian Watson, who worked on the design and high level modelling of an asynchronous Java co-processor architecture alongside an existing model of an AMULET 3 [20] core. The previous accelerator design was described in the LARD [16] modelling language, which can be used to simulate the activity of asynchronous systems communicating through handshaking channels. The architecture described here is a prototype system and is mainly intended to explore the design space, and implementation options, in order to find areas worth researching in order to build better designs in the future.

## 4.1 Self-Timed Design

From the outset is was decided that it would be interesting to design an asynchronous Java accelerator/processor, firstly because to our knowledge it had not been attempted before but also because of the benefits that an asynchronous design style may bring to the architecture and implementation. The principal benefits of a self-timed architecture in this instance are simplicity and efficiency. Decoding Java byte-codes is a data dependent problem. The instruction set has been designed for software interpretation, different sets of byte-codes require different amounts of work

to be executed. This is in opposition to most modern processor architectures which have more orthogonal RISC type instruction sets making each cycle of execution as evenly matched as possible to maximise hardware utilisation and efficiency of code. In a self-timed system, this is not so important, only parts of the system in use at any point in time have any side effects. As long as frequently executed instructions are handled in an efficient manner, performance will be high and power consumption low. Natural cycle times may vary when interpreting different Java byte-codes in an accelerator, this will not pose a big problem at the architectural level in a self-timed system as the cycle time of any unit is not fixed, partitioning of the architecture would be necessary in a synchronous design.

The following features of asynchronous systems are hoped to be capitalised upon in the architecture and implementation of the Java accelerator described here. There is further explanation of these points on page 43. As well as being potential benefits of self-timed systems, they are also very desirable as architectural features considering the target market of hardware Java accelerators is almost entirely in embedded systems:

**Average case performance** High performance for simple, common byte-codes will be natural, while more infrequently used complex translation and optimisation stages can be incorporated without compromising the simplicity of the architecture.

**Clean, simple and fast** Flexibility of the asynchronous pipeline model should allow for a simple yet high performance architecture without compromise or complexity which may be needed in an equivalent synchronous system.

**Low power** Parts of the architecture not in use will not be activated, so minimal power will be consumed for any set of input data. Without extra power management features, power efficiency will naturally be a feature of the system.

**Security** Depending on the exact implementation technology, security against non-invasive attacks inherent in asynchronous technology may be available without extra work on the underlying architecture. One approach to this is to use the balanced dual-rail logic SPA employs [38].

**EMC** Good electro-magnetic compliance should be achievable if processing is parallelised well. Absence of a common clock will reduce peaks of electro-magnetic power output.

**Composability** Self-timed interfaces between blocks are not dependent on global timing constraints or specific frequency requirements. The Java decoder module will also not be constrained by a maximum latency. These features allow for a flexible architecture which can be quickly implemented and integrated with the asynchronous host processor core.

## 4.2 Architecture Overview

The main architectural qualities needed in the first version of a Java accelerator were decided to be the following:

1. Small Size
2. Low Power
3. High Efficiency
4. Quickly Implementable

As Java accelerators are targeted at embedded systems, cost and power consumption are always key factors. People can not charge a lot of money for embedded products and volumes are large, hence making the architecture simple should allow for a cheap to produce design. Low power is essential also, many embedded applications requiring high levels of Java performance are battery powered and need to be practical. Lower power means longer battery life, or physically smaller systems (as battery size can be reduced) which are both desirable features in personal embedded systems, such as mobile phones or palm-top computers. The architecture should also provide a high level of performance for its size, otherwise there would be no advantage over conventional processors other than the asynchronous implementation strategy, this would not really be making the most of the research opportunity.

The first version of the architecture is intended as a prototype. An implementation can be used to reliably show that the idea is a viable one. The initial architecture must therefore be quickly implementable in order to show architectural features are possible to achieve and that realistic assumptions have been made about what can be done in the future. One concern here is that asynchronous designs can typically be larger than synchronous equivalent due to the extra control circuitry, it must be seen if this can be balanced out through a simpler architecture and implementation free of clocking concerns.

### 4.2.1 Java Processing

The decision was made that Java acceleration would be implemented as an embedded unit in an existing processor. The accelerator would act as an extra decoder unit taking much of the work away from a software JVM interpreter, generating native instruction sequences for the host. One reason for this was that an embedded processor was already in development within the group. Secondly, in order to minimise the size and power consumption of such a unit, tight integration was important. This reduces the cost of interfacing and provides maximum bandwidth to the Java decoder. Having an established processor around the system also makes writing support software, testing and debugging simpler. Finally, this option was seen as relatively fast to implement, hence hopefully the most insight could be gained from the experience as more progress would be made this way.

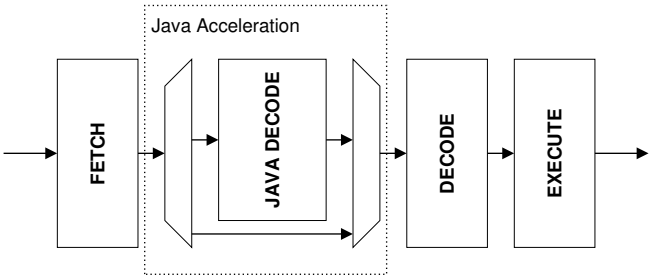


Figure 4.1: JASPA Architecture.

The Java decoder unit sits as an extra stage in the pipeline of the processor, this way it is decoupled from the rest of the processor, and has clean interfaces requiring minimal modification of the surrounding hardware. The decoder interface and general architectural structure is shown in Figure 4.1, the decoder unit is seen between the fetch and ARM decode stages, just as in ARM's Jazelle system. This also makes sense as in the future it may be possible to use the same system software as ARM's system, if the software-hardware interface they use is adhered to, currently this is only partially the case.

### 4.2.2 The Host Processor

The choice of host processor was very limited, as not many asynchronous processors have been developed across the world, however, luckily within the AMULET group in Manchester there were two choices. Either the AMULET 3 [20] which had been

used in Ian Watson's original model, or the new SPA [38] processor. SPA was chosen, not only because it was currently in active development, but also because it has a simple pipeline structure with well defined communications protocols, making the interfacing of extra Java components straightforward and helping make the effects of the decoder more transparent to analysis. SPA is entirely synthesised using the Balsa [6] hardware description language, unlike AMULET 3 which has a full custom silicon data path design, alongside hand designed standard cell control logic. Using Balsa would make modifications to the processor easier to implement and allow rapid construction of the decoder architecture.

SPA is an ARM compatible self-timed microprocessor with a 3 stage pipeline. Although still in development during the course of this project, it was mature enough to be used as a basis for the construction of a Java aware embedded processor. Figure 4.2 shows a detailed view of the module structure and pipeline of SPA. Although this architecture was chosen to be modified with Java extensions, it is thought that modifying the Java decoder unit for use in other (ARM) processors should not be a big problem, as most of the interfacing requirements should be constant between architectures.

An interesting feature of SPA is the fact that it is designed with security as its primary feature. Security is obtained by synthesising the processor with a specially developed dual-rail technology mapping. This technology mapping allows the hardware description written in Balsa to be realised as a dual-rail circuit with extended security features. This circuit technology uses two wires to transmit a single bit, one wire signalling a 1 and the other a 0. Combined with modified latch designs and arithmetic components, in the library Balsa uses to generate circuits, this should make power analysis attacks which are a common smart-card hacking technique much more difficult, if not impossible. Power analysis attacks look at tiny variations in power supply consumption to extract information such as protected encryption keys from hardware. Such variations are minimised in a dual rail system where ideally the same power is consumed communicating zeros and ones. Further research in this area is underway [54], exploring the use of asynchronous design styles as a method to make secure chips. As Java is now being used more and more in secure applications such as smart-cards, extra motivation is provided for the choice of SPA as a host processor.

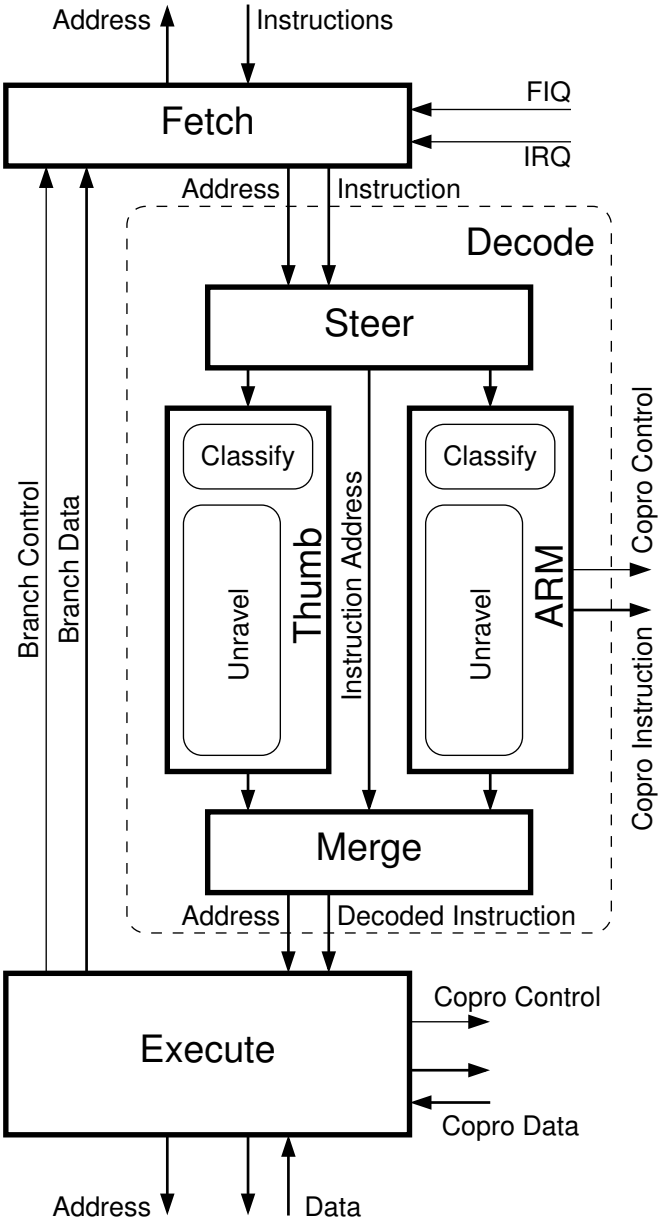


Figure 4.2: The SPA Architecture.

## 4.3 Integrating Java into SPA

Figure 4.3 shows the whole Java aware JASPA architecture. The Java decoder module is shown taking the output of the fetch stage in the pipeline. The result gathered here is either forwarded directly to the ARM decode stage or latched for processing by the Java decoder depending if the processor has been switched into Java mode. When in Java mode the processor therefore gains an extra stage in its pipeline to make translations from Java byte-codes to ARM instruction streams.

The decoder is placed as an extra decode stage in the pipeline in order to leave a clean interface at the input and output. Java byte-codes arrive in groups of 4 at the input, as the ARM processor has 32 bit instructions and SPA is single issue. ARM instructions are needed at the output. If the decoder was placed in parallel with the ARM and THUMB decoders shown in Figure 4.3, there would be less latency but the decoder would have then been sensitive to changes made in the SPA execute unit/ interface and would be less portable between different ARM cores.

### 4.3.1 ARM Extensions

As some information was available on the ARM Jazelle decoder system [3], it was decided to make the decoder conform as much as possible to the specifications of that system, in order that the system be comparable directly, and possibly in the future be compatible with the Jazelle support software. This decision mainly made the modifications to the ARM part of the processor more sensible, as designing a new interface standard to do the same job as Jazelle's would be fruitless. Only the following changes were needed in the ARM core:

1. Java bit added in processor status register.
2. `bxj` instruction added to jump to a block of Java binary code.
3. ARM program counter must be able to store 32 bit byte address in Java mode.

The `bx` instruction was already present in the instruction set to jump to Thumb mode (an extra 16 bit instruction set with high code density) from ARM mode, and also used to return. A small modification therefore allowed entry to Java mode and the corresponding return instruction was synthesised when an unsupported byte-code was detected. Adding the Java mode flag to the status register and branch to Java instruction also follow the same pattern as the Thumb extensions. As SPA is

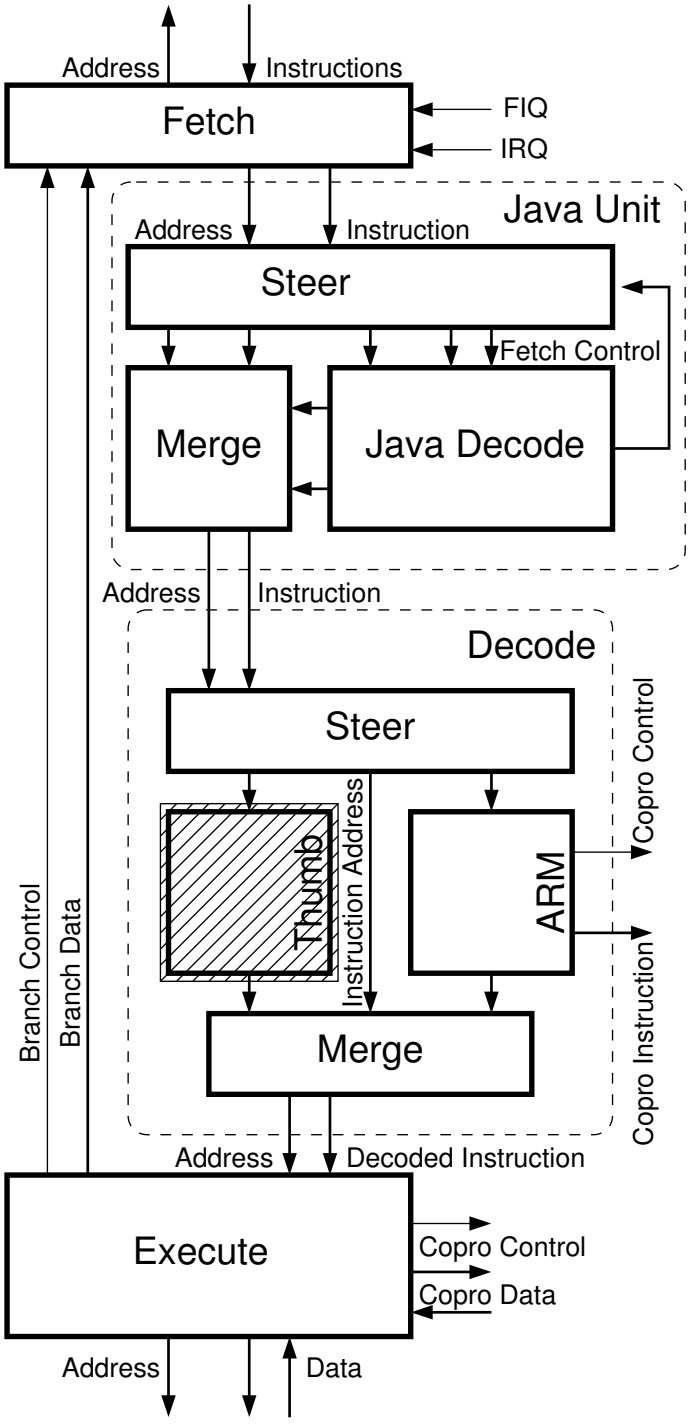


Figure 4.3: The JASPA Architecture.

a Thumb capable ARM compatible core adding these modifications was relatively trivial, although it was necessary to ensure the ARM instruction decoder supported the `bxj` instruction even if the Thumb decoder was absent. The instructions function remained the same; returning the processor to ARM mode. This instruction remains illegal in ARM mode when the Thumb decoder is not included in the design.

In order for the processor to support Java it must be able to support byte-addressing, as the start of a block of Java code could be at any byte in memory, as Java byte-codes are one byte in size and upward. As the ARM architecture has 32 bit instructions it only needs to fetch instructions from 32 bit word boundaries. Luckily, the ARM program counter register is byte-addressed, so can be set to any valid Java start address. Unfortunately the fetch behaviour of SPA is word aligned, so a mechanism for extracting the correct byte-codes is needed. The decoder block also has to cope with multi-byte byte-codes stretching over word boundaries.

### 4.3.2 Java Execution

As it was decided to accelerate Java solely through translation of byte-codes, Java stack operations must be mapped onto the ARM load/store RISC instruction set. Initially this mapping follows the convention adopted in ARM's Jazelle system. This is shown in Figure 4.4.

| Register(s) | Usage |
|---|---|
| R0–R3 | Stack Cache |
| R4 | Local Variable 0 |
| R5 | Points to handler routines |
| R6 | Points to Java Stack |
| R7 | Points to Java variables |
| R8 | Points to Java constants |
| R9 – R11 | For software JVM use |
| R12 | For hardware JVM use |
| R13 | Points to ARM stack |
| R14 | Java link register |
| R15 | Java program counter |

Figure 4.4: The ARM Register Mapping for Java Execution.

The most important feature of this register allocation is the use of registers R0 through to R3 as a stack cache. In order to perform any operation on data in a RISC architecture such as ARM, the input operands and data *must* be stored in the register file. In a stack based architecture such as the JVM the operands must

be popped from the top of the stack, with the result being pushed to the top of the stack. In order for convenient rapid execution in this case, the data currently being processed is stored in these four registers only requiring relatively slow loads and stores from memory when this small cache overflows or underflows. In this way single ARM instructions can perform the operation of simple arithmetic byte-codes, in contrast to JVM software where many ARM instructions would be executed decoding and handling such simple cases.

Figure 4.5 shows a sequence of four Java byte-codes which performs the Java statement: `local2 = local1 + 3;`. This example shows that in such simple sequences Java byte-codes can map onto single ARM instructions, as the stack cache is usually of adequate size for simple statements. This is the ideal case in this mapping scheme, as no instruction folding is attempted in this version of the Java decoder.

| ***Java Byte-codes*** | ***ARM Instructions*** |
|---|---|
| `iload_1` | `ldr r3, [r7, +#4]` |
| `iconst_3` | `mov r2, #3` |
| `iadd` | `add r3, r2, r3` |
| `istore_2` | `str r3, [r7, +#8]` |

Figure 4.5: Simple Translation Example.

Currently the JASPA Java decoder unit handles 80 byte-codes directly in hardware generating ARM handler sequences in each case; currently only integer arithmetic is dealt with. The basic handler routine for each byte-code is usually only one or two ARM instructions, although these are supplemented by stack cache management instructions when spills and fills are needed. Currently, when an unhandled byte-code is encountered by the decoder, ARM instructions are generated to branch to the appropriate handler code based on the value in ARM register R5. The corrected program counter for the current byte-code is stored in R12 so the software can return to the Java code if necessary. Table 4.1 shows the categories of Java byte-codes alongside a description of the generated ARM handler code.

From Table 4.1, it is obvious to see how little overhead these few byte-codes have when being translated by the decoder. In the future similar code generation can be built at little extra cost for other common byte-codes such as stack manipulation, type conversion and further arithmetic operations on different data types. The reason the above byte-codes were chosen for acceleration, was that they occurred in

| ***Java Byte-code Group*** | ***ARM Handler*** |
|---|---|
| `nop` | Single cycle |
| `iconst_0..5, fconst_0..2` | Single `mov` instruction |
| `lconst_0..1` | Two `mov` instructions |
| `bipush, sipush` | Two `mov` instructions |
| `iload, iload_0..3, fload, fload_0..3` | Single `ldr` instruction |
| `lload, lload_0..3, dload, dload_0..3` | Two `ldr` instructions |
| `aload, aload_0..3` | Single `ldr` instruction |
| `iaload, aaload` | Single `ldr` instruction |
| `istore, istore_0..3` | Single `str` instruction |
| `astore, astore_0..3` | Single `str` instruction |
| `iastore, aastore` | Single `str` instruction |
| `pop, pop2` | No instructions, just state change |
| `iadd, isub, imul, ineg, ishl, ishr iushr, iand, ior, ixor, iinc` | Single arithmetic instruction |
| `ifeq,ne,lt,ge,gt,le` | 4 to 5 instructions |
| `if_icmpeq,ne,lt,ge,gt,le` | 4 to 5 instructions |
| `if_acmpeq,ne` | 4 to 5 instructions |
| Unhandled byte-code | 4 instructions, correct PC (1), calculate handler address (2), `bx` (1) |

Table 4.1: Java to ARM Translation Table.

a small set of test and benchmark programs working with integers and basic Java language constructs using the SUN Java compiler, the ones implemented in software were ones requiring memory allocation and more ambiguous implementation strategies, for example method invocation with `invoke`. Further work is still to be completed on more detailed profiling of Java binaries to assess what is sensible to implement in future hardware, although other work at Manchester [15] has helped provide some insight into this area.

#### 4.3.2.1 Caching the Stack

The current hardware translation methods implemented rely on a stack cache to maintain a working pool of data to operate on, stored in registers R0 to R3. The problem with including this type of cache is that it implies state. Unless the top entries of the stack are always in the same order and position, different instructions must be generated depending on which registers contain the current top stack entries. Even if the top stack entries were always stored in the same place, there would be state concerning how full the stack cache was. This type of state can potentially

cause problems when taking branches and jumping back to ARM mode handler routines.

As the processor is pipelined, some instructions get fetched by mistake in the *branch shadow* as whether a conditional branch is taken is not decided until it gets to the execute unit. If byte-codes are interpreted as normal in the shadow of a branch which is taken, then the stack state in the hardware will be modified, but instructions generated may be ignored by the execute unit causing inconsistencies in the system. The stack state is also important when returning to ARM mode to deal with unhandled byte-codes, only the decoder block has details of the stack cache state, this is not currently accessible in ARM mode. The simple solution here is to store any cached stack items back to the stack in memory when returning to ARM mode, or when potentially taking a branch. Luckily this takes a maximum of four memory cycles and in the case of branches the stack cache is almost always empty when inside a method, at least with the compilation strategy taken by Sun's `javac` Java compiler.

#### 4.3.2.2 Taking Interrupts

Having stack state also imposes problems when dealing with hardware interrupts and exceptions. Many instruction sequences generated by the Java decoder can be multi-cycle. Due to possible stack cache management an interrupt could be taken at any time during such a sequence. If an interrupt is taken then it is possible that the stack state may be left inconsistent, if an ARM sequence is not allowed to finish. One solution is to disable interrupts until an instruction sequence has completed. This may impose unacceptable latency on interrupt handling making it difficult or impossible to interface with different hardware, or meet real-time constraints. Because of this problem, it must be the case that instruction sequences generated by the Java decoder are restartable. This means that when returning from an interrupt to a partially completed byte-code sequence, the decoder should realise this and allow for a restart, restoring a consistent stack cache state.

Although interrupt handling has not been implemented in the Java decoder architecture yet, it is thought that this can be achieved through checkpoints with relative ease. If the decoder can record the fact sequence generation has started for a particular byte-code, this can be checked at each decode cycle, when returning from an interrupt this problem can then be detected. To circumvent inconsistencies in state, any potentially non-restartable operations need to be logged, so they can

be handled correctly on the second pass. This part of the decoder is very dependent on the precise interrupt mechanism used in the host core.

In JASPA the only issue to deal with regarding returning from interrupts is ensuring that the correct byte-code is restarted, as the PC is only updated at word boundaries on each communication with the fetch stage. Internal state is not left inconsistent through the ARM instruction sequences, as stack-cache updates happen in a single atomic action with one ARM instruction which can only be aborted, not interrupted, once issued. In this sense, all instruction sequences generated by the decoder are restartable.

Figure 4.3 shows that in the SPA architecture interrupts are processed at the fetch unit. An interrupt causes an ARM 'instruction' to be synthesised to save the return address whilst the fetch unit begins supplying the service routine. A mechanism is needed to ensure that the saved return address is the *byte* address of the next pending byte-code in Java mode. However the Java instruction sequences are atomic, reducing the state stored in the Java translator; the interrupt is stalled by the Java decoder until a sequence is complete.

### 4.3.3 Decoder Block Interface

To interface the Java decoder block into the SPA processor is relatively trivial. The SPA core does not have a central control unit, unlike many conventional clocked processors, this allows for a very clean and manageable architecture. This architectural style is perfect for adding easily integrated extensions such as the Java decoder presented here.

Other than the addition of the extra Java mode and branch instruction discussed earlier, the only integration requirement at the Java decoder block is to take the output from the fetch stage of the pipeline and provide output to the execute stage. As well as data needed for processing at each stage, control information such as processor mode and the PC at the time of fetch are communicated to avoid a complex central controller, which would be difficult to design and maintain in an asynchronous environment, as it could potentially add unnecessary synchronisation between units reducing the advantages of asynchronous processing.

As the Java decoder was not intended to act as an extra pipeline stage at all times, the steer unit at the fetch interface is also important. This forwards ARM mode instructions from the fetch unit directly to the execute unit, only latching Java mode words for processing by the decoder. This reduces branch latency when

executing ARM code by a cycle, which is significant as the SPA does not perform any branch prediction.

## 4.4 The Java Decoder Unit

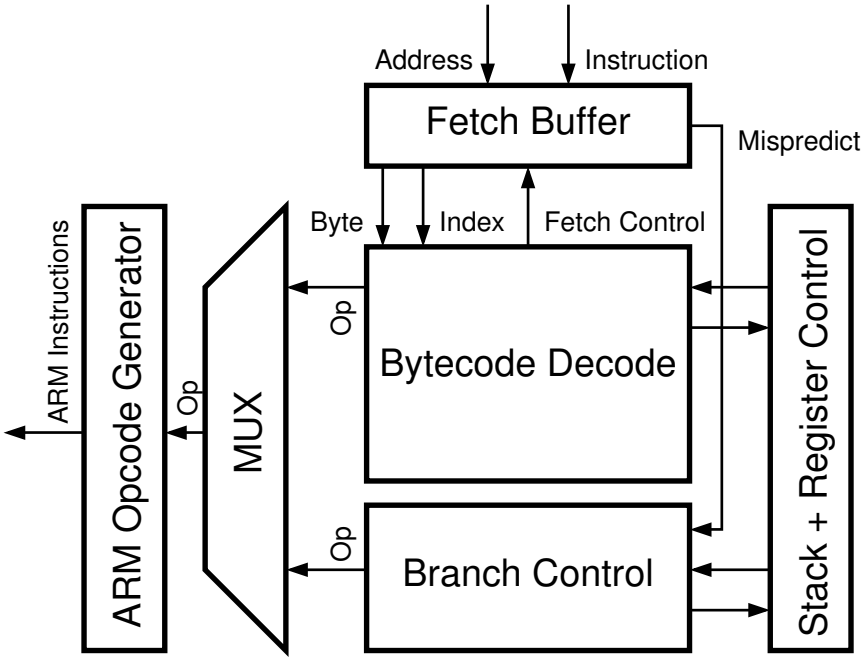


Figure 4.6: The Java Decoder Unit.

Figure 4.6 shows a decomposition of the decoder architecture into its main functional component blocks. Each unit communicates through asynchronous channels, with control originating at the fetch buffer and byte-code decoder. As soon as there are valid bytes to be processed in the fetch buffer, the decoder starts to generate appropriate instruction sequences, in a convenient internal format. These sequences are generated with correct register allocation through the stack and register control unit. Output is then fed to the ARM execute unit through the opcode generator. The branch control unit ensures consistency of the stack when possibly interpreting mispredicted instructions.

### 4.4.1 Fetch Buffer

The fetch buffer is present to latch a word from the fetch unit when in Java mode, and then dispatch individual byte-codes to the decode unit. The fetch buffer hides

the complexities of instruction fetch from the rest of the Java decoder, such as the fact that multi-byte Java byte-codes may run over word boundaries. The other main job this unit does is to keep track of where in a word byte-codes were initially fetched from, this information is needed as the ARM program counter can only be modified by the fetch stage of the processor pipeline. When fetching bytes not at the start of a word the PC will be incorrect and will need modification if a return address needs to be calculated. This situation occurs when jumping to a software byte-code handler as a `bxj` branch back to the subsequent byte-code must be issued at the end of such a routine.

The fetch buffer also has to inform the branch control unit if a branch has been taken, allowing for false state changes in the decoder to be corrected. Byte-codes in the shadow of a branch could potentially alter the stack state, even though the ARM instructions generated will not be executed if a previous branch is taken. The stack cache state will always be modified on instruction issue from the Java decoder, as the unit must assume instructions issued will be executed. The fetch buffer detects that a branch has been taken by looking at a data tag attached to each fetched word from the fetch unit, called instruction colour, this changes when a branch has been taken and the fetch unit has jumped to a non-sequential address. The fetch buffer can therefore only tell if a branch has been taken on word boundaries, this has a negative effect on branch latency as many byte-codes can be processed by the decoder in error on a Branch, a better system is really needed in a future architecture. An approach for branch optimisation is described in Chapter 6.

### 4.4.2 Stack and Register Control

To implement efficient allocation of registers within the stack cache, a circular buffer strategy is used. The state of the stack cache is maintained in this unit and is supplied to the instruction issuing units to provide correct register allocation. As a circular buffer strategy is used the state stored in this unit is: the position of the top of the stack (a register between R0 and R3, currently), and the occupancy. These must be updated accordingly when a stack operation is requested by the decode or branch stage.

When the stack cache is empty or full, and stack items are required for processing, extra ARM instructions are needed to manage the spill of fill respectively, as data processing instructions can only act on registers. The information on spill and fill is returned to the unit which requested a branch, the decoder unit is then responsible

for issuing the corrective instruction(s). When this happens, stack items are stored or taken from the main operand stack in memory. This means that care must be taken when dealing with software handler routines, as the stack in memory is not always up to date and stack cache state is not observable externally. Currently this is dealt with by flushing the stack cache when jumping to a software handler routine, although the stack cache state could be transferred by storing this state in a register when needed by software.

The only other concern in the stack and register control unit is that of restoring consistent stack state after mispredicted instructions have passed through the decoder. A mispredicted instruction in this case is one with incorrect colour meaning it has been fetched in the shadow of a branch; currently SPA only pre-fetches sequentially. In this event, when stack operations have been issued by the decoder the internal stack cache state will be modified, although instructions issued to the ARM decode stage will not be executed. To deal with this the branch control unit issues stack operations to restore the stack state on a branch mispredict. Currently the stack cache is emptied on a branch. This decision was made because the stack cache is usually empty on branches in typical Java code, as the operand stack is not typically used to continue calculations over branch boundaries.

### 4.4.3 Byte-code Decoder / Translator

This unit essentially performs the task of an interpreter loop. It takes bytes in a Java instruction stream one at a time from the fetch buffer, then decodes them such that instruction groups are identified. If the instruction is not handled then a branch is taken to an appropriate handler otherwise an instruction sequence is generated to for execution by the host processor.

The decode stage communicates with the fetch buffer, and is fed bytes and indexes for PC correction. The decode unit simply requests bytes one at a time, the fetch buffer taking care of word boundaries. The decoder can only ever request the next byte in sequence, this may be from a non-sequential address if a branch has taken place, which is why the branch control unit is required, to handle these cases.

Once the byte has been classified the appropriate instruction sequence is generated. Communication with the stack and register control unit allows for correct register indexes to be placed in instructions, correlating with the current stack cache state. If stack cache spill or fill operations are needed, this information is provided in the results returned from a request to the stack and register control unit. In such

an event loads or stores to the memory are generated.

Instructions generated by the byte-code decoder and branch control unit are stored and transmitted in an internal format. This allows for efficient implementation and a degree of portability between processors, either of the same architecture or with differing, but similar instruction sets. In this way, the Java decoder architecture should be portable to new architectures if necessary in the future.

This decoder structure is a very elegant solution, particularly in a self-timed processor. As instruction sequences are generated for frequently used byte-codes without any cycles wasted on decode or register maintenance many times speed up can be achieved over a software approach. However in this case when instruction streams are decoded from a single fetched word, many ARM instructions can be generated without the need for fetches from memory or cache. In an asynchronous implementation this will allow the remaining pipeline stages in the processor to run as fast as possible for the given instructions with no memory bottleneck. The fetch unit's operation should be absorbed by the time taken to issue multiple ARM instructions from the Java decoder. Even if the memory bottleneck is removed by this operation, it is not as good a situation as it seems, as no instruction folding takes place in the current architecture, more ARM instructions will be generated than are strictly necessary. The situation would be worse in a synchronous decoder as each unnecessary cycle will have a fixed worst case cost. Efficient folding mechanisms presented in Chapter 6 and 7 show how improvements can be made to code generation.

### 4.4.4 Branch Control

The branch control unit is required to deal with the effects of pipelining around branch instructions. If a conditional branch is issued by the decoder unit then subsequent byte-codes may be executed in error. These instructions in the shadow of a branch are caused by the fact the processor architecture pre-fetches instructions, and in the case of the Java decoder the fetch buffer can possibly contain up to three extra byte-codes in a single fetched word. In SPA, a branch will only be evaluated at the execute stage in the pipeline, the fetch unit will then be informed about the program counter change. This means a non-deterministic number of instructions may pass through in error if a branch is taken, due to an arbitrated choice at the fetch stage between fetching another sequential word and being interrupted by the execute stage.

Branches are detected through a colour tag bundled with each fetched instruction word. This information is added at the fetch stage and colour change happens when a branch is taken. The fact a colour change takes place is signalled by the fetch buffer with a mispredict message, sent with each fetched byte. If this signal is true then the branch control unit restores stack cache state to that when the branch was taken. Currently this means the stack cache is reset entirely. The stack cache is flushed, by issuing ARM memory operations before any branch is issued by the decoder block. Colour is checked at the execute unit, byte-codes are dispatched from the Java decoder with the same colour as the fetched word meaning only instructions with the same colour currently set in the execute unit will be processed, hence branches are correctly implemented.

Branch latency is a big issue with this system currently as colour change is only detected on each fetched word. This needs to be improved as tens of ARM instructions could be issued in error in the event of a mispredicted branch. The branch will not be detected until all these instructions have been issued to the decode and execute unit to be simply ignored. In a future version of the architecture a taken branch signal should really be sent quickly and directly from the execute unit to branch control to reduce the number of instructions issued in error. This mechanism was not added as the processor core was in development and modifications needed for the Java version were kept to a minimum. The current system would however be reasonable if branch prediction did take place in the fetch unit, although early forwarding of this information would be desirable, so it could take effect between word boundaries.

### 4.4.5 ARM Opcode Generator

This unit translates from the internal opcode format to ARM instruction op-codes. When the instructions are dispatched, necessary extra data fields such as instruction colour are added and sent to the ARM processor for decode and execution. This unit could be re-designed for other processor platforms, or for being embedded at a different stage in the pipeline, for instance generating decoded ARM instructions directly for the execute unit.

## 4.5 Balsa Implementation

Balsa is a high level synthesis tool and language developed within the AMULET group at the University of Manchester. The Balsa language allows for the description of hardware which communicates through asynchronous channels. This is high level in that details of the implementation are abstracted, lower level details are specified from a number of options when circuits are generated at synthesis time. The Balsa language is similar to imperative style programming languages such as C and Ada. Units of hardware are described as procedures; procedures can have local variable storage and can only communicate with other procedures through handshaking channels. These are a basic primitive of Balsa, along with constructs for sequential and parallel statement composition.

Balsa is used to synthesise large scale circuits and it works at a high level. It is not possible to specify very low level behaviour such as signalling protocols. Balsa has many features which support manageable and re-usable descriptions. Firstly the fact that hardware is specified at a high level means that hardware is not tied to a particular technology and can easily be integrated with different hardware styles in the future through re-synthesis. As channels are the only method of communication between hardware units, a clean interface must be provided to other units. Finally hardware can be parametric at compile time with a flexible system of instantiation, allowing even for recursive loops of instantiation making it possible to produce hardware libraries of highly re-usable and cleanly described system blocks.

### 4.5.1 Synthesis

Synthesis of the Balsa language is currently achieved through a purely syntax directed translation of the language to a data flow graph of communicating *handshake components* [53]. Handshake components are a set of macro-modules which implement key language functions such as loops, communication, variable storage and arithmetic. These components are all that is needed to implement a Balsa description, and must be implemented themselves in order to do so. These components communicate asynchronously through handshaking channels but can be built in any asynchronous design style. Handshake components can be implemented in schematic form or using lower level synthesis systems such as Minimalist [37] or Petrify [12]. Some component descriptions must be parametric in order to cope with arbitrary data widths and other such variables.

## 4.6 Implementation Structure

The Balsa implementation of the Java decoder architecture was mainly a problem of organising communication and storage within the unit's different components. As the design is to be asynchronous in implementation communication between blocks of hardware working in parallel must be explicitly synchronised through a handshaking channel, there is obviously no clock to do this job as in the synchronous world. In this sense, variables can not be shared between procedures or parallel blocks of statements in a description. Arbitration can easily be built in the form of a wrapper procedure, if such a resource is needed, thus the interface to a shared variable or resource would be encapsulated by the wrapper using arbitrated communications channels.

The design was partitioned as in the architectural description, although interfacing to the host processor and instantiation of the different procedures was done in the fetch buffer block. The procedures were described with the Balsa interfaces shown in Figure 4.7.

As can be seen from Figure 4.7 the implementation only consists of three separate procedures, this is less than the five blocks shown in the architectural block diagram in Figure 4.6. The procedures perform the following function:

`jumbodecode()` acts as the parent module, instantiating the rest of the decoder unit and interfacing with the host processor. As the interface with the host processor is established here it made sense to incorporate the fetch buffer to reduce the interfacing requirements with the rest of the decoder procedures.

`jumbo()` (Java University of Manchester Byte-code Optimiser) This procedure implements byte-code decode and instruction generation. The stack and register control is also implemented here as shared procedures local to the jumbo procedure as they can never be called from parallel blocks of code. Branch control is also situated here as this function happens only as bytes are fetched from the fetch buffer, also taking place as part of the decode step.

`armgen()` This is the ARM opcode generator and converts instructions from the internal format supplied by the `jumbo` procedure. The internal instruction format used is 30 bits in size, and is fully orthogonal, this allows implementation as a Balsa record type where each component can be constructed or translated in parallel. If ARM instructions were constructed internally, as well as

```
procedure jumbodecode
(
  -- the signals from the fetch unit
  input  i_Pc       : address ;       -- the instruction address
  input  i_Mode     : spaMode ;       -- the mode fetched under
  input  i_Colour   : colour ;
  input  i_Instr    : instruction ;   -- the instruction itself

  -- the signals to the arm decode unit
  output  o_Pc      : address ;       -- the instruction address
  output  o_Mode    : spaMode ;       -- the mode fetched under
  output  o_Colour  : colour ;
  output  o_Instr   : instruction     -- the instruction itself
) is
...

procedure jumbo (
  input  bytecode   : byte;
  input  byteindex  : 2 bits;         --offset relative to ARM PC
  input  mispredict : bit;
  output status     : jumbo_state;
  output arm_output : instruction
) is
...

procedure armgen (
  input  operation   : armgen_type;
  output instruction : instruction
) is
...
```

Figure 4.7: The Main Balsa Interfaces for Decoder.

reducing portability, construction would be much more difficult as different instruction types have different layouts making sharing of hardware at different points in the decoder (`jumbo`) block more difficult.

### 4.6.1 Communication and Integration

The `jumbodecode` procedure has the same interface as the SPA decode stage at both its inputs and outputs. This means that the procedure can be inserted into the SPA code at the top level when the pipeline is instantiated. Only minor modifications to the SPA were needed to facilitate Java acceleration. Balsa enabled many of these changes to be incorporated very efficiently, the abstract typing feature was particularly useful, for instance the Java mode could be added by changing the type for processor mode, this involved adding an extra bit, careful checks were needed on the usage of this information to make sure usage was consistent throughout the design though. Adding the `bxj` branch to Java instruction was the most work, but followed the template provided in the `bx` branch to Thumb instruction previously implemented by the SPA team.

The `jumbo` procedure requests single bytes at a time from the `bytecode` input channel, a byte index is supplied in parallel along with a mispredicted branch flag. The output of this unit is ARM instructions generated through the `armgen` unit instantiated internally, and status on the current state of decoding. This instruction passes out through the `jumbodecode` procedure to the host processor, the status information being used to determine whether another byte-code is required, an ARM instruction has been generated, or both. This instruction issue interface is quite subtle and required many iterations to achieve a satisfactory, working result.

`armgen` simply takes an internal instruction word on input `operation` and generates an ARM instruction, based on the instruction type on output channel `instruction`. This unit is connected directly back to the output of `jumbo`, the parent procedure. This procedure could also have been instantiated within the top level `jumbodecode` procedure, taking an internal format instruction from `jumbo`. The internal instruction format was designed to simplify the generation of ARM instructions in the decoder module and secondly to simplify extension of the decoder or the switch to another host processor.

## 4.7 Debug and Test Software

To test the Java decoder implementation, Balsa was used to create LARD output. LARD and the LARD2C simulation environments were used to simulate initially just the Java decoder within a test harness written in Balsa, and as the design progressed the entire JASPA core. As the simulation tools are still in development speed increases of an order of magnitude took place during development, making debugging much more practical at around 5 to 10 byte-codes being processed per second. Further speed increases were seen with a new simulator running with C implementations of handshake components being developed in the group, with an event driven scheduler optimised for asynchronous systems [28].

The software used to run a Java environment on the processor design was written by Ian Watson for the previously designed LARD Java decoder. Slight modifications were necessary for correct operation. This software takes a single static Java class file and produces ARM assembler format containing the byte-code and data associated with that class. The standard ARM assembler is used to assemble and link this code to a start-up routine and the software handler routines needed for non-hardware byte-codes. The resulting binary is then loaded into a memory model currently written as a part of a LARD test harness for SPA and now JASPA.

## 4.8 Circuit Implementation

JASPA has been synthesised with Balsa as a single-rail bundled data circuit and dual-rail QDI implementation. Balsa generates a gate-level Verilog netlist for simulation, which has been successfully utilised to verify the design and implementation. Although gate-level simulations do not give a totally accurate picture of how a silicon implementation would perform, they can be used to find problem areas in the design such as long critical paths of gates. Further work is needed in this area to establish problems with the current implementation, and identify good ways of expressing different primitives in Balsa.

As well as gate level simulation and implementation, a dual-rail design, but just of the decoder block, has been taken through place and route to silicon layout. This layout has been through an RC extraction process to find physical resistance and capacitance characteristics of the implementation and this information can be used to make the best guess at how a real implementation on a given technology will

perform. Simulation results are given in the following section.

As far as circuit area is concerned the current design synthesises to 45,000 transistors in single-rail technology, and around 90,000 using a dual-rail technology mapping. These results use a cell library developed within the AMULET group in Manchester targeting ST Microelectronics' 0.18 micron process technology. A plot of the layout is shown in Figure 4.8.

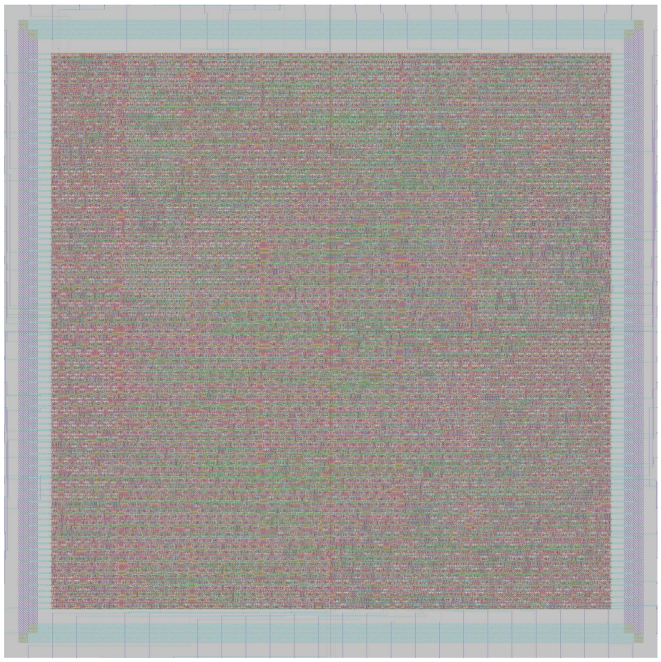

Figure 4.8: JASPA Standard Cell Layout.

## 4.9 Simulation Results

The following section presents a breakdown of where performance increases are achieved by the hardware, in comparison to a minimal threaded software interpreter. Such an embedded interpreter has similar memory requirements to the hardware solution, but with additional handler code for each byte-code handled in hardware by JASPA. This analysis is broken down into two sections: one dealing with a reduction in RISC execution cycles, and the second looking at benefits gained through the use of an elastic, self-timed, pipeline.

A more detailed breakdown of performance is given in Chapters 6 and 7 comparing the simple byte-code translation scheme with more optimised approaches.

### 4.9.1 Code Generation

When executing byte-codes in hardware stack management is handled internally. A software interpreter must either manage such a cache with extra state maintaining code, or use a stack stored in memory. Further to this, RISC code dispatched by the Java decoder does not have to be fetched from memory, hence will not pollute the cache and reduces memory traffic.

Figure 4.9 (from [10]), shows comparative performance in terms of the number of RISC executions needed for a selection of byte-codes, ranging from the most efficient, to the most problematic (goto). The software routines use main memory for the operand stack, removing problems with state management. Instruction counts shown in black are for best case timings, while the grey bars indicate the worst case timing. The hardware timings often have poor worst case timings as there is the possibility of stack cache spill and fill. The only case where this is worse than software is for goto, when the stack cache must be flushed; this involves up to four memory stores.

In reality, worst case timings are very rarely incurred. Importantly, the Sun Java compiler tends to minimise operand stack depth for a given expression. In common examples this will fit in the stack cache of four registers, or will require few extra memory operations. As far as the goto byte-code is concerned, using Sun's compiler we have never experienced the need for a stack cache flush as it has always been emptied by preceding code.

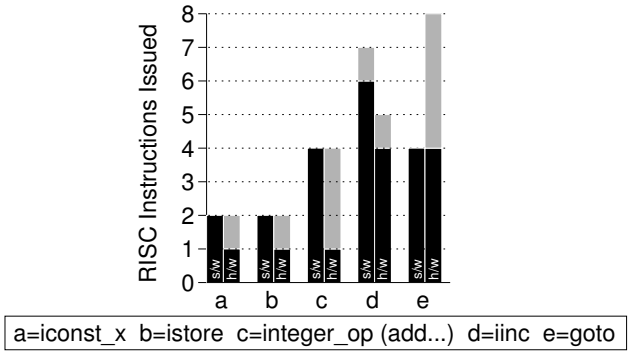


Figure 4.9: Code Generation Comparison.

A summary of overall performance, for the previous byte-codes, including the handler dispatch overhead is shown in Figure 4.10. In practice instruction sequences

generated while executing simple arithmetic benchmarks resulted in typically a factor of 4 speed increase over interpretation. Including the effects of memory accesses a factor 7 improvement has been observed with some Java code.

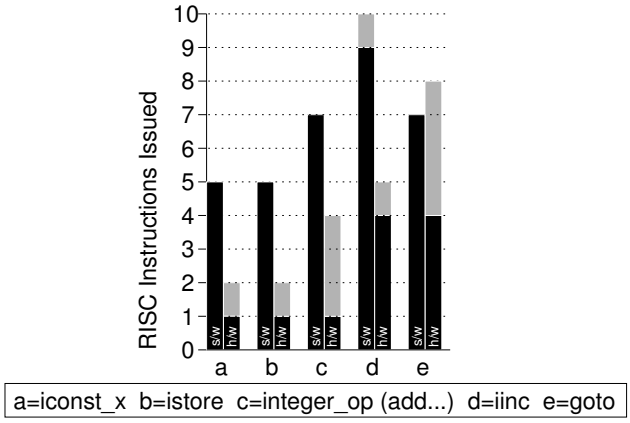


Figure 4.10: Cumulative Difference.

Figure 4.10 shows a big difference in instructions issued between the implementations. This is because the interpreter loop must fetch the byte-code from memory in order to decode it and then branch to the correct handler routine; these branches may introduce additional overheads, although it is possible to append the *next instruction fetch to* every handler, which removes one branch.

### 4.9.2 Timings

The elastic pipeline latency inherent in the self-timed design allows simple operations to complete faster than more complex ones. When little processing is required to translate a byte-code then the time taken to produce the associated RISC instructions will be reduced. The main example observed during simulation was the difference between operations requiring many operand stack operations and those which do not. A difference is also experienced between when a fetch is required to the word buffer, and when bytes are already available. The best possible timing for each byte-code is achieved therefore resulting in overall average case performance. Even with this simple scheme there is a big difference between fast and slow translations, this would make a synchronous design either more complicated or globally slower.

Simulation results for the self-timed Java decoder were run using Nanosim, on a Spice netlist extracted from silicon layout in Cadence. The Java decoder was

simulated in isolation to remove bottlenecks present in SPA and discover the best possible performance of the unit.

Averaged timings over 100 repeated byte-code decode cycles showed a variation in latency between 30ns and 99ns per issued RISC instruction. Repeated loads, stores and arithmetic operations were used to time basic operations. Timing is apparently mainly dependent on the stack management requirements of each byte-code. Fetch latency seemingly has little impact, but was hard to test in isolation, and may have more effect when simulated with the SPA design. When RISC output was part of a sequence, latency hit the lower bound of 30ns, along with byte-codes such as `iconst_x` (34ns). Surprisingly, simple byte-codes such as `iadd` took up to 99ns as usually a single RISC operation was issued, but after 3 internal stack cache checks. The decision to implement serial stack operations was made to reduce gate count and improve power figures in the average case. A parallel implementation may improve performance in some cases although the complexity of such a unit would potentially be too great a cost due to the large state space involved in up to 3 simultaneous stack operations. The parallel approach may be the only option in a clocked design.

Unfortunately the Java decoder unit's absolute performance was very slow, considering the 0.18 micron process used. However, it is faster than the SPA ARM execute unit, and would suit usage in a secure smart card environment. Part of the performance problem is related to the secure circuit style used, and mostly down to the wholly non-hierarchical one pass place and route flow. No time could be afforded for timing optimisation at the layout stage. Importantly it was shown that simple byte-codes could produce RISC output 3 times faster than in more demanding cases.

## 4.10 Summary

The architecture presented in this chapter was designed with the aim of being simple, efficient and simple to build. The result although not performing any complex optimisation strategies is designed to be effective at accelerating Java by a factor of 4 and upward over a software interpreter. The cost of this speed up will be quite minimal in terms of design effort and actual hardware, as a minimal approach has been taken with the aim of assessing the impact of different features in the design. Although simple, extension of the architecture is well provided for with the modular approach taken in the design.

The self-timed nature of the architecture is hoped to provide additional increases in performance while not causing any complications in the architecture. Average case execution times provide the performance, but also due to simplicity and asynchronous circuit operation low power, good EMC and security should also be key beneficial features. More complex decoders described in Chapters 6 and 7 take further advantage of self-timed design to allow infrequent but more complex optimisation steps to happen as part of the decoder's operation, without having a negative effect on the clean design and cycle time of simpler operations. The fact the decoder can issue instructions for Java through the processor with reduced memory accesses possibly very rapidly is also an attractive feature.

The JASPA processor implementation has been presented detailing the small and effective Java decoder unit acting as an extra pipeline stage feeding instruction sequences to the host SPA, ARM compatible host architecture. Design goals and philosophy have been described, allowing for the design and construction of a prototype system. The translation system and host processor architecture have been described. Advantages over software interpretation have been made clear, along with a detailed description of the decoder architecture. Self-timed design and implementation have been taken into account and have been put across as an important feature of the architecture, resulting in good performance, simplicity and implementability.

The reason for implementing the architecture is to show the advantages of the self-timed design. Previously a similar system had been modelled in LARD [16] by Ian Watson; LARD however only models the behavioural characteristics of such a design. In order to see how the system would perform an implementation can be simulated at many different levels, all the way down to layout with capacitance effects. This option was much more accurate than writing a simulator or model, although presented problems with analysing performance in detail. Balsa - a high level synthesis system for asynchronous circuits made the implementation possible. Balsa provides a rapid development route through a well structured high level language, simulator and test interface. The problem however being the difficulty assessing the detail in timing of the design, requiring an implementation down to silicon layout (or at least gate level), in order to extract timing information.

#### Basis for Further Work

While much has been learnt from the hardware implementation presented, unfortunately describing hardware for future designs, would consume an excessive amount of time and resources. This realisation lead to the construction of a simulation environment, described in chapter 5, designed explicitly to profile the execution of such processor extensions. This approach allows for a wide variety of parameters to be varied across the architecture without committing to time consuming low level designs. Flexibility in the timing of different operations can also be experimented with.

It is unclear from this work, whether in the 15 general purpose ARM registers will be enough to explore more efficient mapping strategies, incorporating forms of instruction folding. One of the aims of this work is to look at sequences of byte-codes to determine an optimised strategy for execution. This issue is discussed further in Chapters 6 and 7.

### 4.10.1 Conclusions

A Balsa implementation of the Java decoder architecture has been completed, this has been implemented along side the SPA ARM core to form JASPA [10]. Although further analysis is needed to assess performance issues, the design, implementation and integration took only 6 to 8 months to complete showing a positive side to using Balsa as an implementation strategy. Potential problems with branch latency have been identified but on balance the design has been successful in meeting its aims of simplicity, efficiency and being quickly implementable. Future work will assess the detail behind the performance levels achieved, although there remains much scope for improvement even in this implementation as more is learnt through the experience. As it stands the implementation here is a good framework for future decoders employing novel techniques for efficient embedded Java processing.

# Chapter 5

# Architectural Simulation

Detailed simulation, data collection and analysis is needed to understand how different architectures will perform when executing Java code. It is not necessarily even sufficient to be able to run benchmark programs with approximate timing information, as this will not always give insight into where the architecture is succeeding or failing.

In order to design an appropriate, efficient scheme for supporting the execution of Java code in a RISC pipeline it was decided that a higher level of abstraction was needed to allow rapid prototyping of designs. When implementing the Java decoder described in chapter 4 a large proportion of time was spent dealing with low level implementation issues, rather than on the architectural details necessary for improving performance. If behavioural modelling was used, rather than implementing the system down to the gate level using a synthesis too,l many more design alternatives could be constructed. As well as more rapid prototyping, simulation performance would be much greater as fewer low level, implementation dependent, communications would need to be modelled.

Higher level simulation, while providing a means to evaluate new architectures, must also make valid assumptions about issues which will come into play at the implementation level. This is especially important with regards to system timing, assigning realistic values based on correct assumptions or real data.

The circuit level implementation of JASPA provides concrete timing information based on characterised timings of physical silicon gates. It is important to realise that no information on system performance was available until the circuit had been synthesised. This makes the design process very difficult, and has also proved a problem in the design of the host SPA processor. Modelling at a higher level allows

for performance information to be available at all points in the systems design, even if this information is only based on assumptions, it can be refined over time with additional experiments and characterisation against implemented circuits. Moreover, integrated, structured systems of data collection and analysis can be embedded in higher level languages used for behavioural simulation, giving additional feedback on how the design will perform.

Higher level languages provide the additional advantage of having thorough typing systems, and can provide for extension of types, or even object orientated modelling of system components. These features greatly control the problems associated with managing complexity during implementation. A further benefit of using such language features is the extent to which designs can then be made configurable for the evaluation of variations in architecture. Radically different alternative designs for sections of the architecture can also be cleanly instantiated and tested under a fixed framework with minimal interfacing effort.

The architectural level simulator, implemented for evaluation of hardware support for Java, was eventually constructed in the Ada language. The reasons for this were primarily due to the features for programming in the large: modules, tasks and data types. Importantly run-time performance and communication and scheduling of parallel threads allowed for an efficient timed model of the asynchronous processor pipeline required. A more detailed description of why this approach was chosen over using more established hardware description languages, or more recent system level modelling tools is given in section 5.3 of this chapter.

### 5.0.2 Chapter Overview

This Chapter provides the rationale behind the construction of an architectural level simulation environment. The following sections describe the detail behind choosing this approach and an implementation strategy.

#### 5.0.2.1 Requirements

This section gives a detailed description of the requirements of the simulation system. Many of these requirements justify the construction of the simulator. The two main areas of focus are: the ability to gather conclusive results about candidate architectures, in terms of timing and implementation/performance issues.

#### 5.0.2.2 Implementation Options

Many options were available with respect to the implementation of the simulation environment, each able to provide the basic requirements, each with different positive and negative qualities. This section describes alternative approaches and how Ada was chosen to implement the system as a software system level model.

#### 5.0.2.3 Simulation System

This section describes the detailed working of the simulation system used to evaluate embedded Java processor pipelines. A simplified event driven simulation is described, designed to model a processor pipeline using Ada tasks, without the need for an explicit event queue. An explanation is given of how the system meets the requirements set out at the start of the chapter.

#### 5.0.2.4 System Performance

An analysis of run-time performance of the simulation engine is undertaken. This shows the performance of the simulator on a variety of computer platforms. Particular attention is payed to the optimisations incorporated into the simulator including its inherent multi-threaded design.

#### 5.0.2.5 Summary

Summarises the chapter, making conclusions about the benefits of the resulting simulation system.

## 5.1 Requirements

In order to compare different approaches to the acceleration of Java execution within a processor pipeline, an efficient approach to the problem was required. Many lessons were learnt from the implementation of a JASPA. Not only were ideas generated for the improvement of decoder design, but many issues were raised at the implementation and evaluation level. There were several key concerns at this stage:

**Implementation time:** The combination of using the Balsa synthesis system and the pre-existing SPA core brought about the comparatively rapid development of an asynchronous Java enabled processor. Unfortunately, a large proportion

of the development time was spent dealing with low level hardware design issues, resulting in a rather fixed and rigid implementation. As almost all of the planned improvements were at the code translation and register allocation level, a more efficient system of evaluation was required. Flexibility and a reduction in implementation time is of great importance.

**Implementation quality:** One of the drawbacks with the JASPA implementation was the absolute performance, being very slow (around 10 to 20MHz cycle rate). The Java part of the pipeline was being held back by the surrounding RISC processor. Making modifications to this would have been prohibitive in terms of time. Taking a higher level approach would remove this type of problem, as control would be gained over the whole architecture. This would allow different scenarios concerning other parts of the processor pipeline to be tested, at a relatively small cost.

**Simulation performance:** The time taken to gather simulation results from layout or even gate level simulations was costly when developing JASPA. This situation had to be improved, in order to simulate representative benchmarks, and allow differences to be seen between alternative architectures.

**Fairness of comparison:** In order to compare different Java decoders, a fair system of testing was required. This would have been difficult to achieve when designing each alternative in Balsa, as changes in implementation structure can alter performance figures significantly. Additional variables such as variation in layout provide further room for inaccuracy. When evaluating new decoding styles, it is important not to introduce artificial barriers to performance. A higher level model would factor out some of the low level implementation differences which might well alter performance depending on time spent at the implementation stage of each design. It is important to impose realistic assumptions on the high level models to retain fairness, these assumptions can at least be made with transparency.

### 5.1.1 System Level Simulation

After considering the issues explained above, the primary requirement of the simulation system is run-time performance. Work on improving the Java acceleration system will be without purpose if it is not possible to run sufficient code to make

conclusions about different architectures strengths and weaknesses. There must however be enough detail modelled to allow for the results to be useful and valid.

Secondary therefore to the performance concern, but also of high priority, is modelling accuracy. Timing must be modelled at an arbitrary level of detail, as required by each design. A level of configuration will be required to support the analysis and evaluation of any assumptions made about timing. Simply counting the number of RISC instructions generated for a particular Java program run is not sufficient. It is a key requirement of the simulator to provide the ability to model the processor pipeline as a synchronous, clocked system, and also as an asynchronous, self-timed design. This must be tested as it is a key idea behind all of the improved Java decoders described in this thesis.

The two primary issues identified above are related to the run-time simulation engine. The simulation environment must also cater for the rapid implementation of the Java enabled processor pipeline model, and the different decoder alternatives. Modelling at a higher level will provide a large advantage over having to describe low level implementation details as with JASPA.

Other implementation requirements are practical concerns, and relate to the effort required to implement the final system. The primary point here is implementation time must be minimised by choosing an implementation strategy which can cope well with the functional requirements of the simulator.

### 5.1.2 Requirement Summary

#### 5.1.2.1 Performance

The simulation environment must be capable of allowing the execution of the order of millions of Java byte-codes as part of the analysis of different architectures. The aim being the realistic evaluation of each system, without imposing restrictive limitations on what code can be run on the processor models.

#### 5.1.2.2 System Level Modelling

The simulation environment must be able to model synchronous and asynchronous processor pipelines, with delays associated with specific operations within each pipeline stage. This will allow models to reflect the behaviour of circuit level implementations at a reasonable level of accuracy. Delays from existing implementations

such as JASPA or Amulet 3 could be used to calibrate such models to provide further grounding in reality.

#### 5.1.2.3 Configurability of Delays

The delays within different system level models should be configurable. With the ability to customise delays for different operations within pipeline stages, many important scenarios can be simulated. The impact of different timing assumptions can be tested and quantified for transparency of results. This feature should also allow for a degree of calibration against real circuits, such as JASPA, or existing processor pipelines.

#### 5.1.2.4 Pipeline timing

The simulation system must be able to model both clocked and self-timed pipelines while maximising component re-use. This relates to the previous point about delays. If delays can be controlled enough at each pipeline stage, synchronous versions of self-timed models should be simple to synthesise. This would be done by adding a fixed worst case delay and removing all data dependent delays within the component.

#### 5.1.2.5 Configurability of Components

It is important for the simulated processor pipeline to be configurable where required. This will allow for evaluation of performance across a range of implementation options. Examples include the use of parameterisable buffer sizes, bus widths and the size of the register file.

#### 5.1.2.6 Profiling

One of the most important requirements of the simulation environment and pipeline models, is its ability to be profiled in detail. The design and simulation of novel Java aware processor pipelines will only be purposeful if conclusions can be made about how the systems perform. The system should allow for the profiling of different instructions and sequences of instructions for the duration of a program's execution. This will facilitate the analysis of how proposed systems are performing in a realistic environment, while providing a detailed breakdown of positive and negative qualities of an architecture.

## 5.2 Implementation options

An important practical decision regarding the simulation system had to be made, regarding the implementation environment. This decision goes hand in hand with the simulation techniques used to model the proposed new architectures. There are many environments and languages suitable for the modelling the type of hardware under consideration in this thesis. The main tradeoff is between providing the required level of modelling detail in terms of timing and signalling, while giving flexibility, simulation speed and simplicity. The requirements detailed are already above the low level modelling usually associated with traditional hardware description languages. Modelling at a higher level in custom software is an accepted alternative and could be a good choice here.

Traditionally, in synchronous hardware design, HDL's have been used to design behavioural, and synthesisable register transfer level designs. Behavioural level code simulating much faster, and allowing rapid development without concern for encoding of all state and sequencing in hardware. Both approaches require a full event driven simulation model. Taking a higher level approach has become more common in processor design, as people need to run as much code as possible on new designs in order to evaluate possible improvements on realistic application code. Design of the software tool set and application code can also begin before hardware is available. This works well in the synchronous world as cycle by cycle simulations can be constructed in many languages, without the need for complex simulation models. Indeed models can be constructed to an arbitrary level of detail, a good example is the AMD K6-2 microprocessor, modelled and formally verified using C during its development [43]. One problem with modelling in software is the lack of arbitrary precision types often useful when modelling large registers, busses and signal values.

The remainder of this section describes alternatives considered for implementation of the simulation environment, used for the evaluation of Java aware processor architectures in the remainder of this thesis. A brief description of each alternative is given, along with rationale for its usage. Finally the reasoning behind the use of Ada for the construction of the final solution is given.

### 5.2.1 Hardware Description Languages

#### 5.2.1.1 VHDL

VHDL is a high level HDL, based on the Ada language, originating from the US military [25]. This language facilitates the description of arbitrary hardware, not restricted to clocked models, allowing for sensitivity of parallel units of hardware to changes in signals on inputs. At its lowest level, it allows for the description of networks of components, to be instanced in a netlist. VHDL has the advantage of allowing declaration of types to simplify higher level modelling and improve code clarity.

#### 5.2.1.2 Verilog

Verilog [26], provides similar features to VHDL, with a more minimal C like syntax. Verilog does not have as complete a type system as VHDL, although it is being extended with features such as interfaces in the upcoming SystemVerilog standard [1]. SystemVerilog was not known about at the time the choice was made, and tool support is minimal even at the end of the project.

#### 5.2.1.3 SystemC

SystemC [27] is an attempt at providing system level modelling and high performance simulation using standard compilers and development software. The advantage being that systems can be modelled at a very high level using existing software libraries and achieve rapid simulation performance across multiple computing platforms. SystemC provides a simulation kernel library, provided as source code, which allows for management of time and parallelism in simulation models. Waveform dumping, arbitrary precision logic and arithmetic types and other features of typical Verilog and VHDL environments are available, while supporting high level features for *programming in the large* through C++. SystemC has evolved during the course of this project to a level where it would now be considered for the implementation. Version 2.1 brings abstract channels, simplified tasking and more object orientated design features making it an ideal choice.

There are problems with code clarity (mainly in thread control and synchronisation, still present in later versions) and simulation kernel performance. SystemC is still more suited for cycle based synchronous models, where it would bring the benefit of a standard approach to the problem, where in the past bespoke solutions

were the norm. EDA software can also support integration of such models into lower level tool flows.

#### 5.2.1.4 LARD

Lard [16, 21] is a language and simulation kernel designed specifically for modelling asynchronous circuits and systems. While providing the abstractions necessary for investigating such designs as Amulet3 [20], clarity, modularity and support libraries are limited. During investigation, support was being withdrawn and simulation performance was very poor compared to the alternative choices.

#### 5.2.1.5 Balsa

While Balsa [6, 7] was used to implement the JASPA hardware, it could also be used to test alternative configurations and improvements to the architecture. In order to do this, some higher level of timing would be needed, as synthesis and gate level simulation would add many variables and cost dearly in time. The main problem here was the performance of the simulator at the time of development. The simulations ran through LARD, which while improving only provided under ten simulated instructions per second on the SPA/JASPA test suite. During the course of this research much work has been done to improve the situation [29, 28], making it a much better system for this kind of design/prototyping activity.

### 5.2.2 Software Models and Programming Languages

In order to model micro-architectural features in software, a simulation environment must be constructed. There are many approaches to this problem and different compromises can be made depending on the level of timing detail required. When modelling clocked units only, a cycle based model can be made, assuming no communication between clock events. In such a model, each modelled unit can be run in turn each cycle. Inter-module communication is biffered between clock events. When modelling asynchronous modules, some form of discrete event simulation engine is needed.

The main languages considered for building a software model were the following:

#### 5.2.2.1 C

Low level language, providing a very minimal run-time environment, suitable for maximum speed. Does not support function overloading or object orientated programming directly. Generally complex to work with required features such as threading and large (over 64 bit) integer types.

#### 5.2.2.2 C++

Similar high efficiency to C, but with object orientated features enabling better clarity and extensibility of models. SystemC is an example of how a generalised modelling environment can be built to exploit the flexibility of such a language.

#### 5.2.2.3 Java

Java [32] could equally be used to model hardware, allowing extensibility through its features for object orientated programming. Threading is handled in a standard way allowing transparent portability between platforms. As with the other languages mentioned so far, abstractions for asynchronous communications channels and synchronisation between modelled units would have to be built from existing basic primitives in the language.

#### 5.2.2.4 Ada

As with other high level languages considered, Ada [39] provides structures for data abstraction. Also as with C++ and Java it supports generic programming and object orientation. The advantage Ada brings is its mature support for threading, or *tasking* in Ada terminology. Support for messaging and remote procedure calls allow for a good mapping to asynchronous communication channels, used in the hardware being modelled here. Especially of interest is the *rendezvous* [24] which has similar behaviour to that of a four phase handshake.

### 5.2.3 Implementation

The decision was made to use Ada to make a high level software model of the Java processor pipelines being considered in this thesis. In terms of the requirements set out it was the best match for the following reasons:

- Performance: Similar performance to other compiled languages such as C can be achieved with Ada, although it has the additional overhead of run-time checks associated with ranges and tagged types. Combined with an appropriate simulation model, Ada can provide good implementation performance.
- System Modelling: Ada provides language features such as synchronised channels and entries which can be used to directly model system level communications behaviour.
- Pipeline Timing: Simulation time can be modelled centrally in a distributed fashion, abstracted in appropriate data structures, allowing modelling of abstract synchronous and asynchronous designs.
- Delay Configuration: High level configuration can be read from file and encapsulated in abstracted form.
- Configurability and Commonality: Tagged, extensible types, tasks and modules allow for a flexible implementation, allowing re-use between different architectural models.
- Profiling: Profiling can be integrated into the simulation system. The system can be abstracted away from the simulation models in a higher level module.

## 5.3 Simulation System

The following section describes the structure and implementation of the simulation system built to model Java enabled processor pipelines. The aim being to find a good timing model, while achieving improved performance over a full blown discrete event simulation, which would be more sensibly constructed in an existing HDL. Key to the approach taken is the simplification of modelling relatively short processor pipeline structures. Additional benefits of using the language features of Ada, in the modelling process, are discussed.

### 5.3.1 Modelling Problem

Figure 5.1 shows the main components of an example high-level processor pipeline model. The granularity of the model is the pipeline stage, which allows the parallel behaviour of modern processors to be captured simply, with each pipeline stage

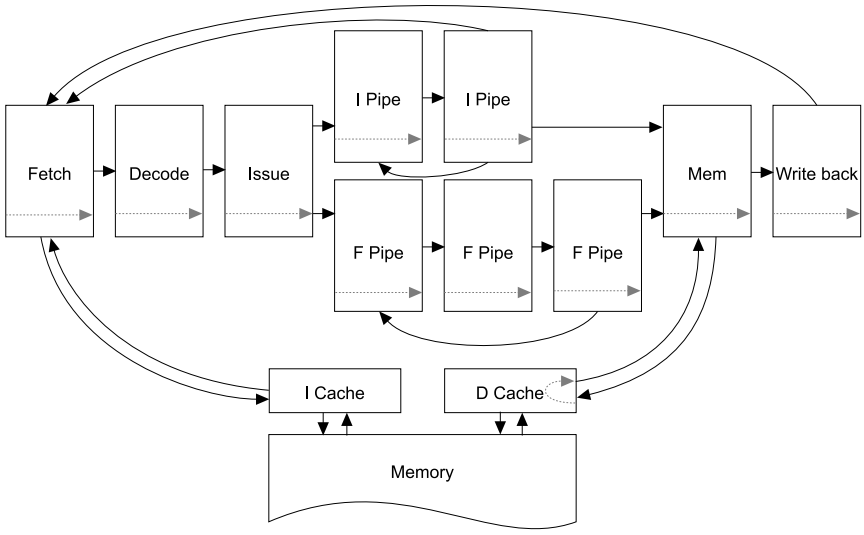


Figure 5.1: Example Processor Pipeline Model.

being modelled as a separate process, or thread. Each pipeline stage will process the incoming data, and when complete, pass the results on to the next stage, so all pipeline stages are processing data in parallel. Note that this example is illustrative and an actual system would probably have many more forwarding paths.

As far as timing is concerned, in a clocked design, all units will accept communication on the clock edge, in a synchronised fashion. In a self-timed system, pipeline stages will forward processed data after the time taken for each individual input data pattern, which may vary between cycles. The data will be read by the receiving unit, when it is has finished processing its data. In the self-timed model an acknowledge signal is required to tell the sender when the target has read the data.

The black arrows between pipeline stages in Figure 5.1 show data flow. An important detail to be noted is that the majority of communications are between neighbouring pipeline stages, in a single direction. These pipeline stages are accepting a single input from the previous stage. In a self-timed system, this allows for a simplified modelling approach. The main points of interest are where a pipeline has a choice between possible inputs. Where there is a choice, and both inputs may not necessarily arrive together, arbitration or synchronisation is usually necessary. This is an issue in both the approach to simulation and timing and also in any implementation of the pipeline. Timing is the most important issue, as the ordering of events taking place in the simulation must be preserved to reflect correct behaviour

of the design.

Processing time delays are shown as dotted arrows inside the simulation level modules. In a synchronous pipeline, the system is clocked at the speed of the slowest stage in the pipeline, in the worst possible data dependent case. All units thus complete their computation and communicate in unison at each clock event. In a self-timed model, these delays may vary in order to maximise performance for any given instuction/data stream. Key to simplifying the simulation of time in this case is the lack of external dependencies at each stage in the pipeline. When processing data, a module need only know at what time it arrived, then add on processing time for that stage. Each stage behaves deterministically in terms of function and timing, so no central control of time is needed. The only exception is a point of data dependent feedback, this would typically be in the branching mechanism of a processor, if nowhere else. The simulation kernel described below attempts to take advantage of this observation.

### 5.3.2 Simulation Units

To simplify the simulation system as much as possible and maximise the benefits of using Ada, many units in the simulation mode map directly onto language primitives. Figure 5.2 shows these mappings, in terms of the pipeline structure illustrated in Figure 5.1. This mapping was chosen to minimise the run-time overhead of the simulator, while maintaining a clear implementation. Simulation level modules, which are always pipeline stages, are distinct Ada tasks, which run in parallel. Handshaking communication channels are handled by protected entry calls, which allow for blocking and synchronisation of modules. Entries also provide input selection and arbitration, when stages accept multiple inputs.

A non-deterministic (as far as the receiver is concerned) communication is also shown in the simplified pipeline model of Figure 5.2. This represents a communication which may happen arbitrarily, affecting the behaviour of the receiver. The order in which such events happen in the system must be maintained in the simulation, in order to accurately model the pipeline. The task which receives such a message, or is waiting to, must only progress when it is known that no interfering preceding event could possibly be generated.

If multiple processors are available, this simulation model will run across them so long as multiple units have work to do at a given time. The communications between threads are minimised and occur only with data communications in the

simulated system. The fine grained nature of the pipeline stage models may not however make the best of this possibility, due to latencies involved in synchronisation between processors. When running on a multi-threaded single processor, a distinct performance advantage should be seen, thread switching should be very cheap on such systems.

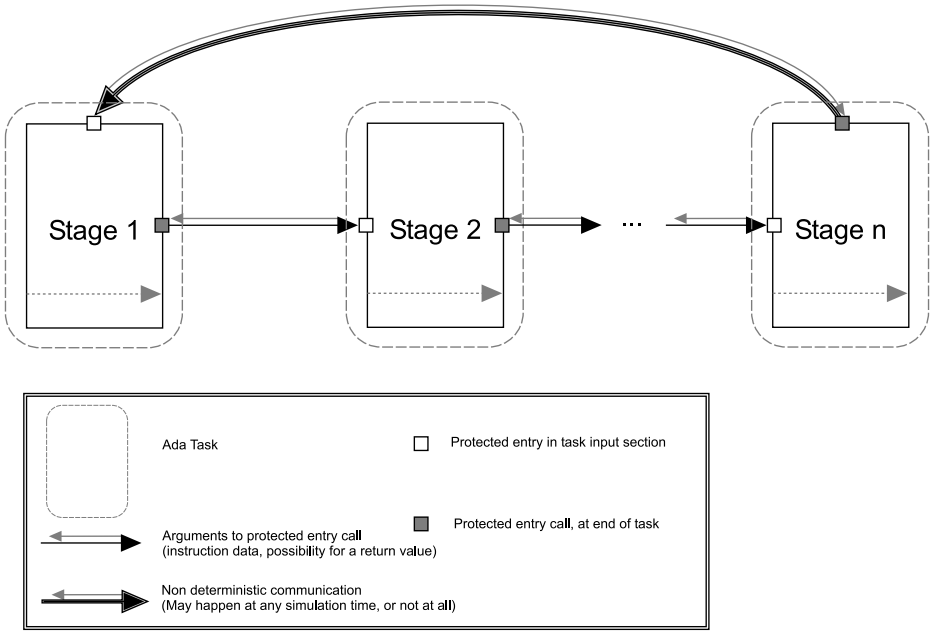


Figure 5.2: Modelling a Pipeline Using Ada.

While the mapping to Ada language features provides an efficient implementation for hardware modelling, one required feature present in Figure 5.1 is missing: a model of system timing.

### 5.3.3 Simulation Kernel

Modelling systems involving a set of communicating processes are typically dealt with using *discrete event simulation*. This simplifies the simulation by modelling time as a discrete quantity, physical processes being simulated by logical processes in the simulation environment, communicating via events. Events occurring at an equal time stamp are resolved as required by the simulator. The *simulation kernel* manages the progression time and events in the system. The traditional way to manage time is to have a central event priority queue, where events are taken off

and processed in the order in which they occur. This guarantees correct simulated behaviour [36].

#### 5.3.3.1 Time-warping

Jefferson introduced the concept of virtual time [30, 13] in order to improve the progress of simulation in the best case. This approach complicates the simulation kernel with the notion of *Time Warping*. In such a simulation, at points of non-determinism, such as when waiting for a possible input to arrive, a check-point is stored encapsulating the current state of the simulation. This allows the simulation to proceed speculatively, without switching or waiting on another process. If an event does occur later in the simulation which interacts with a process which has been run speculatively, simulation is rolled back to the point of this event, with the new event having its proper effect. This approach was deemed to over complicate the simulation engine within this project, designed for ease of instrumentation with profiling code.

#### 5.3.3.2 Distributed Discrete Event Simulation

A more appropriate optimisation for the simulation of pipelines is inspired by the use of local clocks [31] to order events. This brings about a system of *distributed discrete event simulation*. There are many ways of achieving a correct ordering of events and sufficient progression, summarised by Misra in [36]. Deadlock avoidance in the kernel and correctness of simulation become a problem when using this modelling approach. In a distributed simulation environment, each logical process is modelled as a task, as in Figure 5.2, with local communications localised to the routes of data flow, as desired for more optimal performance and transparency of implementation.

Two necessary properties of discrete event simulation describe the problems of moving to a distributed implementation:

**Predictability** The output of every process can be computed up to any time, given an initial state.

**Realisability** No process can guess what messages it will receive in the future.

When the central event queue is removed, any point of uncertainty, where communications are being waited on must be resolved.

In the type of simple embedded processor pipeline proposed for simulation here, there will only be a single point of uncertainty, waiting on a possible branch feedback event. The most simple method of dealing with this situation is the use of *null* messages. Alternatives such as [51, 52] have been proposed, but require complications of simulation models to look ahead to determine expected feedback. A null message establishes the absence of a message, so the receiver can then progress up to the message time stamp. This is how the simulation of the processor pipelines was managed in a distributed and efficient fashion. If care is not taken with null message generation, deadlock can still occur, its avoidance is described below.

### 5.3.4 Simulation Timing Model

A distributed simulation 'kernel' was implemented to model Java enabled processor pipelines. Taking advantage of the linear nature of the pipeline, the following structures were used:

- Pipeline stages are modelled as individual Ada tasks, which map to OS threads
- Communications, and events, are modelled with blocking protected entry calls, without busy wait loops.
- Timing is managed by tagging messages and acknowledgements with local times from the sender and receiver. When a message is received, the local clock is updated as the maximum of the incoming time and the current local value. This time is returned to the sender, so its local clock can be updated similarly.
- Deadlock prevention is achieved using null messages. The only unit needing to do this is the execute unit when signalling a branch. When no branch occurs, a null message is sent to the fetch stage. The fetch unit can then proceed with activities up until the time stamp on the null message.

#### 5.3.4.1 Deadlock prevention

In order to prevent deadlock of the simulator, null messages are used in the branch feedback path of the processor pipeline. Figure 5.3 shows the feedback path as modelled in the processor simulation system.

The optimised processor model in the final implementation has only one point of non-determinism, which is at the fetch unit, shown at the left of Figure 5.3. The

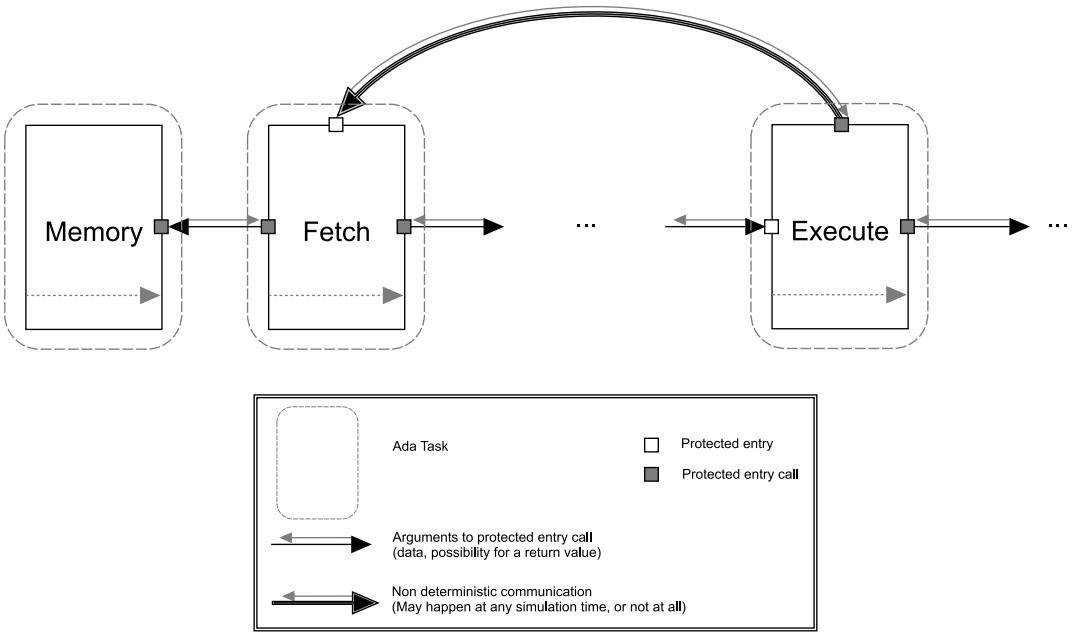


Figure 5.3: Deadlock Prevention in Pipeline Model.

situation is simplified further due to the behaviour of the fetch unit and the fact that there are no other incoming communications apart from the branch feedback information from the execute stage.

In order to make deadlock impossible in such a model, with only a single point of feedback and indeed non-determinism, a single condition must be met at the fetch stage. In order to guarantee progression, there must be an event, null or otherwise, occurring on the feedback path with a time stamp in advance of the next scheduled memory access. This enables fetches to occur up to the time stamp of the null message, or a correctly timed branch to be acted upon, if appropriate.

This condition is guaranteed by taking the following actions:

1. The pipeline is initialised such that at startup a null message is sent on the feedback path.

2. The fetch unit only takes any action when a message has been received on the feedback path.

3. A memory access will take place if the time stamp received is greater than or equal to that of the local clock (both will be 0 at startup).

4. After an instruction has been fetched, it is guaranteed to eventually reach the execute unit, where it will trigger a feedback event (with an advanced time stamp).

5. When the feedback is generated an appropriate number of fetches can take place up to this point in time, along with a branch if not a null message.

6. An instruction will be fetched in either case, so progression is guaranteed

When running the pipeline in a clocked configuration, this in effect makes all pipeline delays equal and synchronised. Therefore there is no change necessary in the deadlock prevention mechanism. A simplification could be made by requiring a valid feedback signal on each cycle determining the action of the fetch unit in the next cycle. This would not be possible in the self-timed case as the pipeline is no longer synchronised, or necessarily even fully occupied.

The introduction of null messages to the simulation does not in itself limit the reachable state space of the simulation, it simply allows the correct ordering of simulation events. In the processor model, if two events occur simultaneously at the point of branch feedback, the branch event is favoured over the pre-fetched instruction. In a silicon implementation, arbitration could favour either outcome leading to potentially different branch shadow sizes. In the implementation there would be much more variation in timings based on data dependencies leading to a greater number of event orderings. A realisation of the design may suffer a slightly higher branch penalty, this could vary depending on the branch feedback implementation. The important point is that there is consistency between simulations of different Java decoders. Alternative behaviours could easily be explored through modification of the model.

#### 5.3.4.2 Simulation Termination

Termination of simulations is initiated when address -1 is read from memory. This causes a termination message to be propagated around the processor pipeline. Once a pipeline task has forwarded this signal to all of its down stream units, the task cleans up and exits. This procedure allows all processing of valid instructions to complete before the simulation process dies.

## 5.4 System Performance

The performance of the final simulation environment while profiling the execution of different Java aware architectures was approximately two orders of magnitude faster than when simulating hardware. Later versions of the Balsa simulation environment provided the fastest simulation option with JASPA but without any timing information, at a rate of round 10-100 simulated Java byte-codes per second. Verilog simulation providing timing information at the cost of performance; around 10-20 byte-codes per second. A major goal of this synthetic simulation environment was to provide acceptable performance to run more meaningful tests. This goal was met, with the simulator giving a throughput of around 10,000 to 100,000 byte-codes per second.

### 5.4.1 Performance Testing

Simulations were run on a variety of workstation platforms available at the university. These included multi-threaded and multi-processor PC's with processors from different vendors.

The benchmark was run on the simple Java decoder model, with byte-code profiling enabled for sequences of length 1 to 4. Byte-code profiling is discussed in detail in Chapter 6, and adds a large extra overhead to the simulation runs. This can slow down simulation by a factor of 2 or more. Benchmark code consisted of 100000 iterations of an empty loop. This test will produce possibly the lowest simulated byte-codes per second, as branch prediction is absent in the modelled processor. A figure of simulated pipeline cycles will give a better idea of simulation kernel efficiency.

The simulation benchmark was run in the following machines: rain (2.0 GHz Athlon XP), lloyds (2.0GHz Athlon 64), jamaica (3.0GHz pentium 4, with multi-threading ('hyper-threading') enabled, and disabled), antigua (dual 2.8GHz Pentium 4 Xeon) and ringo (1.5GHz Pentium M). The intention being to evaluate the rate of simulation on different platforms, and to determine if the threaded nature of the simulation system brings any performance gains on multi-threaded CPU cores and on multi-processor systems. The results are shown in Table 5.1.

The results show a marked difference when running on different architectures. The most efficient being those with short pipelines, the AMD Athlon series and the Pentium M. Running on a dual CPU system, no performance gain was observed

| *Workstation* | *CPU* | *Threads* | *User Time* | *Sys Time* | *Total* |
|---|---|---|---|---|---|
| rain | Athlon XP (2000) | 1 | 30.4 | 13.8 | 46.2 |
| lloyds | Athlon 64 (2000) | 1 | 27.2 | 10.0 | 37.9 |
| jamaica | Pentium 4 (3000) | 1 | 43.2 | 57.3 | 101.5 |
| jamaica | Pentium 4 (3000) | 2 | 65.1 | 69.0 | 87.2 |
| antigua | P4 Xeon (x2) (2800) | 2 | 62.0 | 63.6 | 85.5 |
| ringo | Pentium M (1500) | 1 | 28.0 | 12.0 | 40.0 |

Table 5.1: Table of Simulation Times.

over equivalent speed uni-processors. Looking at kernel times on this system, shows that any advantage of running parallel threads is countered by a heavy synchronisation cost, due to the high frequency of communications between pipeline stage threads/tasks.

The results clearly show an advantage is gained from the parallel, distributed simulation kernel when running on multi-threaded processors. This is demonstrated on the second generation Intel Pentium 4 'prescott' CPU based machine tested. A gain of 16.5 percent is shown when enabling hyper-threading, allowing the long processor pipeline to be filled with instructions from parallel threads when otherwise stalled. Communication delays are reduced over the dual CPU case, although kernel times show an increased operating system overhead when managing the multiple logical CPU's.

In terms of rate of simulation in pipeline cycles simulated per second, the final simulation clock at the end of the benchmark was 270,017,600 time units. For the benchmark, the delays were configured at 100 time units per cycle on all pipeline stages. This gives a figure of 71,244 simulated cycles per second on lloyds, the fastest machine tested. In comparison with the approximate figures given in terms of byte-codes per second at the beginning of this section for other simulation environments, this translates to 13,000 byte-codes per second, as each loop iteration of 5 byte-codes was completed in 2700 time units, when memory, and misfetched instructions are taken into consideration. This is due to the most basic translation hardware being used, for fair comparison with earlier JASPA simulations.

## 5.5 Summary

An extensible simulation framework for simulating embedded processor pipelines has been presented. The purpose being to evaluate Java extensions to such processors,

with minimum run-time overhead.

Essential to this simulation environment was the ability to model self-timed pipelines, along with synchronous counterparts. This was achieved using a distributed simulation approach, and optimised around the tasking features of Ada, to meet requirements set out at the beginning of this chapter.

Performance was a central requirement, and was met by modelling only the necessary synchronisation between pipeline components. The overhead being reduced over full blown discrete event simulation environments; such would be incurred using Verilog and VHDL. A clear and concise implementation structure was maintained through mapping to Ada tasks, allowing modelling shortcuts to be taken, along with the introduction of simulation profiling features, detailed in Chapter 6.

The benefits of the bespoke simulation environment have been at the expense of a simplified route to hardware implementation. This is not seen as an issue, as the goals of this research are to improve high level architectural support for Java. Having a route to implementation through the simulation environment would not have helped in this respect, and could have possibly introduced variability into resulting designs.

# Chapter 6

# Instruction Folding

This chapter details the implementation and analysis of attempts to improve the performance of Java processor extensions through instruction folding. The idea being to make efficiency gains over the translation techniques used in the JASPA system detailed in Chapter 4. In order to investigate these techniques, the a high level model of an embedded RISC pipeline was created in the Ada based simulation environment, explained in Chapter 5 of this thesis.

While providing simple hardware decoder extensions based around a stack-cache provides significant gains over software only interpreters, there are many potential gains to be made over the approach taken in JASPA. A reoccurring pattern of execution is that of loads from (local) variables onto the stack, followed by arithmetic operations followed by storage back to the variable space, in our case memory. Variables are often repeatedly processed in this way in close succession. This has led to the adoption of a deep register file or local on chip memory specifically to cache the top of stack, and local variable storage in more performance orientated Java cores, such as PicoJava.

The aim of this chapter is to show, how such sequences of repeated memory accesses can be made unnecessary by applying folding to sequences of byte-codes. This can be done to some extent without needing to cache large amounts of the Java stack, improving efficiency over the stack cache decoder used in JASPA but without the hardware cost of additional registers. Such an approach will improve the power efficiency of the Java decoder through reducing the number of processor cycles needed per byte-code, at only the cost of slightly increased decoder complexity.

There are drawbacks with this approach in that there is less scope for folding when the entire local variable pool is not available in registers for immediate access,

in contrast to dedicated processors such as PicoJava. Improvements in efficiency are however very much worth investigation, complementing the existing power and silicon cost benefits of the Java decoder pipeline extension approach to acceleration.

In order to measure and test the introduction of folding approaches, a profiling capability was added to the processor pipeline model. This allows timings to be traced for each byte-code as it is processed and retired during execution. This enabled the effect of folding to be measured over the course of a benchmark program, breaking down times and frequencies for individual byte-codes and more importantly groups of byte-codes processed during a simulation run. As well as providing concrete information on where improvements had been made, through analysis of which sequences of byte-codes took the most time to execute over the course of an entire program run, it suggests other areas where optimisation can be applied.

## 6.1 Architectural Profiling

Profiling of the effectiveness of Java pipeline extensions is essential for the further understanding of such systems. A better breakdown of what is being achieved by the system is needed, over cumulative timings achieved running different benchmarks. A benefit of modelling the system in software, is the ease of integration of profiling code. This section discusses the final processor model used to experiment with improved approaches to accelerating the execution of Java through processor pipeline extensions, along with the profiling system used to analyse their performance.

### 6.1.1 Model of Java Processor Architecture

Chapter 5 describes the optimised approach taken to modelling processor pipelines using Ada, the key simplification being the single point of feedback in control flow present in the model, namely the branch feedback system. A single, extensible architectural model was used to evaluate Java acceleration approaches within an embedded RISC processor core. The main point of interest being the internals of the Java decode pipeline stage, and how it produced translated RISC instructions for the remainder of the pipeline to execute.

As previously described, the simulation environment models each pipeline stage as an independent task or thread in the system. Timing information is passed along with data in messages between modules, providing timing and a correct ordering of events in the simulation.

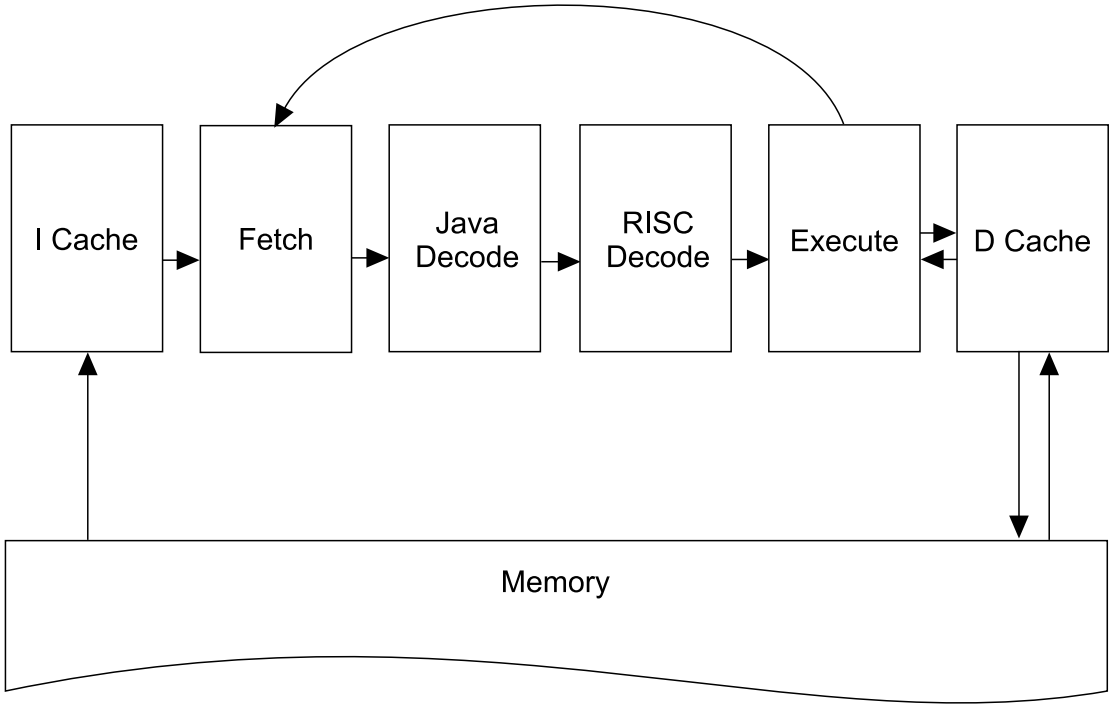


Figure 6.1: Simulated Architectural Processor Model.

Figure 6.1 shows the simplified processor pipeline model used to test enhancements to the Java enabled architecture. The pipeline is simplified to a three stage configuration, before addition of the Java decoder, common in low power embedded cores. Although more advanced processors may have extra stages in order to increase the maximum operating frequency, it is felt that a three stage system will be totally adequate for investigating improvements to the Java processing subsystem. Simulation speed will also be marginally improved as a side effect of the simplification.

The pipeline stages are modelled in the following manner:

#### 6.1.1.1 I Cache

Models a perfect instruction cache, with a configurable, fixed latency for instruction fetch.

#### 6.1.1.2 Fetch

Fetches 32 bit quantities (RISC instructions) from memory via the instruction cache. The instruction fetch unit can not handle misaligned fetches, only fetching the corresponding word aligned data. This is to mirror the behaviour of the fetch unit present in JASPA, and other embedded processors such as the ARM. This will slightly increase the number of fetches needed when executing Java code, due to the non-word aligned nature of Java byte-codes. This will only occur on branches to a new block of byte-codes, subsequent sequential accesses be on word boundaries.

Branches are handled in the same way as in the AMULET [18] and (JA)SPA [38, 10] processors. A colour is associated with each instruction fetch. The colour is passed along with the instruction through the pipeline, where it is only executed if it matches the current colour at the execute unit. This colour is inverted at the execute unit when a branch is taken, and the branch information is sent back to the fetch unit. When the fetch unit branches to the new address, its colour is changed also. Any incorrectly fetched instructions in the pipeline will not be executed, as their colour will no longer match at the execute stage.

#### 6.1.1.3 Java Decode

This unit is where different approaches to decoding Java byte-codes can be implemented and tested. This is placed logically in series with the RISC decode unit, although its output passes through the RISC decode unit without processing, exposing the decoded instruction interface to the execute unit. This modelling trick allows for simulation of both serial and parallel decoder extensions and for more interesting experiments to be carried out with different Java decoder models.

The execute unit and its interface is parametric, this allows more registers to be simulated than available in the standard ARM instruction format and architecture. Beyond exploring the possibilities of other embedded RISC platforms, the impact of architectural limits can be assessed, possibly justifying extensions to the execute unit for Java acceleration.

Within the Java decoder module, many different timings can be assigned to operations, allowing the exploration of the effects of self-timed operation. Examples include: the modification of stack cache state having an assigned cost, extraction of a byte-code from the currently buffered word and the generation of an instruction for the execute unit. These timings can be calibrated against measured values in the JASPA core.

The Java decoder passes thorough RISC instructions, with zero latency when not in Java mode.

#### 6.1.1.4 RISC Decode

This unit decodes a subset of the 32 bit ARM instruction set. This allows existing code from JASPA to run without modification on the new high level model. Many simplifications have been made, such as the removal of supervisor modes, asynchronous interrupt handling and 16 bit memory operations. This allowed for rapid implementation and minimal run-time overhead, without putting restrictions on the JVM environment under test.

The RISC decode stage is also responsible for passing through the output of the Java decoder to the execute unit. This output is in the decoded instruction format required by the execute unit. Through the variation in delay associated with this operation either parallel or serial Java decoder operation can be simulated. In the parallel case, zero latency is added, in the serial case the RISC instruction decode latency can be added.

#### 6.1.1.5 Execute

The execute unit essentially implements the features of the ARM execute unit, allowing processing and storage to registers with the operations defined in the ARM instruction set. This set of operations is typical of most embedded RISC cores. In order to provide support for the simulation of other target architectures, and to investigate the efficiency of Java acceleration approaches in a broader light, the execute unit was made parametric, the parameter being the number of entries in the register bank. A flat register bank was implemented, with an arbitrary number of registers possible. The op-codes used to implement the subset of the ARM instruction set can be applied to any of these registers, the Java decoder being able to issue all such operations through the decoded instruction interface.

The branch feedback mechanism, with null messages notification of the absence of a branch, is implemented as in Chapter 5. This avoids the unnecessary pipeline synchronisation and stalling associated with an actual simulated communication back to the fetch unit on each execution cycle.

#### 6.1.1.6 D Cache

Models a perfect 32 bit wide data cache, with a configurable, fixed latency for data accesses from the execute unit, allowing parallelism with instruction fetch.

### 6.1.2 System Timing

Delays are managed in the simulation system, described in Chapter 5, as defined by individual pipeline models. A local clock stores the current observed simulation time, in each simulation process. As processing on data is carried out, appropriate delays are added onto this local clock, which is propagated through the system when communications occur between modules. These delays can be made data dependent in order to model self-timed design styles.

In order to allow more flexibility in the timing system, a delay description file format was invented. This format allows for the assignment of delays within the system at run-time, from a file, enabling different scenarios to be tested on the same model with different timing profiles. The main purpose being to allow for the removal of data dependent delays to simulate synchronous behaviour, applying a global worst case cycle time to all pipeline stages. The file containing latencies is loaded and parsed at start-up, latencies for all modules being stored in a central database. Each module extracts the named times in turn, from the database passed at instantiation time.

```
module RiscDecode
begin
  riscDecodeTime := 100;
  javaDecodeTime := 100;
end
```

Figure 6.2: Part of a latency description file.

An example of the delay format is shown in Figure 6.2. This case showing how a serial or parallel decoder structure can be emulated, in the timing description for the RISC decode stage, by replacing the Java decode latency with zero.

### 6.1.3 Profiling Byte-code Execution

For the purpose of better understanding the performance of different approaches to Java execution a profiling system was designed and integrated with the processor

model. This allows for the timings of each different byte-code's execution to be recorded over the duration of a simulation session. The idea behind this being to extend this profiling approach to groups of byte-codes. Not only does this provide the frequency of execution of different sequences, but also their contribution to the execution time of a benchmark. Thus optimisations can be both targeted at improving specific bottlenecks, and be analysed in a balanced and detailed way.

The profiling system was separated into a separate Ada package, or abstract data type. This allowed a profile structure to be passed to a series of function calls updating the profile based on new timing information about a byte-code. Abstracting this part of the system caused minimal disturbance to the clarity of the code in the Java decoder module, as well as making the profiler reusable between different decoder implementations.

The collection of profiling information is complicated slightly by the fact that multiple pipeline stages are involved in the execution of a Java byte-code, and this may involve multiple execution cycles. A more serious problem is introduced by the uncertainty of misfetched byte-codes, those in the shadow of a taken branch. These issues become more of a problem when dealing with the profiling of sequences, as it must be ensured that only the executed sequences are added to the profile, with time spent processing the shadowed byte-codes being counted towards the branching byte-code(s).

#### 6.1.3.1 Profiling in a Pipelined Environment

The time taken to execute a byte-code can be determined at the Java decode stage of the pipeline, by tracking the local clock time between the start of processing one byte-code and the start of the subsequent one. These timings can be measured at the time the byte-code is extracted from the currently buffered instruction word. The only problem is visibility of whether the current instruction colour is valid, as this information is only ever correct at the execute unit. This information is required, as it determines whether the timing information is to be counted against the current byte-code or as part of a branch shadow in a previous byte-code.

The branch problem is solved by introducing a system of queuing to the profiler package. Two FIFO queues are used to buffer information:

**Byte-code FIFO** Stores information about byte-codes passing through the Java decoder.

**Branch FIFO** Stores information about branch results.

As each new byte-code is processed by the Java decoder module, a notification along with details of the current local time is sent to the profiler. The profiler places this information onto the byte-code queue, which will make a *potential* time profile of the byte-code when the next byte-code is queued, giving a start and end time. This information is only a potential profile, as it is not known whether the byte-code in question is in the shadow of a branch or not.

In order to commit any timing information to the profile about how long a byte-code has taken to execute, it must be known whether any branch results are outstanding. In order to acquire this information, a flag is sent when informing the profiler of a new byte- code, detailing whether it will issue a branch or not. This is known in advance for all byte-codes in the decoders investigated here. This flag causes an increase in a pending branch result counter in the profiler data structure. If the pending branch count is zero, then items from the byte-code queue can be committed to the profile as each byte-code is processed. When there are pending branch results, only items up to the branch can be committed until the corresponding branch result is known. An arbitrary number of pending branches can be present using this system.

Although the Java decoder module has a knowledge of instruction colour, this is not sufficient to determine if a branch has been taken or not, as no notification will be given when a branch is not taken. If multiple possible branches are outstanding, and a colour change is observed, it will be impossible to tell which one was taken in a such a self-timed processor design. One proposed solution to this problem was the introduction of colour shades, which change when a branch is not taken, and can thus be observed by the Java decoder. Another solution would be to tag times to instructions, only assigning these times as they are executed. The problem with this tagging would be managing the split between profiling in the Java decoder and execute unit tasks, running in parallel. A more simple solution was eventually implemented, whereby on each branch in Java mode the execute unit communicates directly with the profiler informing it of whether the branch was taken or not. This information is placed in the branch FIFO queue and is accessed by the profiler as each byte-code and associated time is sent from the Java decoder, this happens when any pending branch results are used to commit information to the profile. The FIFO is implemented as a protected object in Ada, assuring mutual exclusion and atomic operation between the calls from the execute and Java decode threads.

This structure is shown in Figure 6.3.

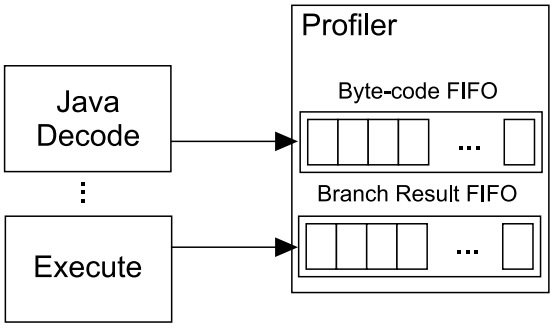


Figure 6.3: Branch Profiling Queue System.

### 6.1.4 Profiling Byte-code Folding Systems

The idea behind extending the Java decoder with folding systems was to improve the performance of certain groups of byte-codes which can be executed more efficiently by circumventing redundant memory operations. In order to assess the timings of different groupings of byte-codes, the profiler was extended to store information not only about the execution of each individual byte-code, but also arbitrary groupings of byte-codes executed during simulation runs.

The approach taken was to constrain profiles to a range of byte-code sequence lengths. The profiler can be set up to profile sequences in any continuous range of sequence lengths. The mechanism for dealing with this was another buffer stage before byte-code timing information is committed to the profile. When a byte-code's branch status and timing is resolved it is inserted into the profile buffer. This is a circular buffer containing the last byte-code profiles up to the maximum sequence length. At this point, any new sequences are added or modified in the profile database, as appropriate. The buffering structure of the final profiling system is shown in Figure 6.4.

There are different ways of measuring the timings of byte-code in sequences, when folding is being applied in the Java decoder. As multiple byte-codes are combined into a single cycle execution operation the time for each instruction in the batch is effectively spread across the folded set of byte-codes. In order to cater for this in the profiler, the concept of byte-code profile batches was introduced. The start and end of a batch is signalled when informing the profiler of the execution of a new byte-code, only committing information in a batch when it is complete. The

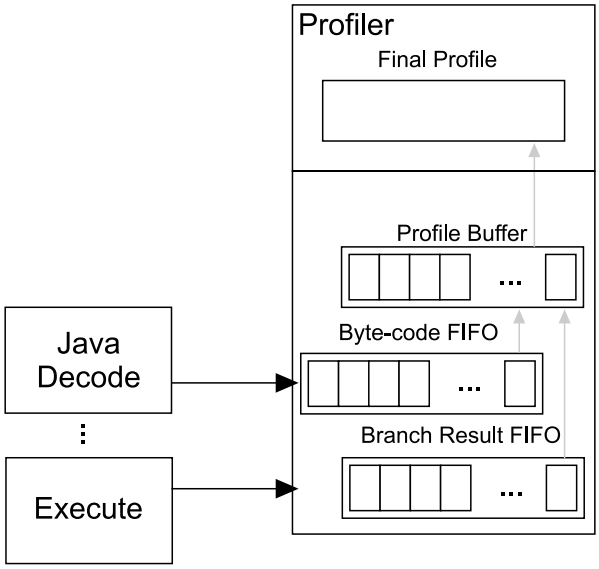


Figure 6.4: Complete Profiling System.

profiler can then add up latencies as individual times or averaged over batches, which makes more sense as it reflects the processing time as part of a folding operation. Otherwise there would be spurious zero delay execution for folded byte-codes, with the last byte-code in a batch being assigned the time taken for all instructions in the batch.

#### 6.1.4.1 Profiler Summary

The resulting byte-code profiler allows for the time spent executing each Java byte-code, and different byte-code groups of arbitrary length to be recorded. This helps to show where time is being spent in different benchmark runs, allowing for a detailed analysis of architectural improvements to the embedded pipeline in support of Java execution. This information can be used to explore further improvements and angles of attack, in the quest for an efficient acceleration strategy, making the best use of available resources in such processors.

## 6.2 Stack Cache Decoder

The stack cache decoder model was implemented as a cycle for cycle copy of the Java decoder in JASPA, described in Chapter 4. This implementation is used as a control to compare improved folding decoders against, within the same simulation

framework, it has identical memory and instruction execution models. The re-implementation also allowed for a slight extension of the original decoder, free from the restrictions of the ARM compatible host core, with the potential to use more registers for caching the stack in the parametric execution unit.

The translation mechanism is therefore exactly the same as in JASPA, caching the top of the operand stack in registers (four by default). These registers can then be operated on by the RISC execution unit, the load/store architecture not able to operate directly on values in memory. The advantage of this approach is the simplicity of the design. This has been shown to result in a small, low power solution, easily integrated into an existing processor core. Performance is not optimal though, as stack based instructions spend many cycles pushing and popping from the operand stack, causing expensive memory operations in this decoder, which should ideally be minimised to improve efficiency further.

### 6.2.1 Implementation

The implementation of the stack cache decoder can be decomposed into the following stages of operation:

**Byte-code Extract:** Extract byte-code and operands from fetched word, this may involve further fetch cycles due to alignment issues caused by variable length immediate instruction operands.

**Code Generation:** Once byte-code is fetched, correct handler sequence is looked up, and issued to the execute unit, this may be a branch to a software handler.

**Stack Management:** The stack cache is managed during code generation, as the operand stack is stored in registers in order to be operated upon by the RISC execute unit. This involves keeping track of current occupancy information.

**Branch Handling:** When branches are taken, stack cache state must be kept consistent, this involves flushing the stack-cache as in JASPA.

Register assigment is also the same as in JASPA, making using four for the operand stack cache and others in the management of the Java and RISC stacks. Other registers are used to pass information to RISC handler code, and manage jumps to and from Java mode. In practice, the four registers used for the stack cache have been relocated to the end of the register space (from R16), so the cache

size can be increased in future tests. This however does not change the observable behaviour of the system when configured like JASPA, with four stack cache registers.

#### 6.2.1.1 Performance

The stack cache decoder unit should perform exactly as JASPA did in earlier tests as the decoder generates exactly the same code sequences. The aim being to show the advantages that folding techniques can bring over this simple approach.

During the decoding process, many of the cycles which occur during extract and code generation phases of operation have quite different complexity and timing. This suits a self-timed implementation allowing such short cycles to complete quickly and longer cycles only stalling the fetch unit. It may however be the case that other stages in the pipeline will stall execution in some of these cases reducing such perceived benefits.

## 6.3 Byte-Code Folding

The execution of multiple Java byte-codes in a single execution cycle is called *instruction folding*. PicoJava [35] and JEMCore [49, 23] both take the same approach to the problem. These processors have access to local variables in registers, so they can execute directly on them, without loading and storing to a temporary operand stack. This can only be achieved for a small set of patterns which can be matched as byte-codes are decoded.

### 6.3.1 Achieving Folding

Initial experiments with instruction folding added a second set of registers to cache locals for direct access at the execution unit. This local variable cache was added at the end of the ARM register space, adding eight new registers. Having this dedicated local cache also freed r4, previously used to store local variable zero. The stack cache could therefore be extended to five entries.

There were several limitations with this approach. The main problem was due to the decoupled nature of the Java decoder and execute unit. As the Java decoder is not aware of whether it is executing code in a branch shadow, the caches must be flushed frequently, whenever a branch is issued. It became clear during implementation, that the cache brought only a small performance gain, in comparison to

the cost of extra registers and increased complexity of code generation. Frequent flushing of the caches to memory proved to be a problem, a cost not incurred if this caching is managed at the execute stage. This is a problem dedicated Java execution cores do not suffer from.

Although functionality for management of the stack cache could be added to the execution unit, this would detract from the clean and portable nature of the Java decoder extensions. The Java decoder would no longer be portable between processor cores. Implementation effort would be much greater, adding complexity to the performance critical execute stage. The result would be more like a dedicated Java processor than an accelerator extension. A new approach to instruction folding is required in this context to make better use of available registers.

#### 6.3.1.1 Using the Stack Cache

An observation was made while looking at traces of byte-codes being executed by the Java decoder. It appeared that local variables were often loaded onto the operand stack repeatedly. Other cases showed that locals were stored and then re-loaded. In the simple Java decoder such operations would result in fetches from memory, when the value was already at another location in the stack cache, or even in the correct register. Applying optimisations to remove these redundant memory operations on locals allows for folding of byte-codes. This solution does not require further caching structures makes efficient use of registers

The initial folding decoder took this optimised approach to implementation. The penalty being storage of extra state at the decode stage regarding the local variable status of stack cache registers. Checks before local load and store operations become necessary, with the benefit being removal of redundant RISC instruction cycles when folding can be achieved.

### 6.3.2 Implementation

The implementation of the first folding capable decoder is a direct extension of the preceding stack cache based decoder stage. Essentially this cache is being extended over the local zero register, r4. Information is kept about any valid local variables being stored at each location in the cache. This was implemented in the simulation environment in such a way that it could be mapped to any register range, enabling tests to be carried out on larger cache sizes outside the ARM register space. This is

thanks to the parametric execution unit described at the beginning of this chapter.

The decoder was implemented as a replacement for the stack cache decoder, with all of the same advantages. All decoding logic is localised to the Java decoder extensions, retaining the advantages of being a portable pipeline extension suitable for embedded processors.

Parallelism could be increased by having a larger window of byte-codes available to the decoder. In an asynchronous implementation this could be achieved by adding an extra pipeline stage, without the need to modify the rest of the system. This approach was not taken as integration with the synchronous model would require modification to the rest of the pipeline, which is not desirable.

#### 6.3.2.1 Folding Mechanism

The proposed folding mechanism works by eliminating redundant local variable loads and stores by reusing values already in the stack cache registers. This approach is not as costly as caching the local variable space directly. There is no requirement for modification at the execute stage of the pipeline for efficient operation.

Keeping track of which local variables exist in an unmodified state is the key new requirement for the folding Java decoder. This allows cached local variable loads from memory to be replaced by a cheaper register copy, or removed completely if the value happens to be at the top of the operand stack. Local variable stores can be deferred until absolutely necessary, removing further redundant memory operations. Examples of folding operation are shown in Figure 6.5.

#### 6.3.2.2 Decoding State

In order to track which local variables are currently in the stack cache a table of 3 attributes per entry was devised. These attributes are summarised below:

**Valid** (Boolean) stating that a valid local variable is stored in the register associated cache entry.

**Variable** (Integer) storing which Java local variable this stack cache entry is currently holding.

**Modified** (Boolean) stating that this entry has been stored as a local by a Java byte-code, meaning it must be returned to memory before being overwritten.

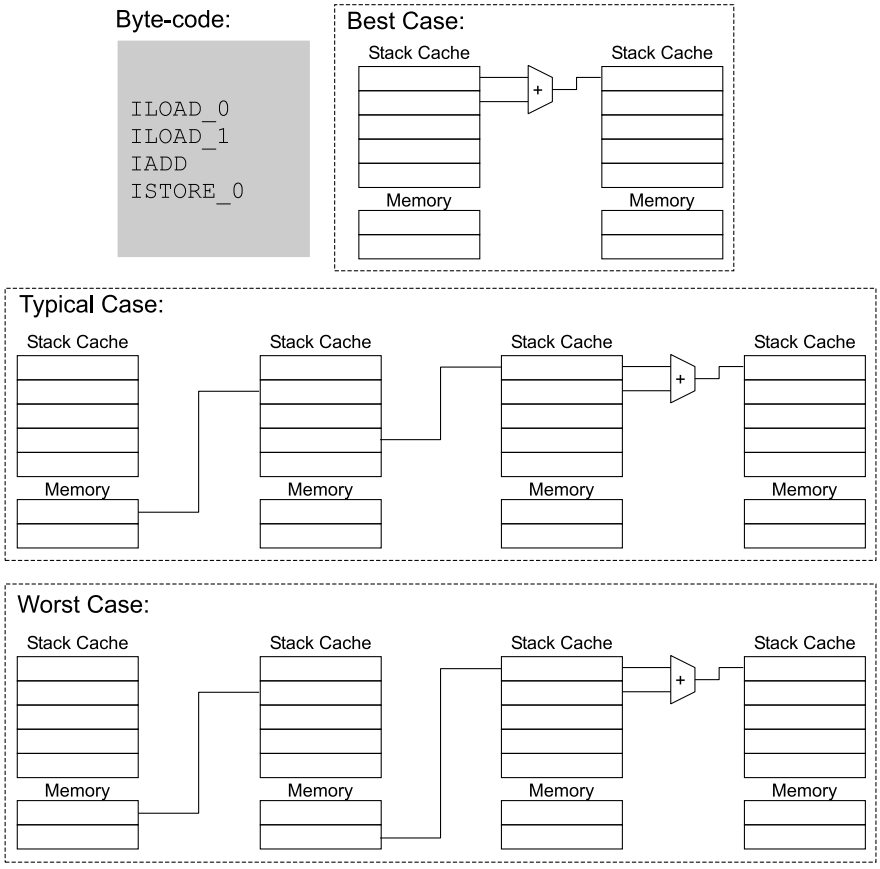


Figure 6.5: Folding Examples.

The *valid* bit is set when a local variable is loaded from memory to the operand stack. The register is now storing a valid local variable. The variable is invalidated as soon as the register is modified by an instruction issued from the decoder. Examples of how valid variables can be left in stack cache registers as a side effect of different byte-code sequences are shown in Figure 6.6.

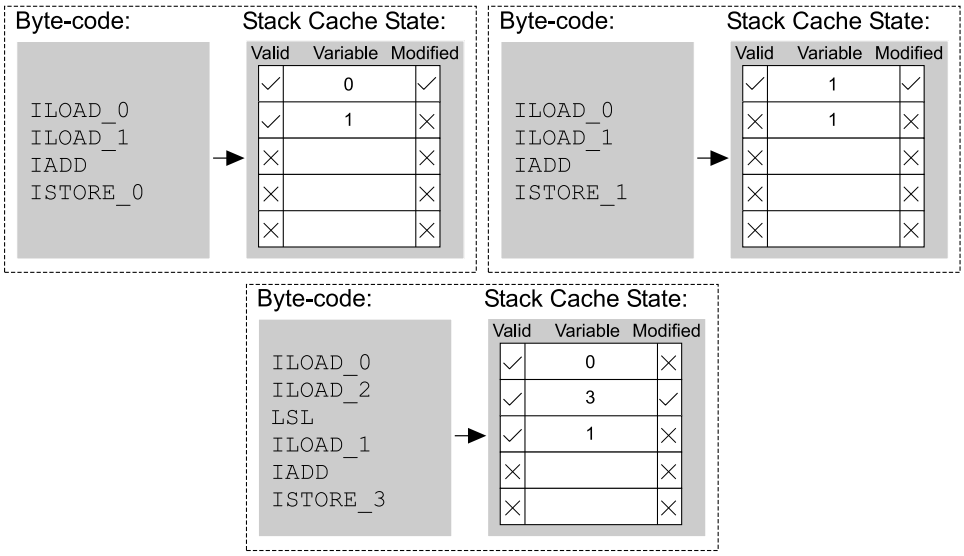


Figure 6.6: Validity Examples.

Multiple copies of a local variable can be present in the stack cache, this can be reflected in the state table. Taking this approach maximises the retention of local values in the cache, minimising memory accesses. The negative aspect of this approach is the requirement to invalidate all duplicate entries when the local variable in question is modified and stored back to the variable space.

The *modified* bit in the stack cache state table serves an important function regarding deferral and folding of memory store operations. After a local variable is loaded into a register and then modified by an arithmetic operation, it becomes invalid. If the result is stored back to a local variable, the value must now be stored back to memory. The modified flag, allows this to be deferred until the register is needed (invalidated) by another operation. The flag also allows only modified locals to be stored in the event of a stack cache flush, which is actually the common case, reducing memory overhead to a minimum in this event. A stored local in the stack cache is very likely to be quickly overwritten as the byte-code responsible operates on the current top entry of the operand stack. In the event of a local store,

duplicate entries of this local must be invalidated, as they will now potentially store an incorrect previous value.

In the event of a stack cache flush, the operand stack is written back to memory, along with modified local variables. Unfortunately when the operand stack is re-cached, any information on parity with local variables is lost. An option for future implementations is to try and store this information in a modified stack frame format, improving folding performance between flush events.

#### 6.3.2.3 Implementation Options

A hardware implementation of the stack cache based folding algorithm, requires a simple table of registers to store the required state. This table will be a fixed size, dependent on stack cache size. A tradeoff can be made regarding the size of the variable field, allowing up to $n$ locals to be included in folding operations at the cost of $\log^2 n$ bits per table entry.

Performance of necessary operations on the table should not pose a problem to implementors. Searches through the table can happen in parallel, due to the relatively small number of stack cache entries. Other operations only involve addressing and simple read and write operations. Stack cache based folding is very simple, thus attractive in terms of implementation cost. Power consumption should not be increased significantly over a simple stack cache decoder without folding, as the only additional logic concerns the small state table, although it must be accessed for each decoder operation.

### 6.3.3 Simulation and Testing

The folding capable Java decoder will always perform at least as well as the the stack cache based JASPA design in terms of the RISC code generated. This is because in the worst case, the decoder will perform exactly the same operations. In cases where local variables are already present in the stack cache, memory operations will be reduced, and RISC execute cycles eliminated.

#### 6.3.3.1 Test Models

In order to test the efficiency of the new Java decoder, a series of benchmarks were run. The new decoder can be compared against the existing stack cache design. Further to this, a new decoder model was built to simulate idealised instruction folding.

In this model, any memory load and store operations related to local variables do not incur any time penalty. This is equivalent to a perfect variable cache, where local variables are always available in a register for operation, in this case appearing on the operand stack cache without delay. This model, referred to from now as the perfect folding decoder, was built from the original stack cache decoder, modifying the decoded instruction format to allow zero cycle time memory operations where appropriate.

#### 6.3.3.2 Simulation Timing

For this series of tests, the processor pipeline was set up with a synchronous timing profile. Each pipeline stage has a cycle time of 100 simulation time units regardless of the operation it performs. This also applies to the perfect memory cache units. This timing profile was used to test the quality of the output RISC code generated by each decoder without confusing results with any further variables.

#### 6.3.3.3 Benchmarks

The following benchmark programs were used to test the Java enabled processor pipelines:

1. Arith - a repeated mathematical integration applied across a range of values, testing the ability to cache several local variables on each iteration.
2. Fib - An iterative loop calculating a value in the fibonacci sequence, using several local variables.
3. NFib - A recursive version of Fib, using only expressions and return values
4. Sieve - A prime number sieve function, from an embedded Java benchmark, used for testing the JOP processor [40].

### 6.3.4 Simulation Results

Figure 6.7 shows the relative performance of each of the Java decoder modules, as a ratio of simulation time taken compared to the basic stack cache decoder. The breakdown of timings can be seen in Appendix A, showing accumulated times for individual byte-codes over each benchmark along with accumulated timings for groups of four byte-codes, targeted by instruction folding.

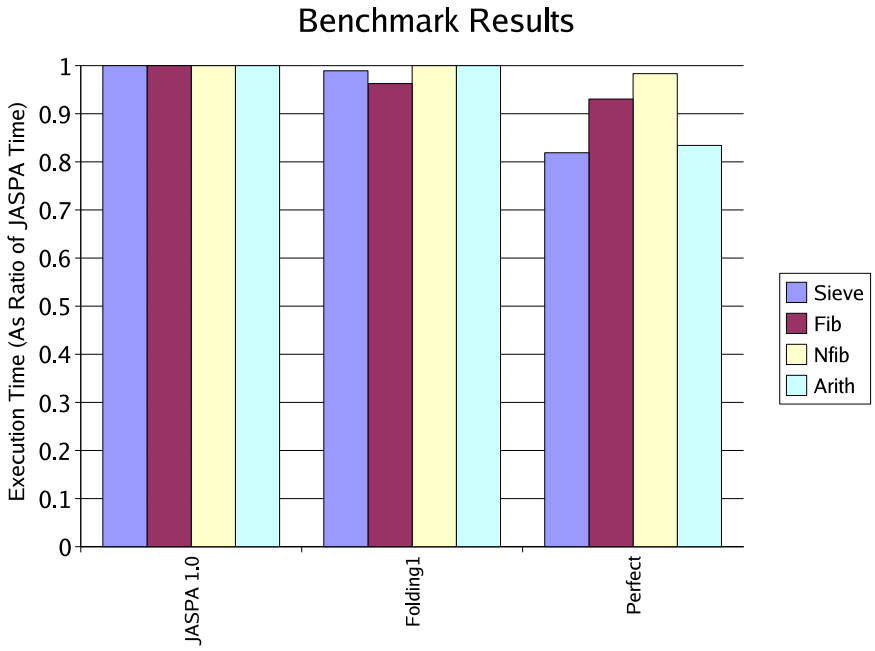


Figure 6.7: Results Relative to Stack Cache Decoder.

### 6.3.5 Conclusions

The results for the byte-code folding system appear to show little gain in performance, the best case being in the order of a 5% reduction in processor cycles when running the iterative Fibonacci benchmark. It was expected that the Arith benchmark would show a significant improvement, but this was not the case.

The breakdown of byte-codes and groupings for the folding decoder are in Appendix A, Figures A.3 and A.4. The problem, at least with the Arith benchmark, is clearly caused by unacceptable branch latencies in both the *goto* and *if_icmpge* byte-code handlers. Both of these instructions are dealt with in the hardware decoder. This issue is dealt with in Chapter 7.

## 6.4 Summary

This chapter has presented a new approach to folding of byte-code operations in a RISC processor pipeline with a Java aware instruction decoder. This approach improves upon existing stack cache based designs allowing for folding of memory operations with local variable processing, when values are already cached in stack

registers. The new Java decoder requires only a simple lookup table for implementation and has the same worst case execution profile as the simple stack cache decoder.

An embedded profiling environment has been presented and has been used to diagnose problems with branch behaviour in the Java decoders presented in this chapter, prompting the work presented in the following chapter.

# Chapter 7

# Improved Instruction Folding

Chapter 6 introduced a new approach to byte-code folding within an embedded Java aware microprocessor core. The limitations of this approach were quickly established, the main problem being caused by branch latencies within the operation of the decoder. This is compounded by the flushing of the cache when branches are issued.

This chapter presents several optimisations to the folding Java decoder to address performance issues. Firstly the branch shadow problem is addressed, making use of available information about branches issued from the decoder to infer the shadow state. The use of more registers is explored and also the impact of the self-timed nature of the design.

A new system of folding is presented at the end of the chapter, aiming to increase the efficiency of this combined stack and local variable cache approach to code generation. The main problem with the first system is the redundancy of registers and frequent overwriting of potentially useful cached local variable values.

## 7.1 Branch Shadow Optimisation

When a branch is issued from the Java decoder, the state of the decoder and any cached registers are flushed. This is because code in the shadow of the branch may corrupt these values if the branch is taken. Secondly, a serious problem is the potentially large number of RISC instructions which can be issued in the shadow of a branch. As each fetched word contains a maximum of four byte-codes, this can lead to tens of erroneous RISC instruction issues per branch. This may be less of an issue in an asynchronous implementation where instructions of incorrect colour

will be handled more quickly in the execute unit, but it is still a problem.

There are several pieces of information available to the Java decoder which can help determine shadow state. Unfortunately, after issuing a conditional branch the shadow state becomes completely unknown until an instruction colour change occurs. When an unconditional branch is issued, the next instructions will always be in shadow, assuming that the branch was not issued in the shadow of another branch. When a colour change is seen at the decoder, this is the result of a taken branch, the following instruction can never be in a branch shadow.

Using the above information, it can at least be determined if the next byte-code's shadow state is either: not in shadow, unknown or definitely shadowed. When it is known that the fetched byte-codes are in shadow, decoding and RISC instruction generation can be skipped until the colour change occurs signalling the branch has been acted upon and the instruction word was fetched from the branch target address.

### 7.1.1 Implementation

This branch optimisation was implemented into the decoder, and had a definite impact on performance, shown in Section 7.3. It particularly helped reduce the effect of goto byte-codes commonly used by the Java compiler in conditional code and loops.

The cost of implementation was a register recording current shadow status. This register holds one of the three possible shadow status values. This flag is set when issuing branches, including those to software handler routines. Once the shadow status is unknown, it remains in this state until a colour change is observed, resetting the state to being unshadowed. The flag is checked when fetching the next byte-code from the fetched instruction word. In order to improve performance further, this system could be extended with feedback from the execute unit or with other mechanisms to increase the visibility of true shadow status.

## 7.2 Further Optimisation

### 7.2.1 Extending the Register Cache

As the folding-capable stack cache was designed in a parametric way, the benchmark tests were run on a version of the processor pipeline with 8 stack cache registers for

comparison with the original size of 5. This test was designed to assess the scalability of this folding approach.

### 7.2.2 Asynchronous Operation

All test runs so far have focussed on a synchronous timing model, where each pipeline stage in the system operates with a cycle time of 100 simulation time units. In order to show how the decoder can function in a self-timed manner, an synchronous timing profile was devised, based upon a worst case cycle time of 100 time units. Each operation within the Java decoder was assigned a time value, based upon evidence from the Balsa JASPA implementation, described in Chapter 4.

## 7.3 Simulation Results

### 7.3.1 Comparison of Java Decoders

Figures 7.1, 7.2 and 7.3 show graphs of simulation times and relative performance of the optimised Java decoders against the original JASPA decoder. This information is supplemented by detailed breakdowns of byte-code timings in Appendix A. The perfect folding decoder from Chapter 6 is also included for comparison.

It can be clearly seen that the effect of the branch optimisations, labelled as 'Folding1-branchopt' have an noticeable impact on performance in some of the benchmarks. The branch optimisation shows an increase in decoding performance of around 10% over the original folding decoder. The timings in some benchmarks improve upon those achieved with the perfect local cache model.

The effect of increasing the number of registers in the stack cache to 8 is shown in the results labelled 'Folding1-branchopt8'. The benchmark times were reduced in all cases, although not by any significant margin. It is thought that the folding scheme used does not make the best use of these registers as caching is localised to a small set of registers around the top of the stack. Potentially useful local values, when modified and stored are overwritten by other operations very quickly, as they occupy the top entry on the stack cache.

Interestingly the Arith1 benchmark shows no improvement in performance on any of the optimised decoder modules. Figure A.6 in Appendix A, shows that the branch optimisations are not effective due to the presence of a conditional branch

around the goto, this is where most of the execution time is spent in all decoder models.

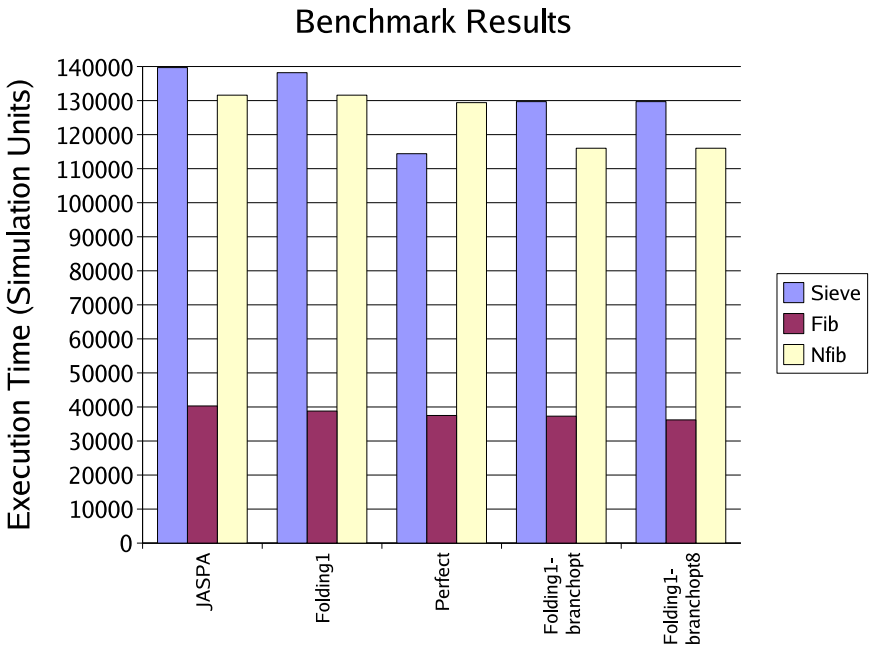


Figure 7.1: Simulation Benchmark Timings.

### 7.3.2 Memory Latency

All of the simulation runs so far have been carried out using the 100 time unit synchronous timing model. The memory subsystem has been modelled, rather unrealistically, as a single cycle perfect cache. The folding optimisations are not only targeted at reducing the number of execution cycles per byte-code, but also the number of memory operations. The simulation results graph in Figure 7.4 show how the impact of increasing memory latency in the JASPA decoder and branch optimised folding decoder while running the Sieve benchmark.

The labels in the Y-axis have a postfix denoting the number of execution cycles that data cache access operations cost in each simulation model. The original models had a memory timing of 1.0, while timings of 1.5 and 2.0 were also tested. These figures accurately reflect the fact that memory in embedded systems is often clocked at a similar rate to the processor core, and may even be on the same silicon die. The changes were only applied to the data cache in order to highlight differences in

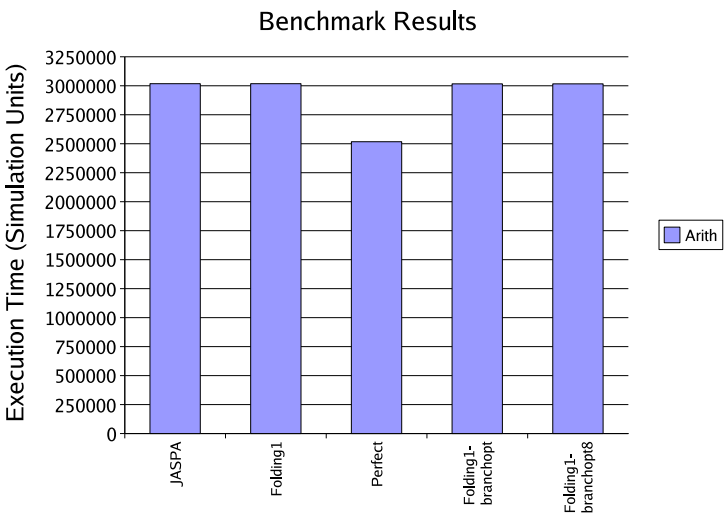


Figure 7.2: Arith1 Benchmark Timings.

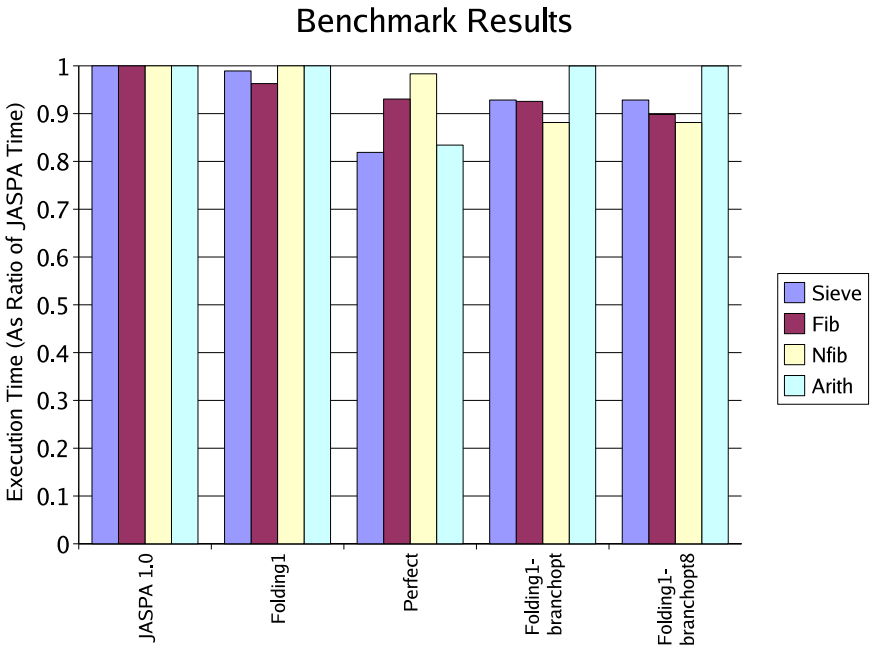


Figure 7.3: Results Relative to Stack Cache Decoder.

memory accesses generated by the folding algorithm.

It was thought that the folding decoder would show increasing performance gains over the stack cache decoder as memory latencies increased. This is clearly not shown in the experimental results. This shows clearly that the number of memory operations saved by the folding decoder is not significant in comparison to the amount of other memory traffic generated when executing the Java code.

The results in Figure 7.4 do not show the absolute timings, which did increase by 35% over the single cycle model when running with a data memory latency of 2.

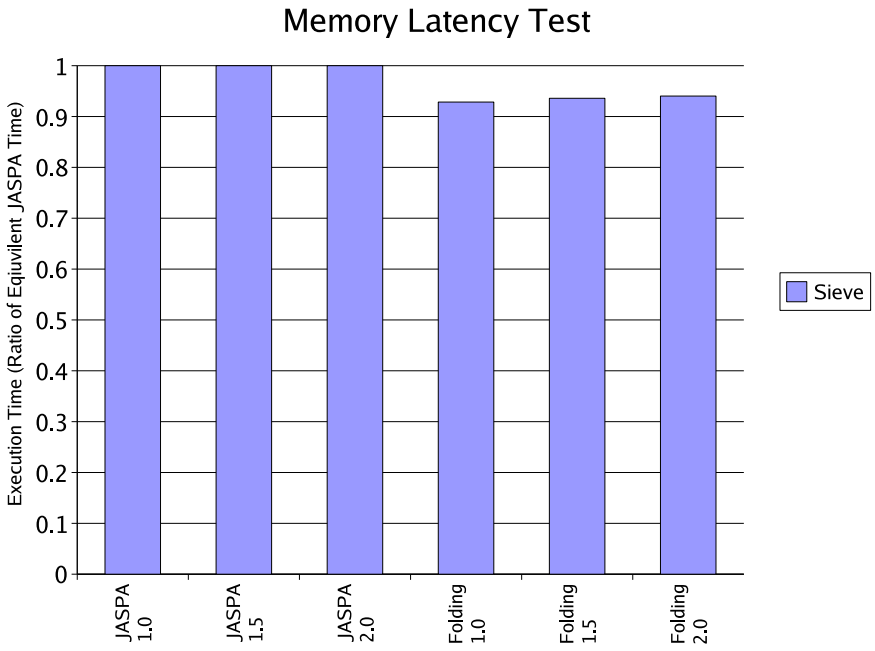


Figure 7.4: Result of Varying Memory Latency.

### 7.3.3 Asynchronous Timing

Figure 7.5 shows the result of applying an asynchronous timing profile to the Java decoder. Previous simulations had a fixed cycle time for the Java decoder, 100 simulation time units per RISC instruction generated. Timings were applied to individual operations within the decoder, such as RISC instruction generation, fetching bytes from the word buffer, updating stack cache state and decoding the byte-code. These timings were based upon timings observed in the hardware JASPA implementation described in Chapter 4. The worst case cycle time is still 100 simulation time units,

matching the synchronous model.

The results show quite clearly that relative to the synchronous model, the folding decoder with branch optimisations performs with increased efficiency using the asynchronous timing profile. A 10% reduction in execution time is achieved over the synchronous pipeline model. This is quite an achievement, as the improvement can be observed even when combined with stalls in the pipeline caused by other stages. Timing in the rest of the pipeline remains the same as in previous simulation runs, and the execute unit timings do not take into account early completion of arithmetic operations.

The reason for the efficiency gain is that operations within the self-timed Java decoder now overlap better with communications with other pipeline stages. Incoming instruction words from fetch, and output of RISC instructions to the execute unit overlap when Java decode cycle times are small, hiding the decoding latency. Instructions issued in the shadow of branches also are processed much more quickly.

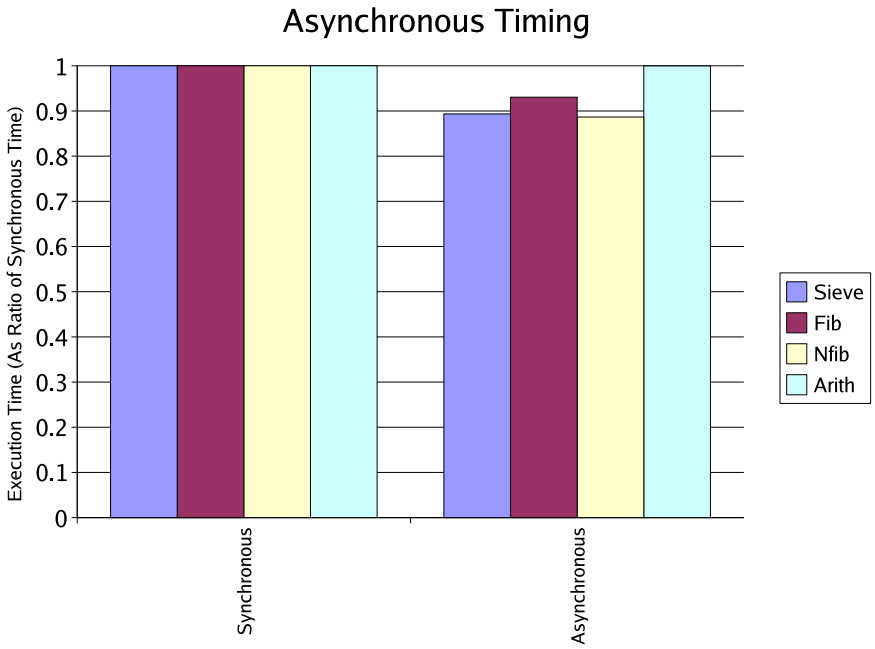


Figure 7.5: Asynchronous Decoder Performance.

Timings for the asynchronous decoder involve the assumption that for some cases it is possible to manage stack operations more efficiently. The reduction in cycle time for the Java decoder block has allowed benchmarks to run faster than in the synchronous pipeline. The timing assumption is based upon the synchronous

implementation not employing further parallelism to speed up the Java decoder. The problem with any improved synchronous implementation is that the whole pipeline would need to be clocked faster to see the improvement. Performance was improved in the asynchronous pipeline *without* without modifying any other pipeline stages.

An asynchronous approach has provided the benefit of a reduced complexity implementation, using a simple serial stack management unit, with an increase in performance over an equivalent synchronous system. The serial approach also aims to reduce the power consumed. A more parallel Java decoder block is possible and is almost certainly necessary in a synchronous implementation. The details of power consumption and complexity need to be explored at the circuit level to clarify what performance benefits are achievable and at what power and area cost. Any performance increase possible will likely come at a high cost as managing stack operations would involve managing many more possible states. A more parallel decoder may however be a viable approach to improving performance in both the synchronous and asynchronous systems.

### 7.3.4 Conclusions

An increase in Java decoder efficiency of around 10% has been achieved with the new folding branch optimisation. Simulation results show that this can be improved by a further 10% by taking advantage of a self-timed decoder design.

Experiments with memory latencies showed that the current byte-code folding scheme, does not really provide a significant reduction in memory operations over the simple stack cache approach. Further reductions could possibly be achieved with an improved approach to folding.

## 7.4 Improving Byte-Code Folding

Folding using the existing stack cache makes efficient use of registers compared to a dedicated set of registers for local variables. A problem observed during simulations is that these variables do not necessarily remain in the cache for very long, being overwritten indiscriminately. The perfect variable cache model simulated, shows a further 10 to 15% can be taken off execution times if the cost of local variable accesses is minimised. In order to improve on this approach, local variables should be protected as much as possible in the stack cache.

To avoid overwriting local variables in the stack cache a table of local variables can be kept, as in the current folding decoder. This table can be checked before writing temporary operand stack values so that they are not overwritten. This approach requires that the operand stack to be cached in an arbitrary not necessarily sequential set of registers. This can be achieved by maintaining a second table of registers currently in use in the stack cache.

Duplication of local variables in the cache should also be avoided to improve the usage of registers. In the original folding scheme, this was encouraged, to prolong the existence of cached values. This would not be necessary in the new scheme, as locals are preserved due to flexible allocation of registers for the operand stack.

This new approach to caching the stack and local variables in a shared register space makes more efficient usage of any registers available for the cache. This is especially important in architectures with more registers, or in an extended execute unit made to support the Java decoder.

## 7.5 Summary

A series of branch optimisations has been presented, providing a 10% decrease in execution time for the Java benchmarks. This optimisation provides some relief from the problems caused by having a decoupled Java decoder, maintaining state early in the processor pipeline. It may be better to implement a system of interlocking between Java decode and execute when issuing branches, in order to reduce the excessive cost of decoding in the shadow of taken branches. This is shown most clearly in the Arith benchmark, where the goto byte-code takes an average of 15 Java decoder cycles to execute.

The self-timed Java decoder model, highlighted the varying complexity of Java decode operations over different byte-codes. This has shown that a self-timed Java enabled processor core can make efficiency gains over its synchronous counterpart, even using the same translation algorithm.

The proposed new folding solution, with separate shared allocation of stack and local variable registers makes much better use of available resources. Importantly the scheme would scale better where more registers are available, this was shown to be a drawback of the current folding approach. This technique should get closer to the figures achieved with the perfect variable cache while retaining a low implementation cost.

In order to improve performance further, over the suggested methods in this chapter, Java support is really needed in the execute unit. The philosophy of being able to extend existing processors with a portable Java decoder extension brings the limitation of not being able to observe execution state. This is most apparent when dealing with branches. The shadowing issues determine that the stack and local variable caches must be flushed to memory on all branches, so they are not corrupted by shadowed byte-codes. If the execute unit maintained the state of the cache, the flushes could be avoided as shadowed operations would not be of the correct instruction colour. If flushes were removed, the number of memory operations for loading and storing temporary stack values and local variables would be reduced dramatically.

# Chapter 8

# Conclusion

Accelerating Java execution in embedded systems is important due to its widespread use in many application areas. Hardware support for Java is important for systems with small memory footprints and requiring low power operation. They can provide increased execution efficiency, while maintaining a compact virtual machine infrastructure. Areas where Java is a standard in embedded systems include: mobile phones, smart-cards and next generation video disc players.

Implementing acceleration as an extension to a processor architecture is an low cost approach taken in the ARM architecture. This thesis has shown that the basic decoding system can be extended to improve performance making better use of registers in RISC load store architectures. It has also been shown that a self-timed approach to processor design can be leveraged in this application area, due to the nature of Java byte-code to RISC instruction translation.

Investigation into the nature of Java aware processor pipelines was supported by the implementation of a custom simulation environment. This allowed for experiments with varying synchronous and asynchronous timing models. The simulation system devised provided support for these timing models but at simulation rates two-orders of magnitude faster than the gate level simulations of the initial hardware decoder design.

## 8.1 Architectural Simulation

Implementing a system level architectural simulation model provided support for rapid construction, exploration and evaluation of Java decoder designs. Flexibility was provided in terms of modelling timing for self-timed and synchronous models.

### 8.1.1 Timing Model

A simplified discrete event simulation environment was implemented, without the need for a centralised event queue. The event queue was replaced by a system of communicating Ada tasks, synchronised with a null message system when required. This distributed simulation environment is capable of running across multiple processors, although during tests ran best on uni-processor machines. The parallel nature however did provide increased throughput on multi-threaded Pentium 4 processors.

### 8.1.2 Profiling

The software based environment used to implement the processor model, also allowed for a sophisticated profiling system to be integrated with the model. This allowed for the execution of byte-codes to be traced through the pipeline showing exactly which individual and groups of byte-codes were executed during a simulation run. Not only frequency of execution was recorded, but also the time spent executing different sequences. A system was devised to correctly attribute time spent executing byte-codes in the shadow of branches.

### 8.1.3 System Performance

Simulations of the fully profiled processor core provided typically simulated the execution of tens of thousands of byte-codes per second. This is a vast improvement over simulations of the JASPA hardware which proceeded at a rate of under ten per second.

## 8.2 Instruction Folding

### 8.2.1 JASPA

Initial experiments with JASPA showed that there were weaknesses with the stack cache translation system. It did not take advantage of the fact that local variables loaded into registers were frequently not re-used when already cached.

### 8.2.2 Improving the Stack Cache

Improvements to the stack cache system were implemented to keep track of local variable values in the cache for re-use by other byte-codes. Further performance gains were achieved through reducing the penalty associated with issuing branches. These improvements led to a decrease in execution times of around 10% over a set of simple benchmarks.

A new folding mechanism has been proposed, which will make better use of the available registers, storing cached locals for as long as possible. This can be achieved by allocating registers for the operand stack dynamically, around registers known to contain local variables. It is predicted that this approach will be of benefit in processors with more registers available for the stack and local cache.

To support Java execution in a processor pipeline with the best possible efficiency, support in the execution unit is needed. Tests run throughout this thesis have shown that problems with branch state, and the need to flush register caches at branches puts a limit on what can be achieved with a decoupled Java decode unit. If stack and local caching was handled in the execute unit, flushes could be avoided, as shadowed instructions issued by the Java decoder would be ignored.

### 8.2.3 Asynchronous Design

Although the asynchronous folding decoder model was showed to give a further 10% improvement in performance the main advantage to this implementation approach is ease of system composition and design flexibility. A self-timed Java decoder is free from worst-case timing constraints allowing much more exotic optimisations to be carried out. This would simplify implementation of different folding algorithms, and allow for performance trade-offs where complex folding operations could take longer than the average decode cycle. Such optimisations are simply not possible in a synchronous implementation without stalling for whole clock cycles.

## 8.3 Future Research

Areas of future research could include the following:

- Exploration of the proposed folding algorithm described at the end of Chapter 7. Is this approach more scalable, or do flushes at branches limit the efficiency of any caching scheme.

- Feedback of branch status from the execute unit to the Java decoder could be looked at as a method of retaining cached operand stack and local variables between branches. Such communications could be decoupled from the execute stage, so not to interlock the pipeline.
- Extensions for managing Java decoder state at the execute unit could potentially solve many of the problems with the current decoder model. Would this be a viable approach or is the state required as byte-codes are decoded?
- Java extensions could be tested with accurate models of other embedded RISC processor cores to assess suitability as a generic method for Java acceleration.
- Assessment of hardware implementation cost of the folding capable Java accelerator.

# Appendix A

# Simulation Data

The following graphs show the cumulative time spent executing individual byte-codes and groups of four byte-codes over the four benchmarks used in Chapter 6 and Chapter 7. This information was extracted from the byte-code profiler implemented as part of the simulated processor core, explained in detail at the start of Chapter 6.

The benchmarks, also described in Chapter 6, were run and detailed results collected for the following cores:

**JASPA** An implementation of the standard stack cache decoder used in JASPA, described in Chapter 4.

**Folding1** The extended stack cache decoder described in Chapter 6.

**Folding1-Branchopt** The extended stack cache decoder with branch optimisations described in Chapter 7.

Each core had uniform timing for pipeline stages, set at 100 simulation time units per cycle.

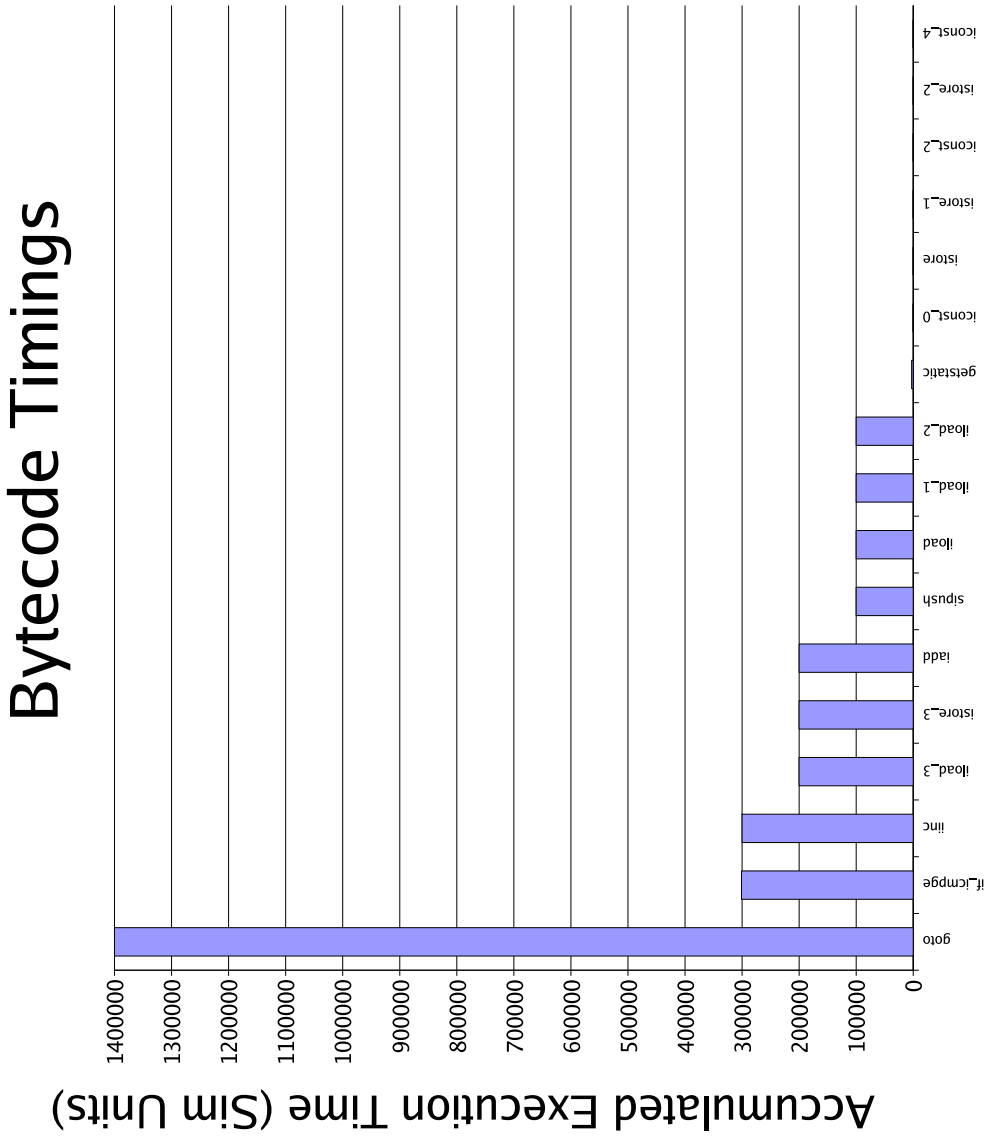


Figure A.1: Arith Benchmark on JASPA, Single Byte-code Breakdown.

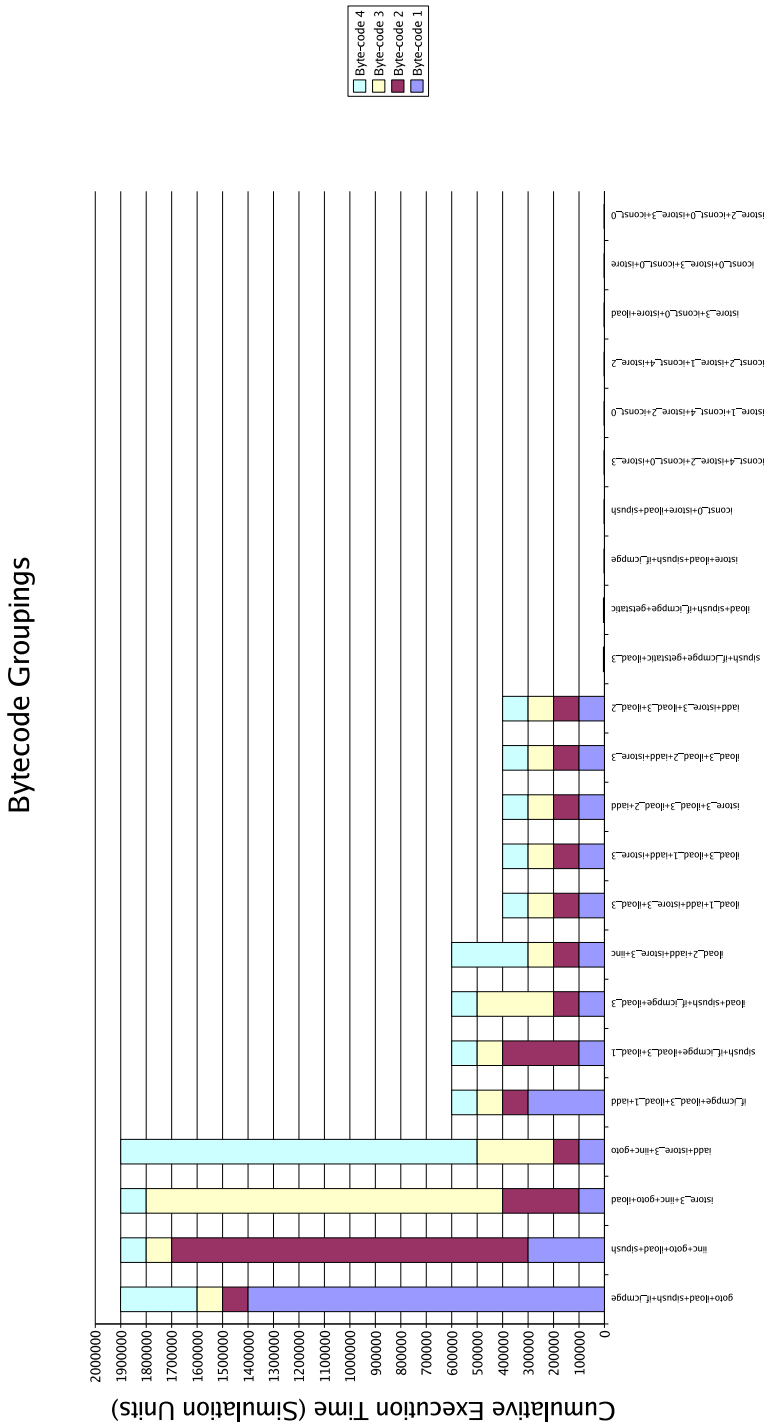


Figure A.2: Arith Benchmark on JASPA, Grouped Byte-code Breakdown.

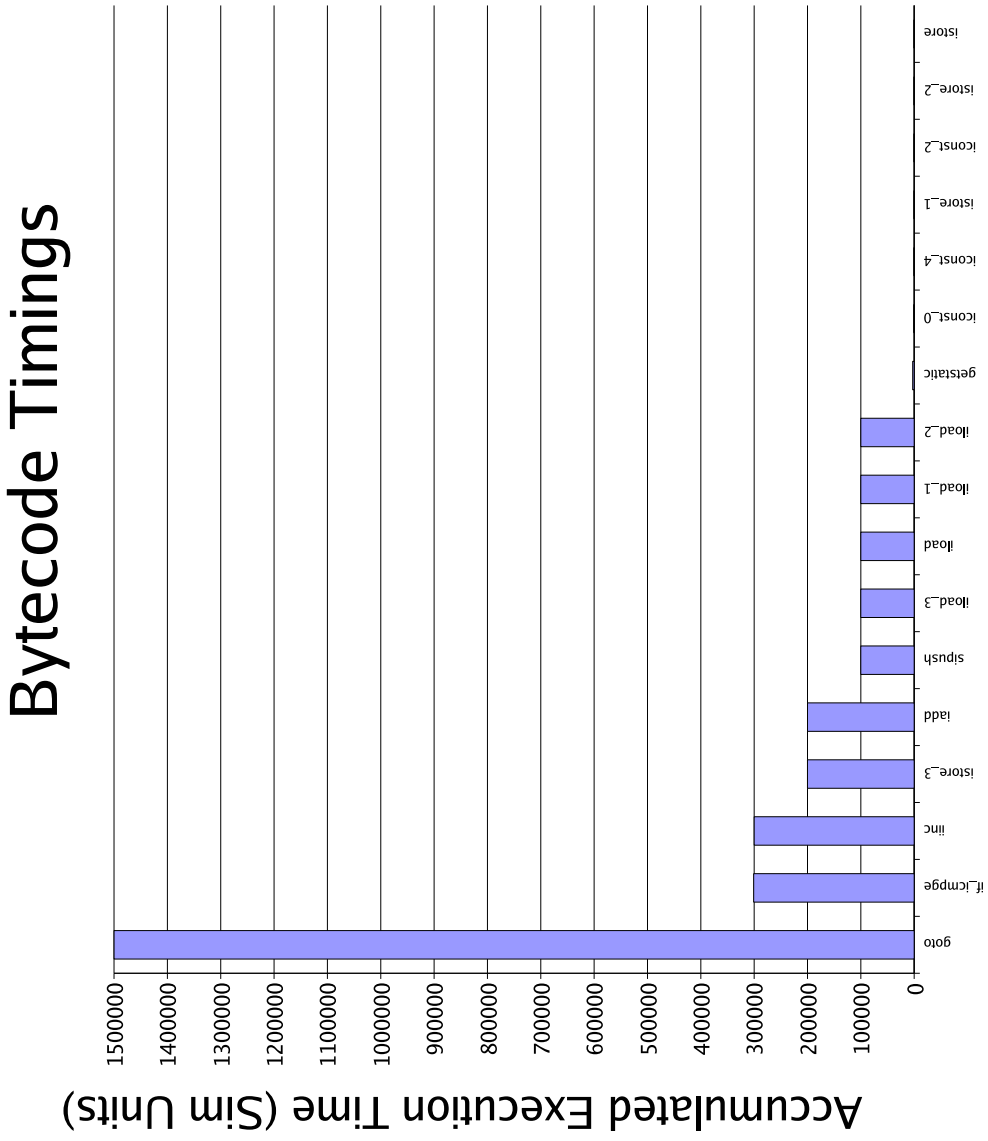


Figure A.3: Arith Benchmark on Folding1, Single Byte-code Breakdown.

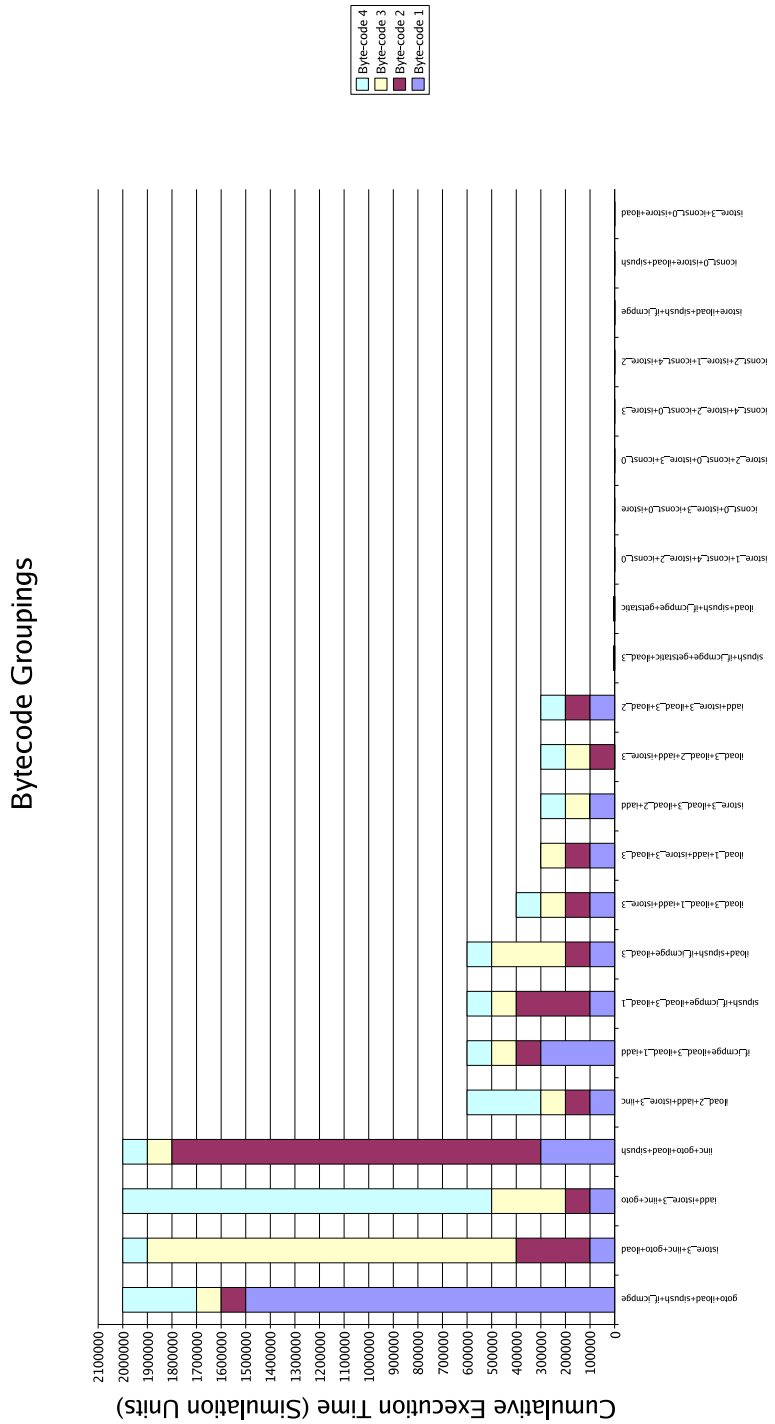


Figure A.4: Arith Benchmark on Folding1, Grouped Byte-code Breakdown.

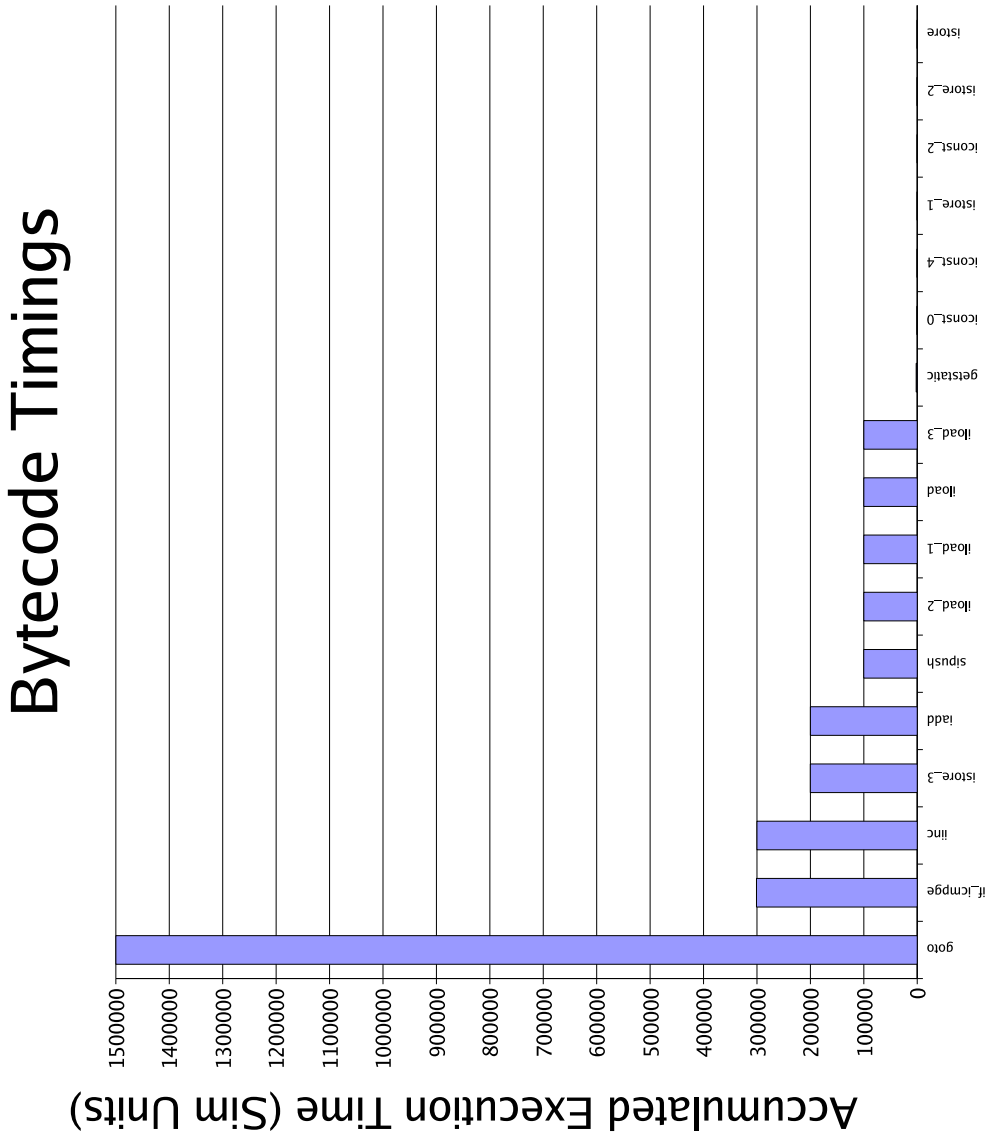


Figure A.5: Arith Benchmark on Folding1-Branchopt, Single Byte-code Breakdown.

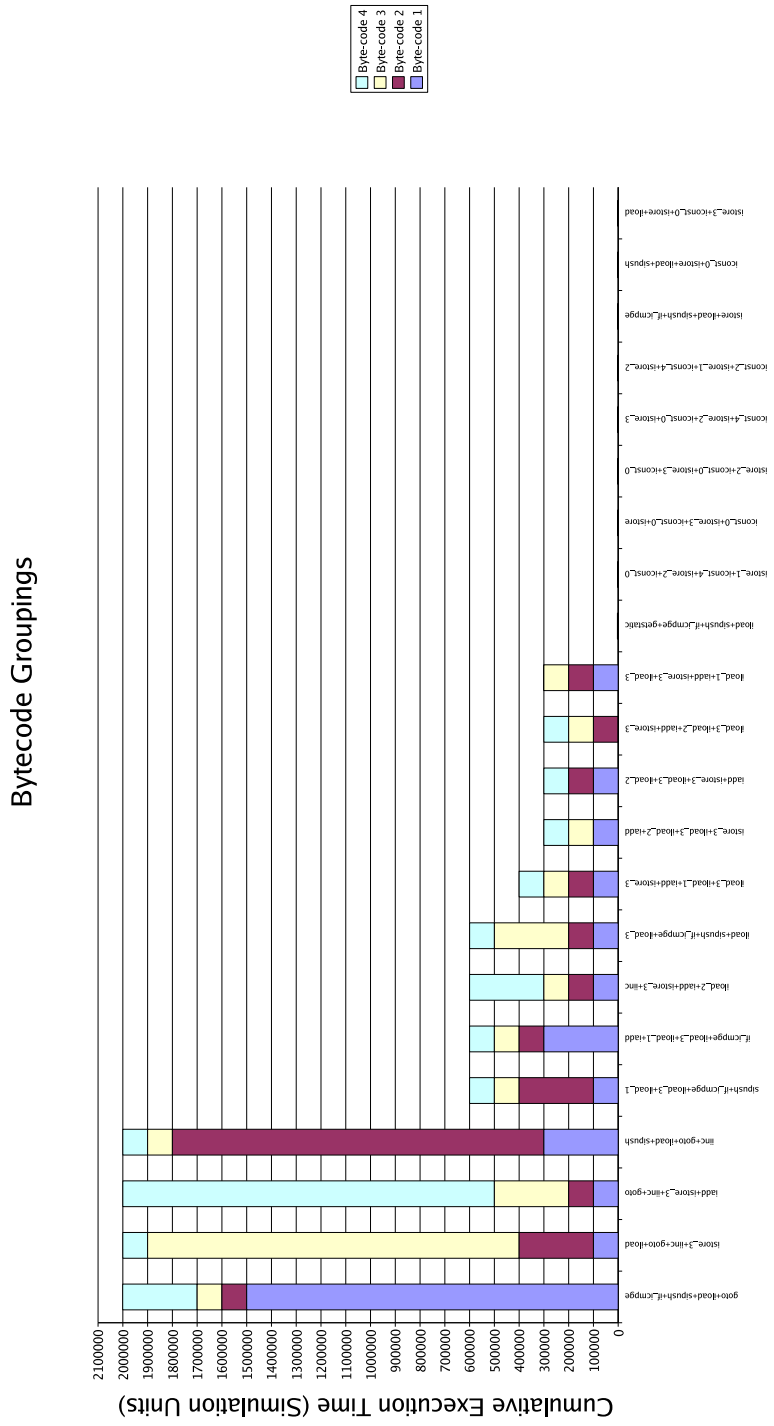


Figure A.6: Arith Benchmark on Folding1-Branchopt, Grouped Byte-code Breakdown.

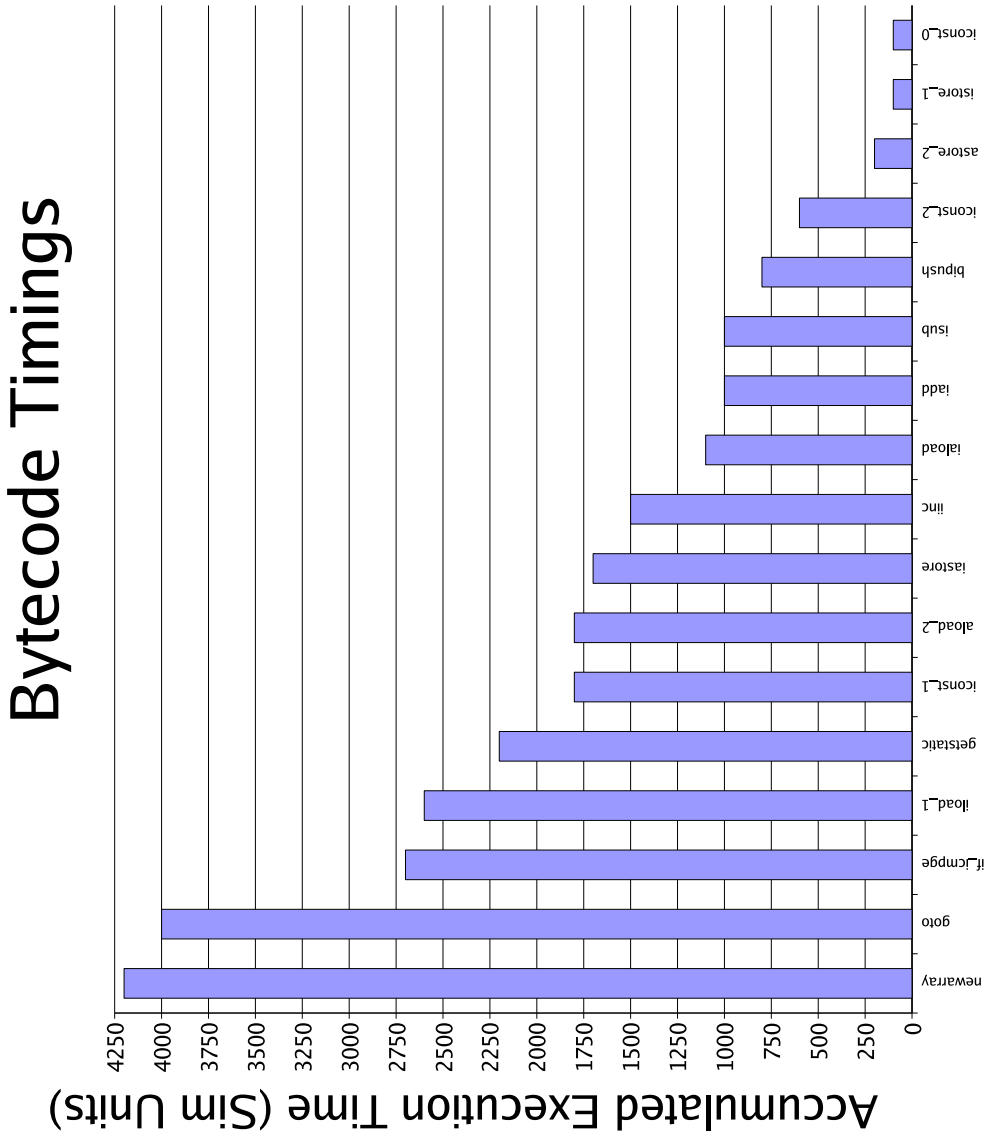


Figure A.7: Fib Benchmark on JASPA, Single Byte-code Breakdown.

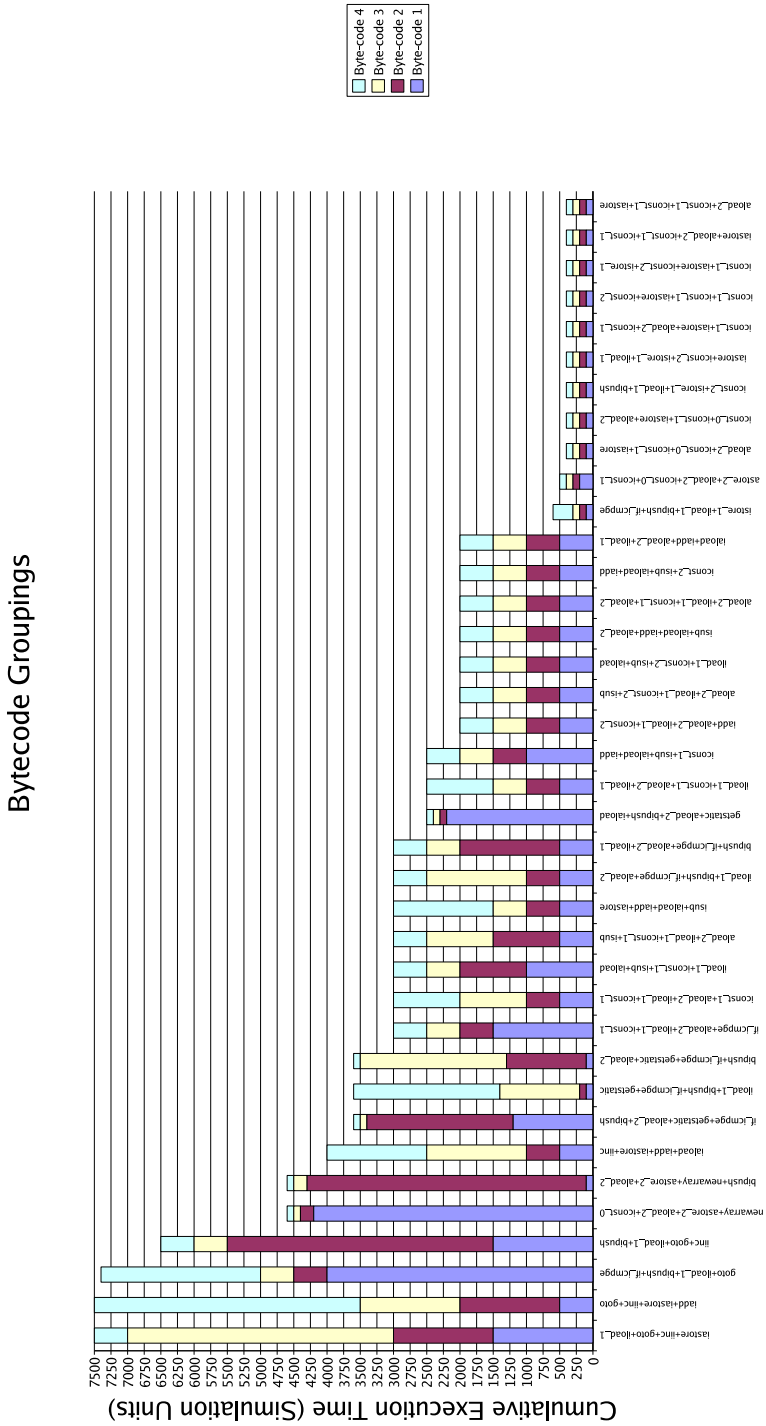


Figure A.8: Fib Benchmark on JASPA, Grouped Byte-code Breakdown.

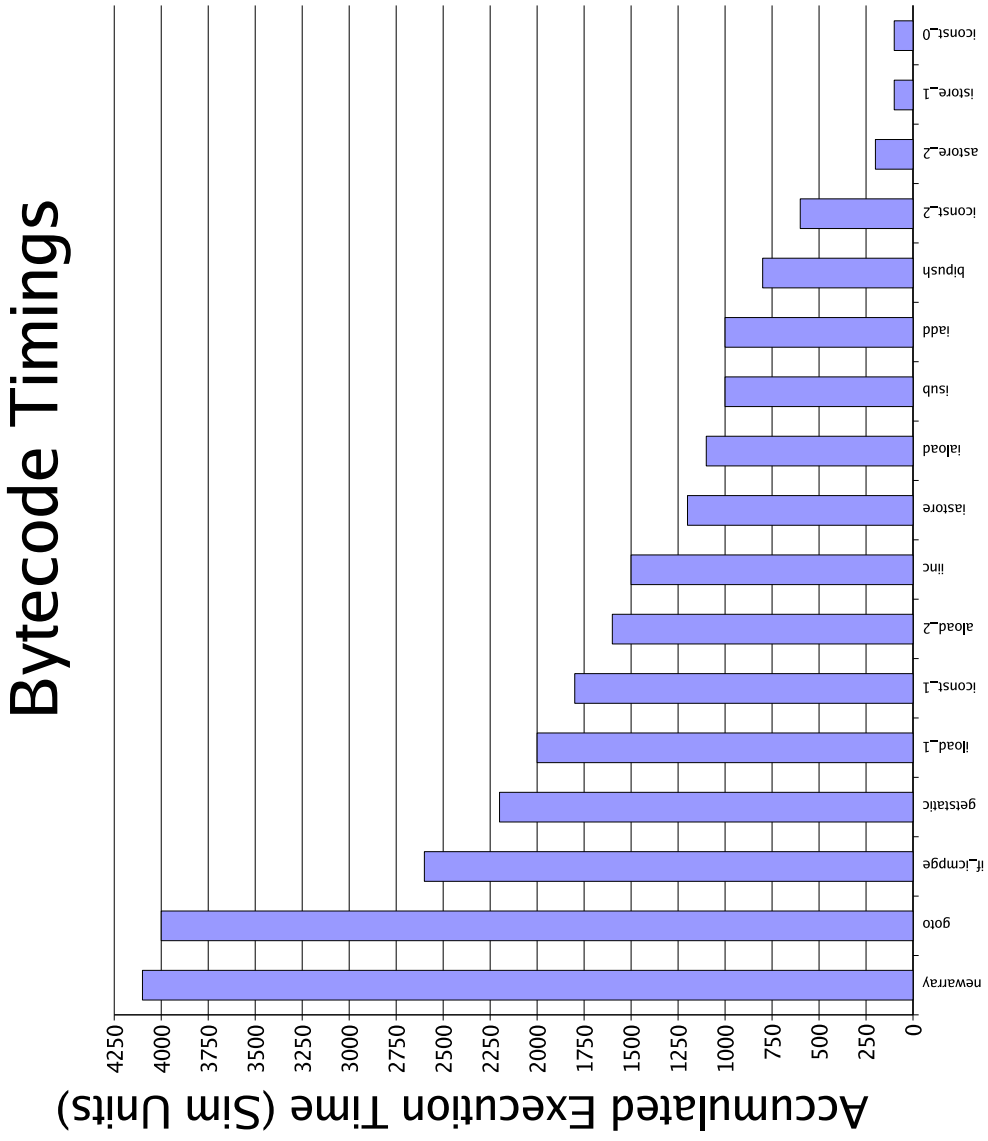


Figure A.9: Fib Benchmark on Folding1, Single Byte-code Breakdown.

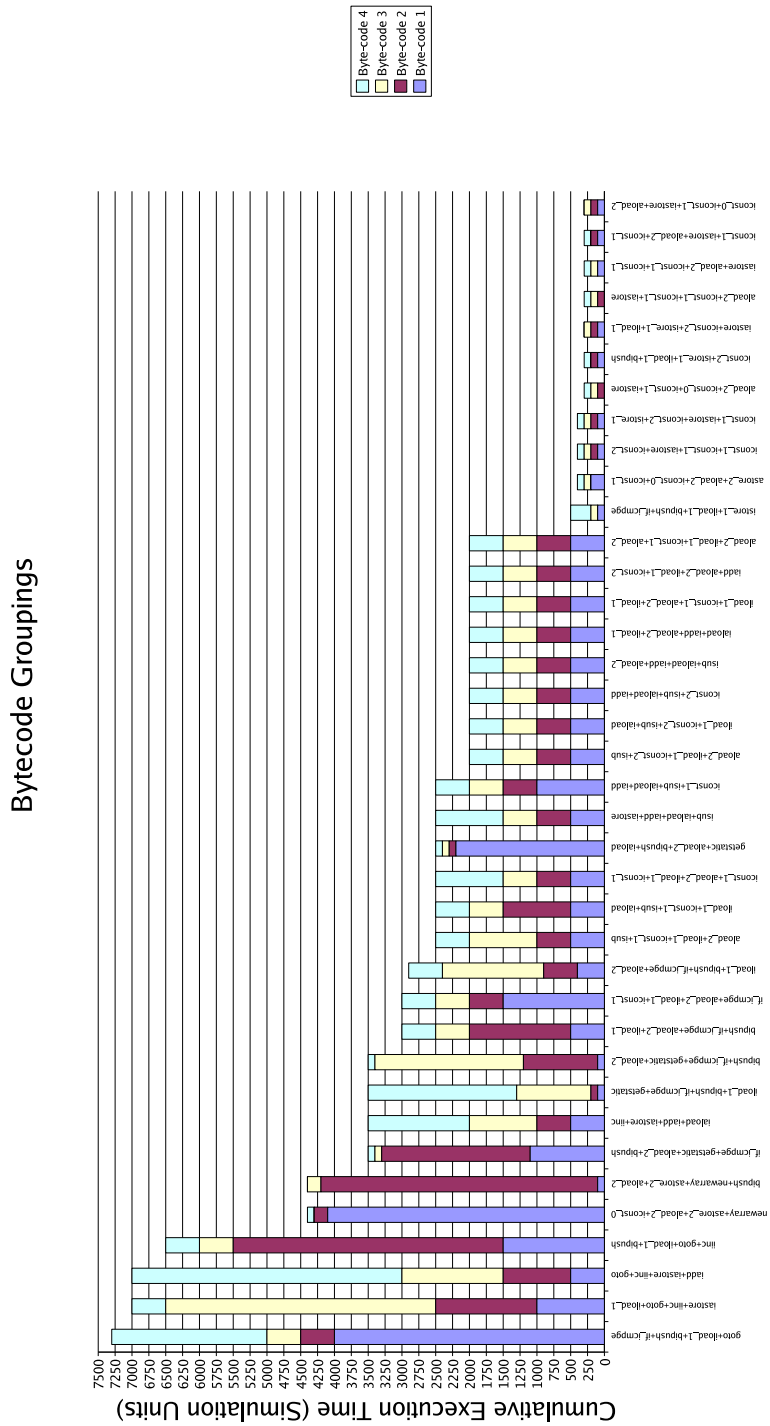


Figure A.10: Fib Benchmark on Folding1, Grouped Byte-code Breakdown.

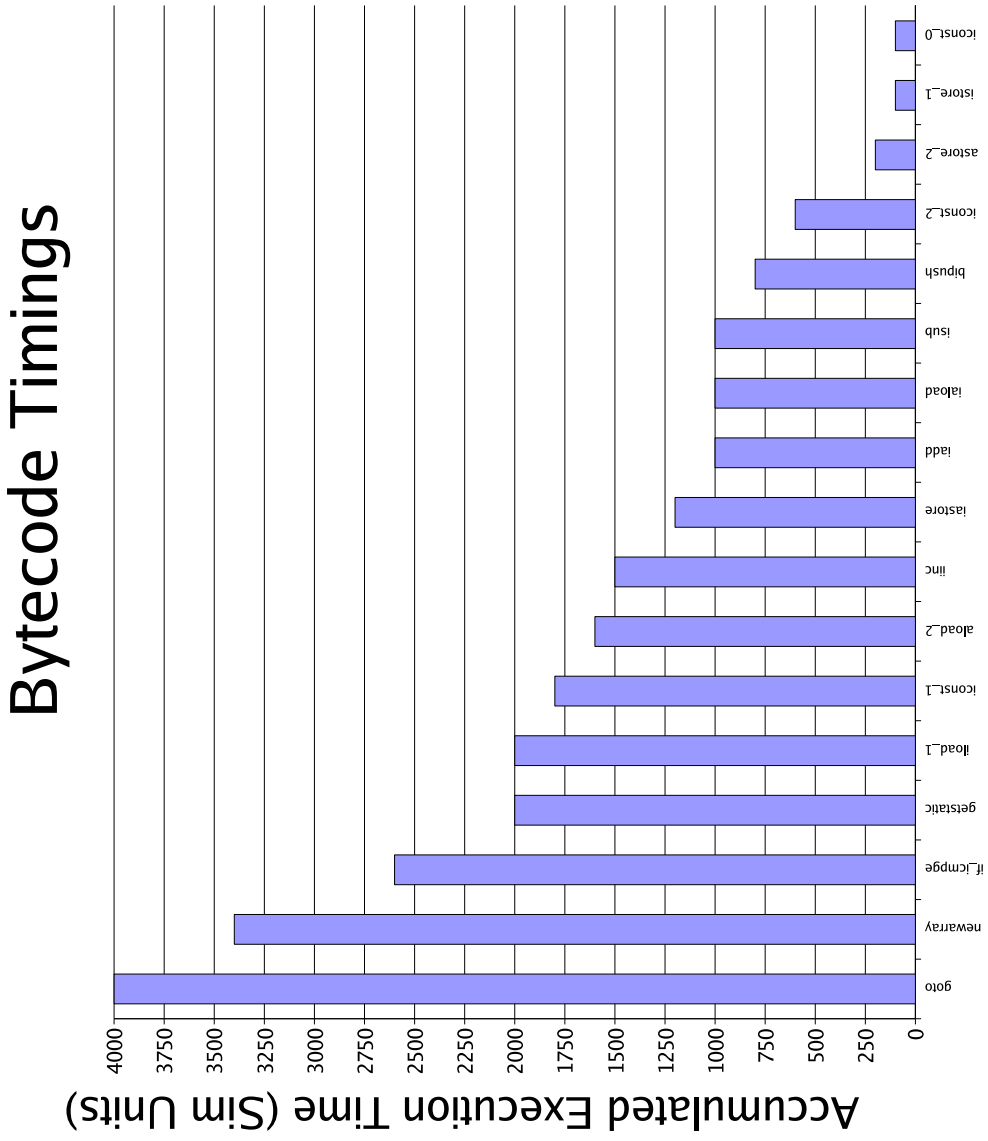


Figure A.11: Fib Benchmark on Folding1-Branchopt, Single Byte-code Breakdown.

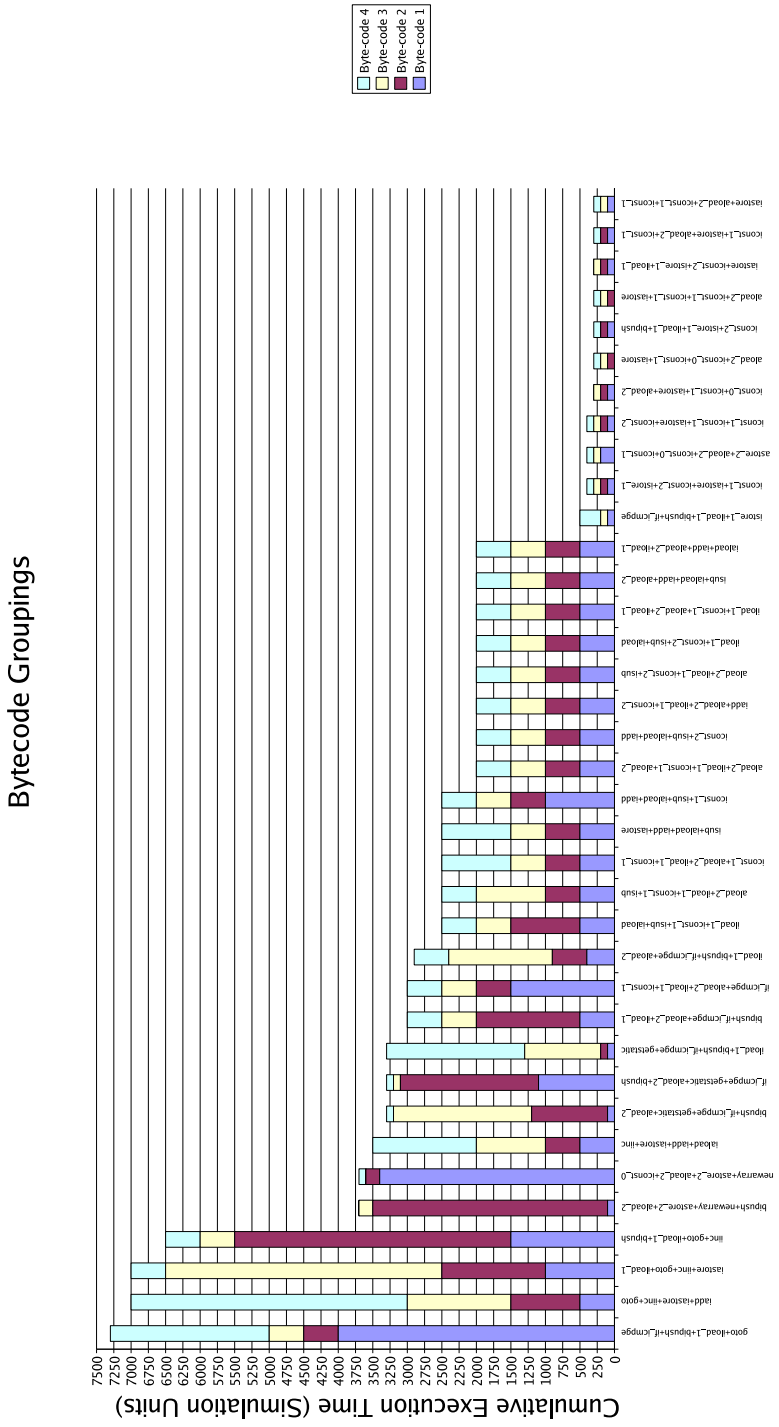


Figure A.12: Fib Benchmark on Folding1-Branchopt, Grouped Byte-code Breakdown.

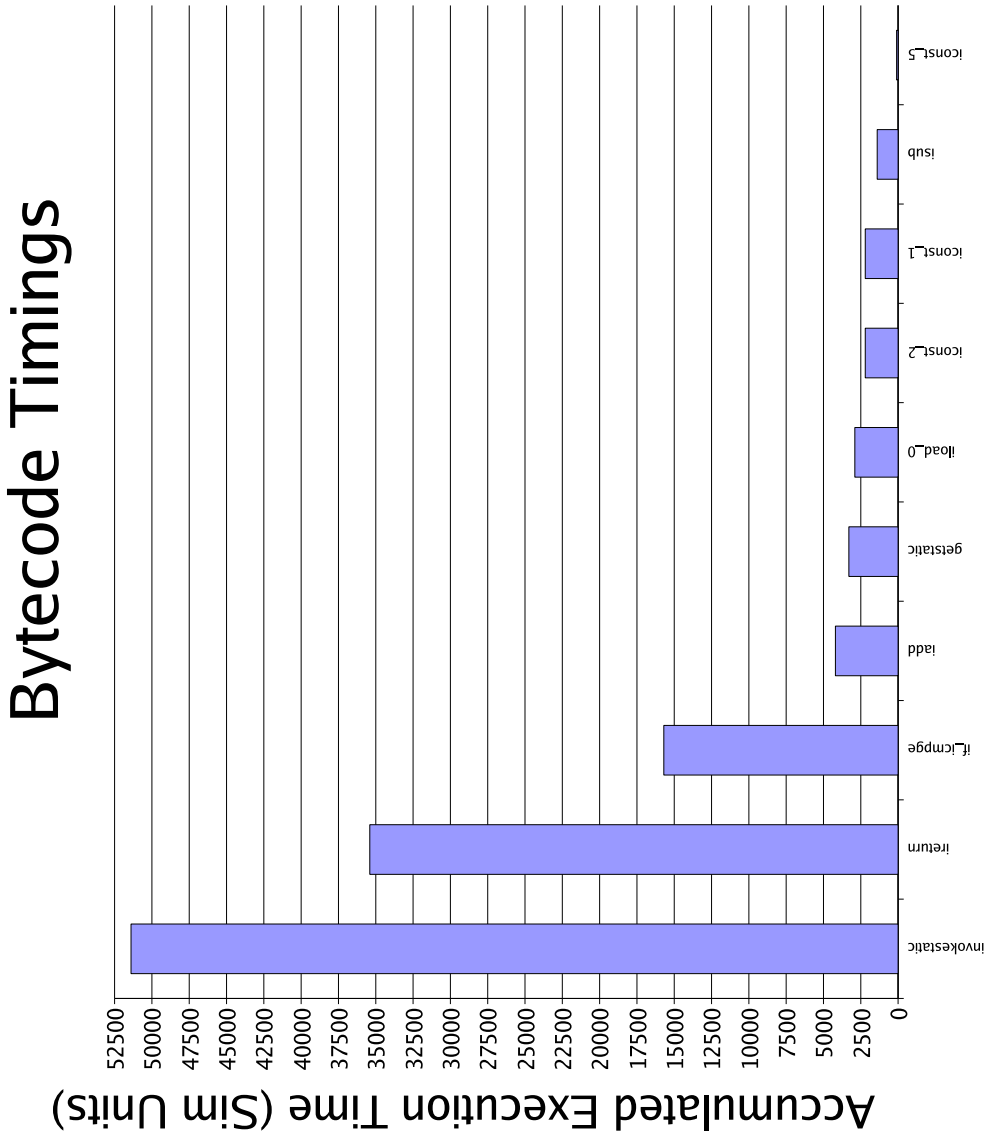


Figure A.13: NFib Benchmark on JASPA, Single Byte-code Breakdown.

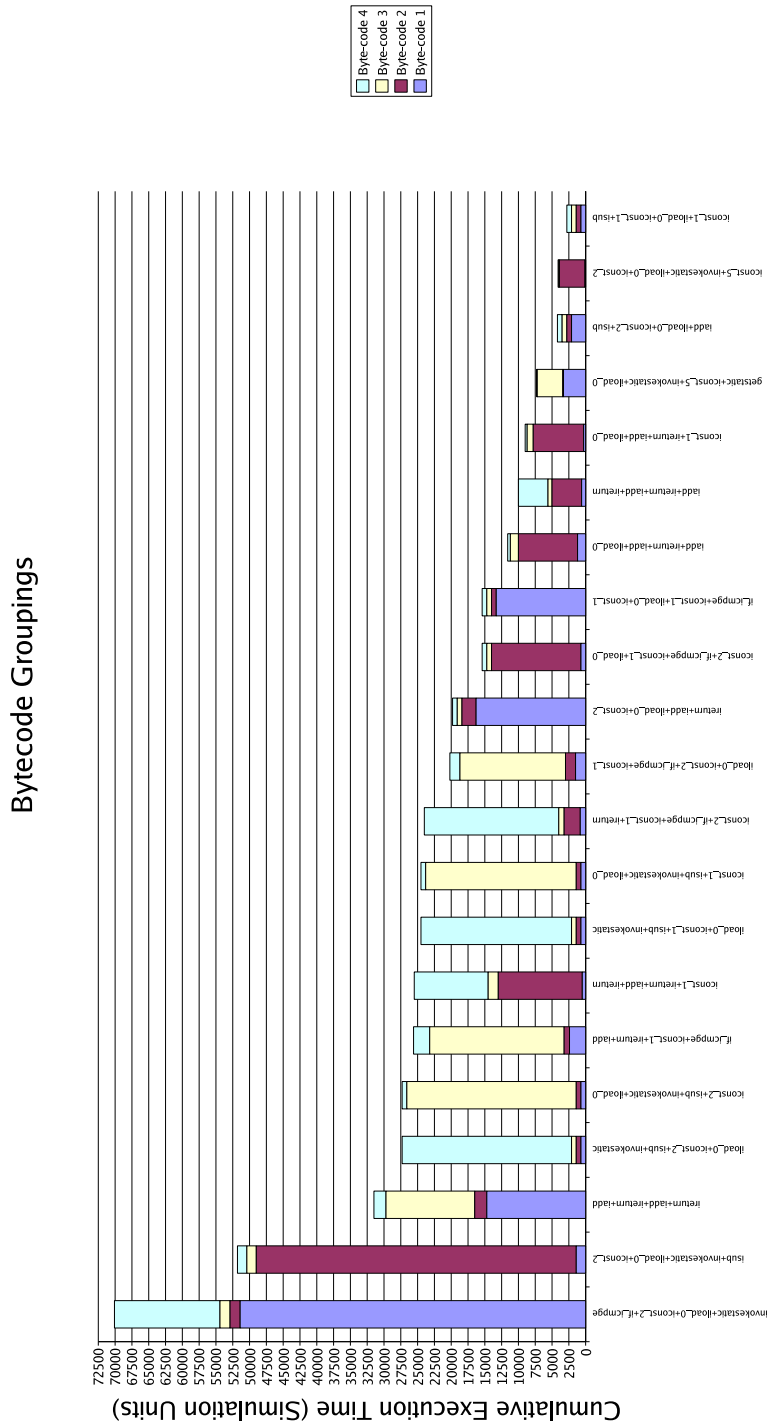


Figure A.14: NFib Benchmark on JASPA, Grouped Byte-code Breakdown.

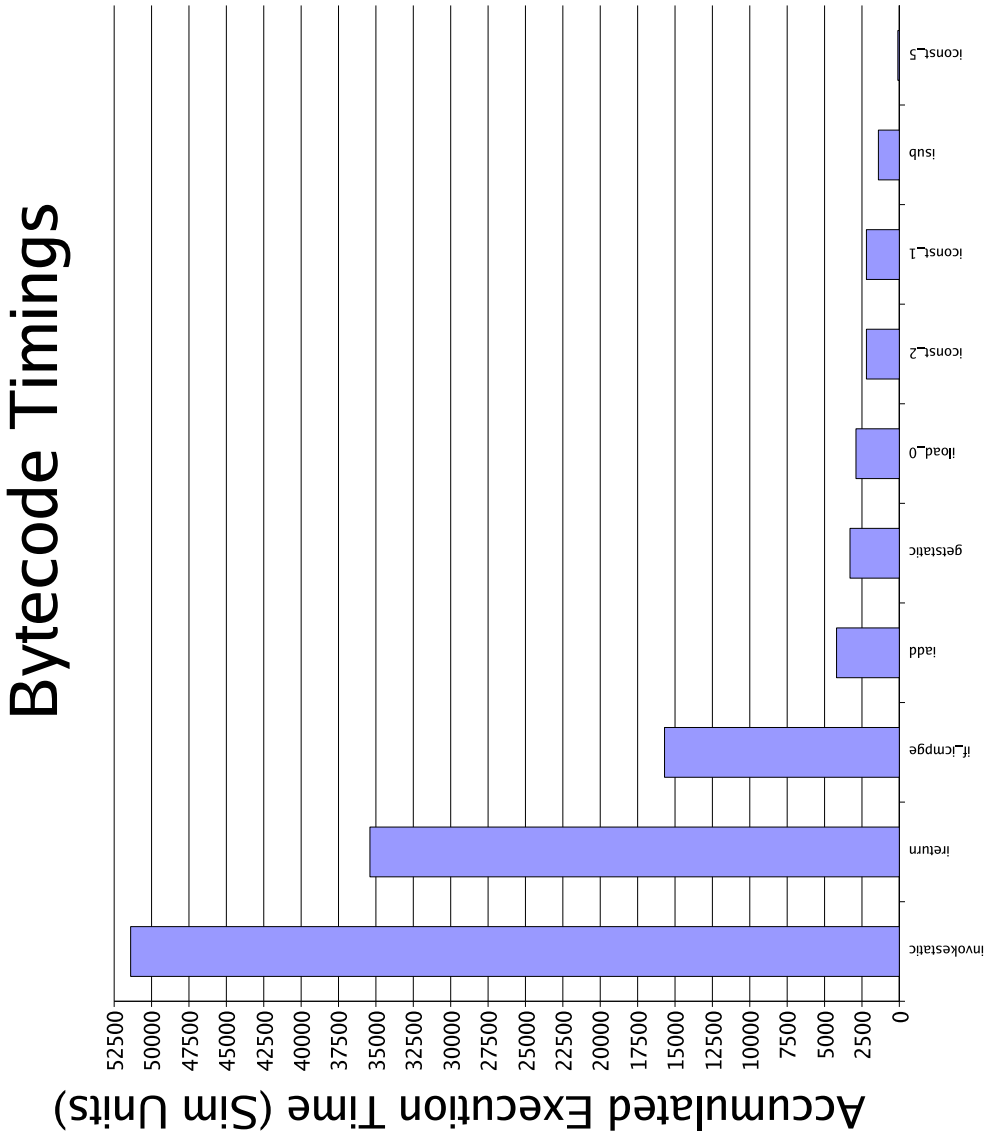


Figure A.15: NFib Benchmark on Folding1, Single Byte-code Breakdown.

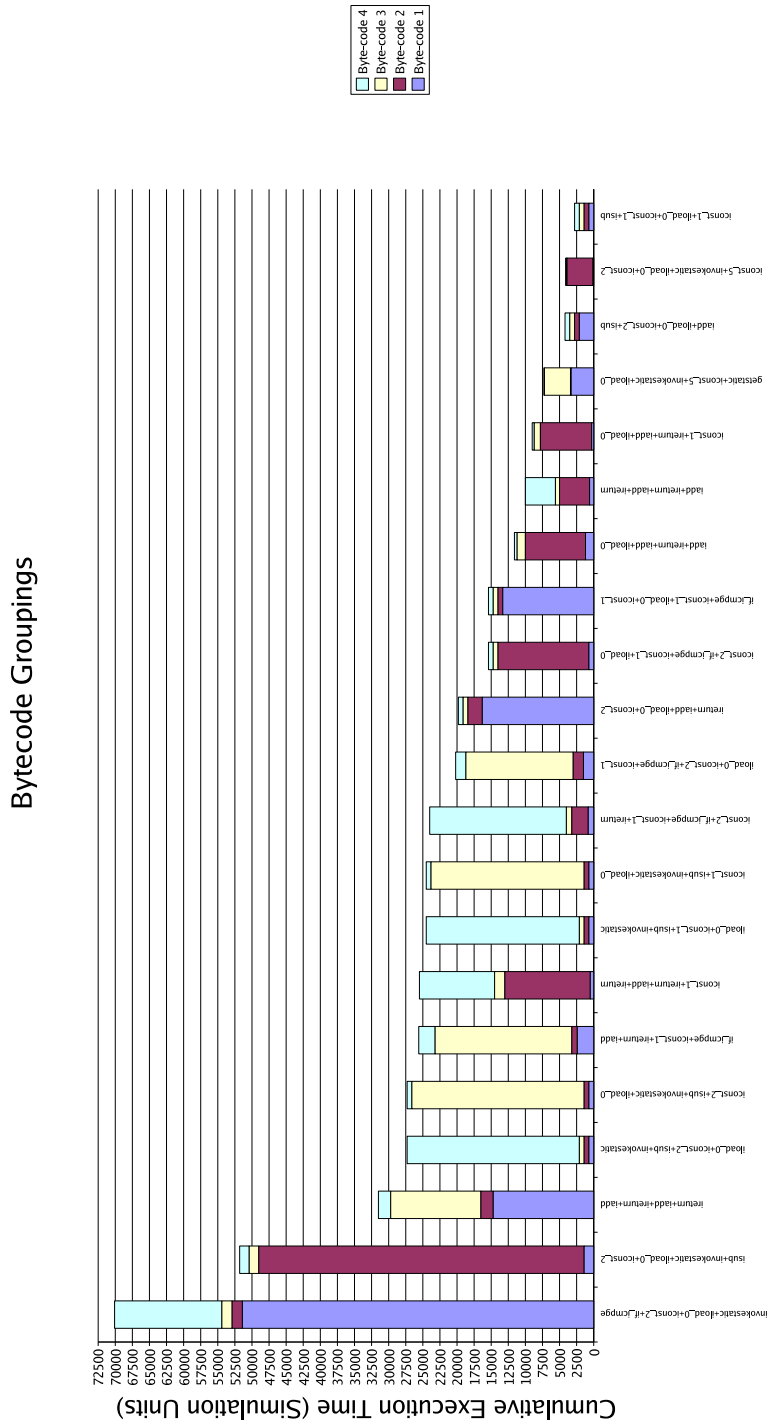


Figure A.16: NFib Benchmark on Folding1, Grouped Byte-code Breakdown.

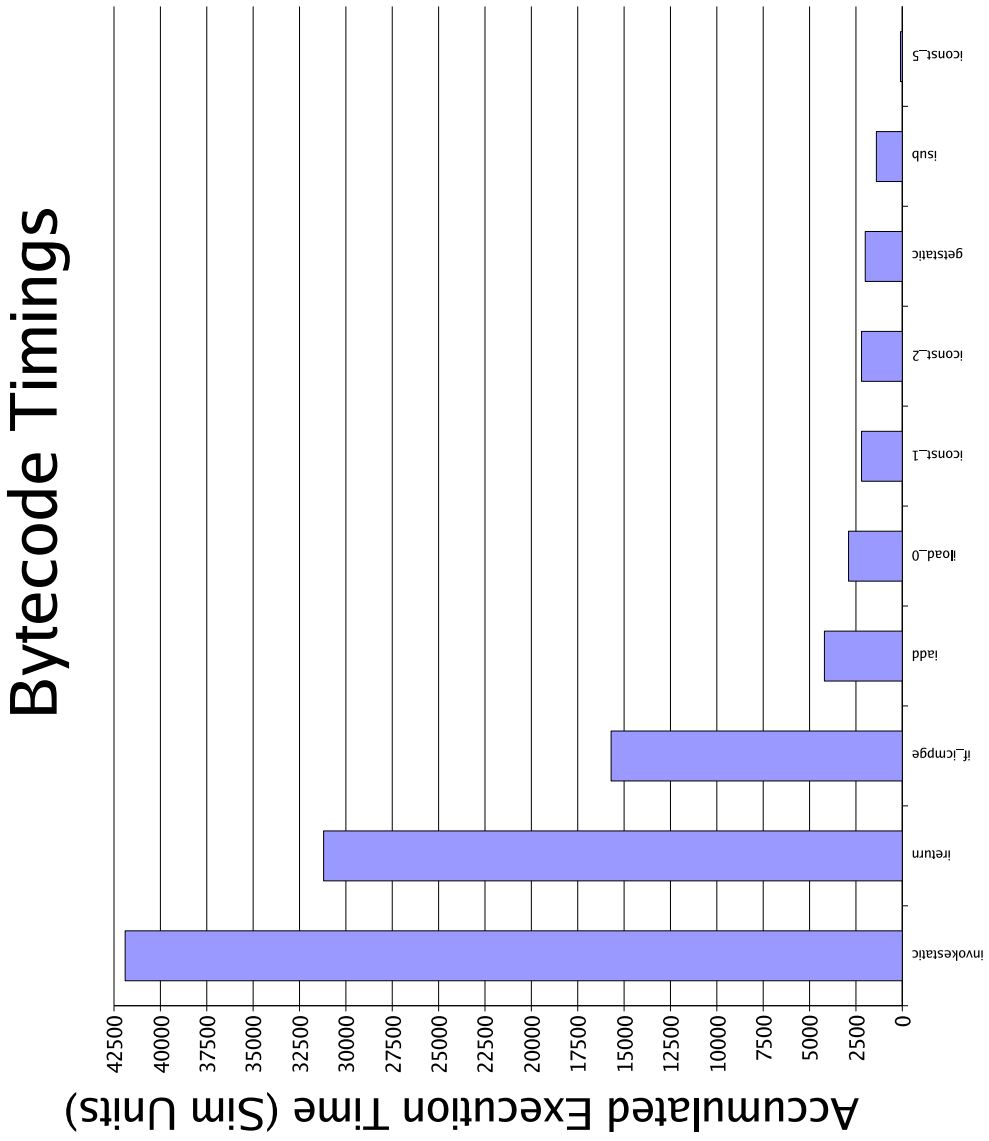


Figure A.17: NFib Benchmark on Folding1-Branchopt, Single Byte-code Breakdown.

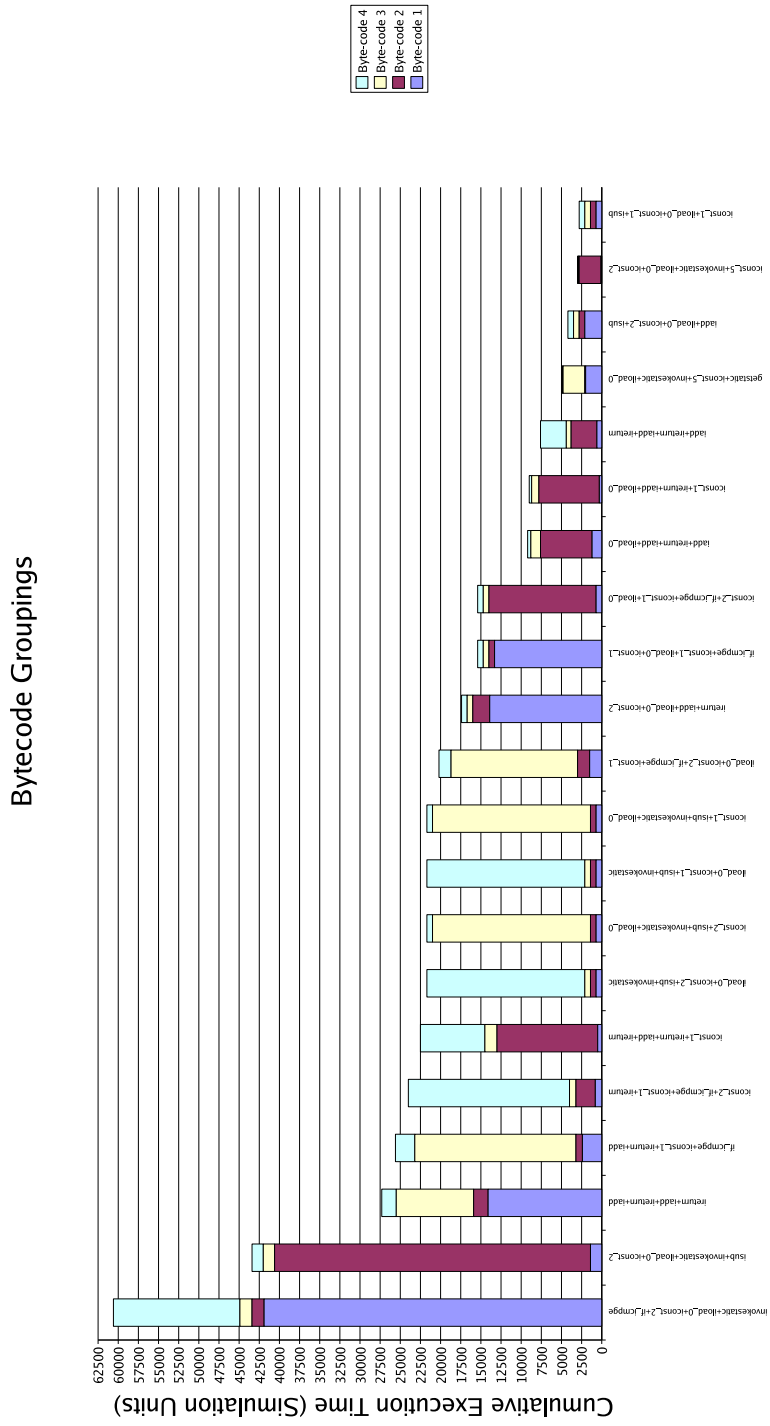


Figure A.18: NFib Benchmark on Folding1-Branchopt, Grouped Byte-code Breakdown.

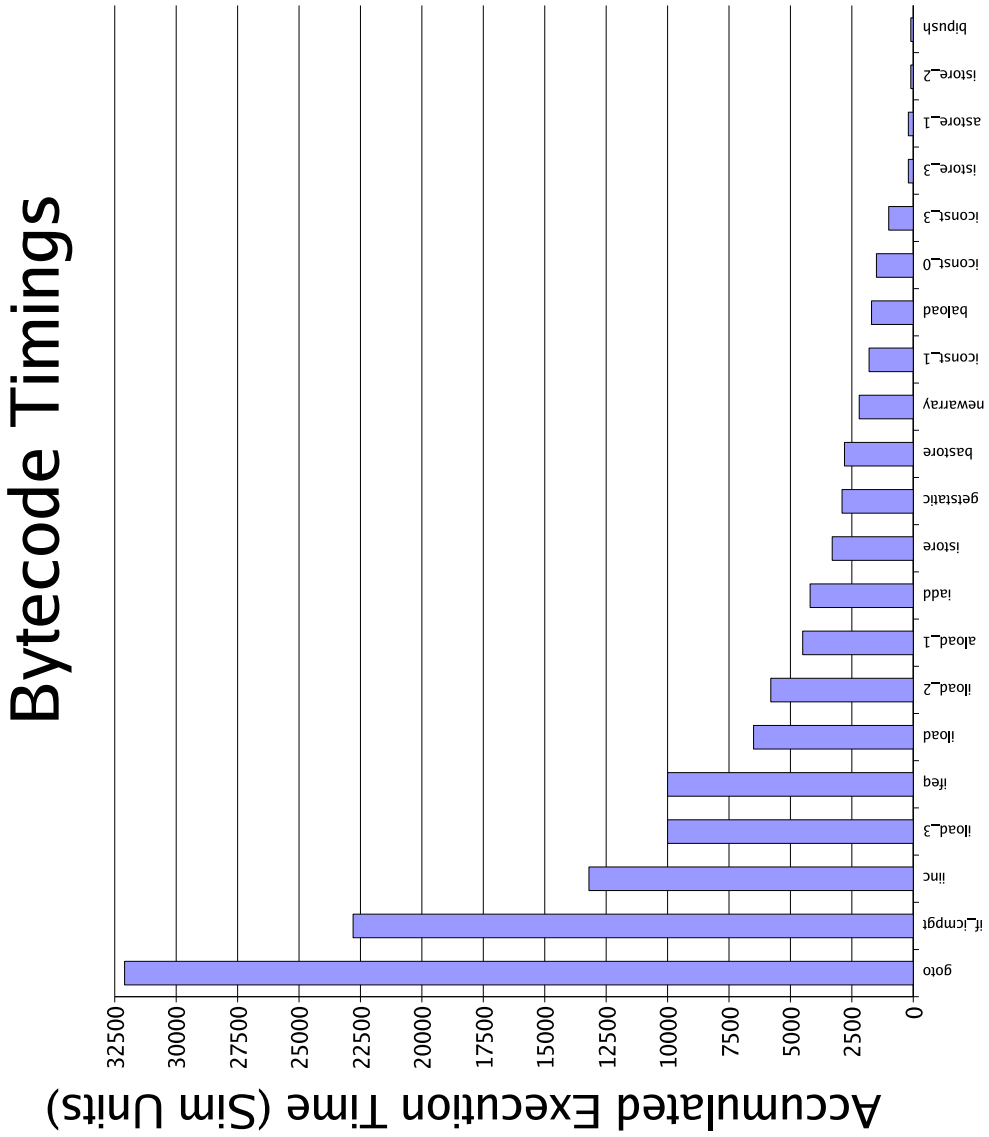


Figure A.19: Sieve Benchmark on JASPA, Single Byte-code Breakdown.

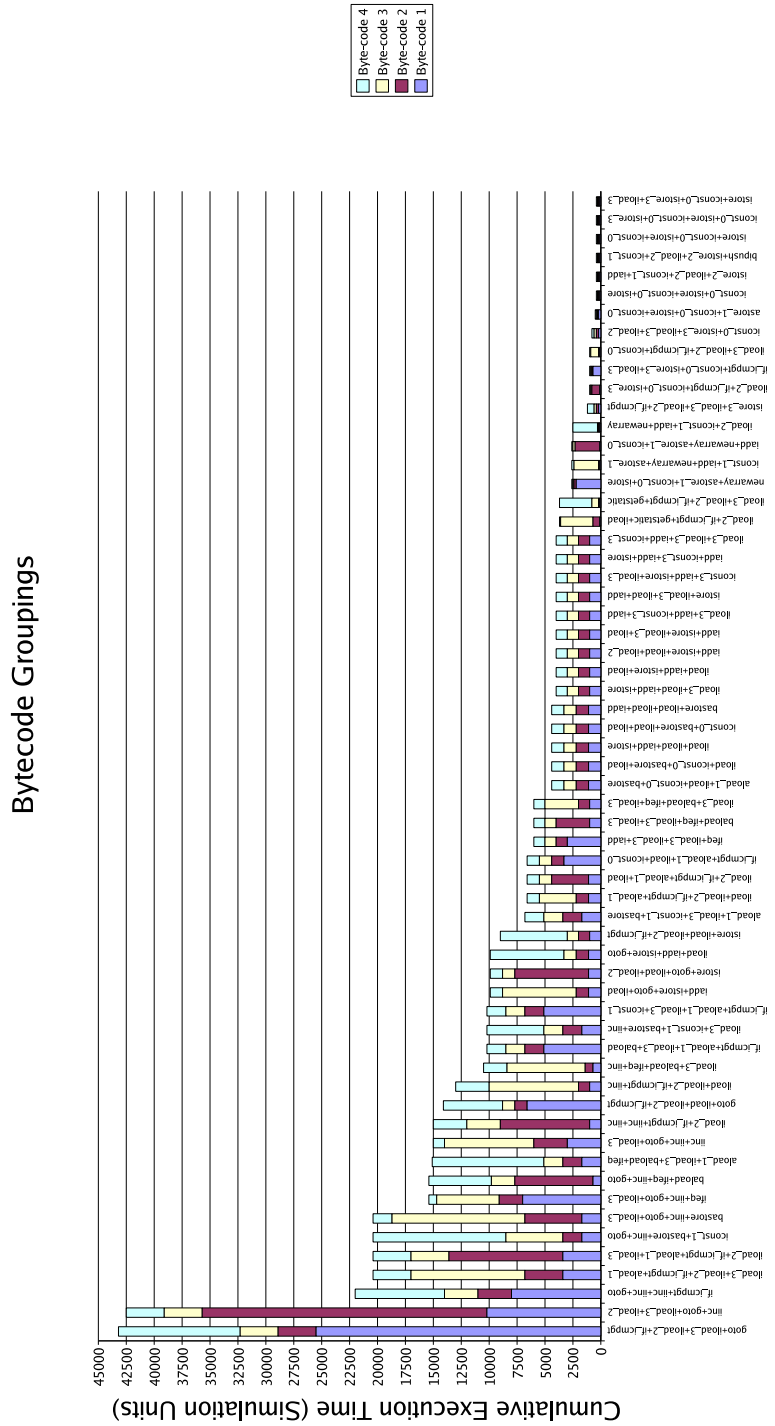


Figure A.20: Sieve Benchmark on JASPA, Grouped Byte-code Breakdown.

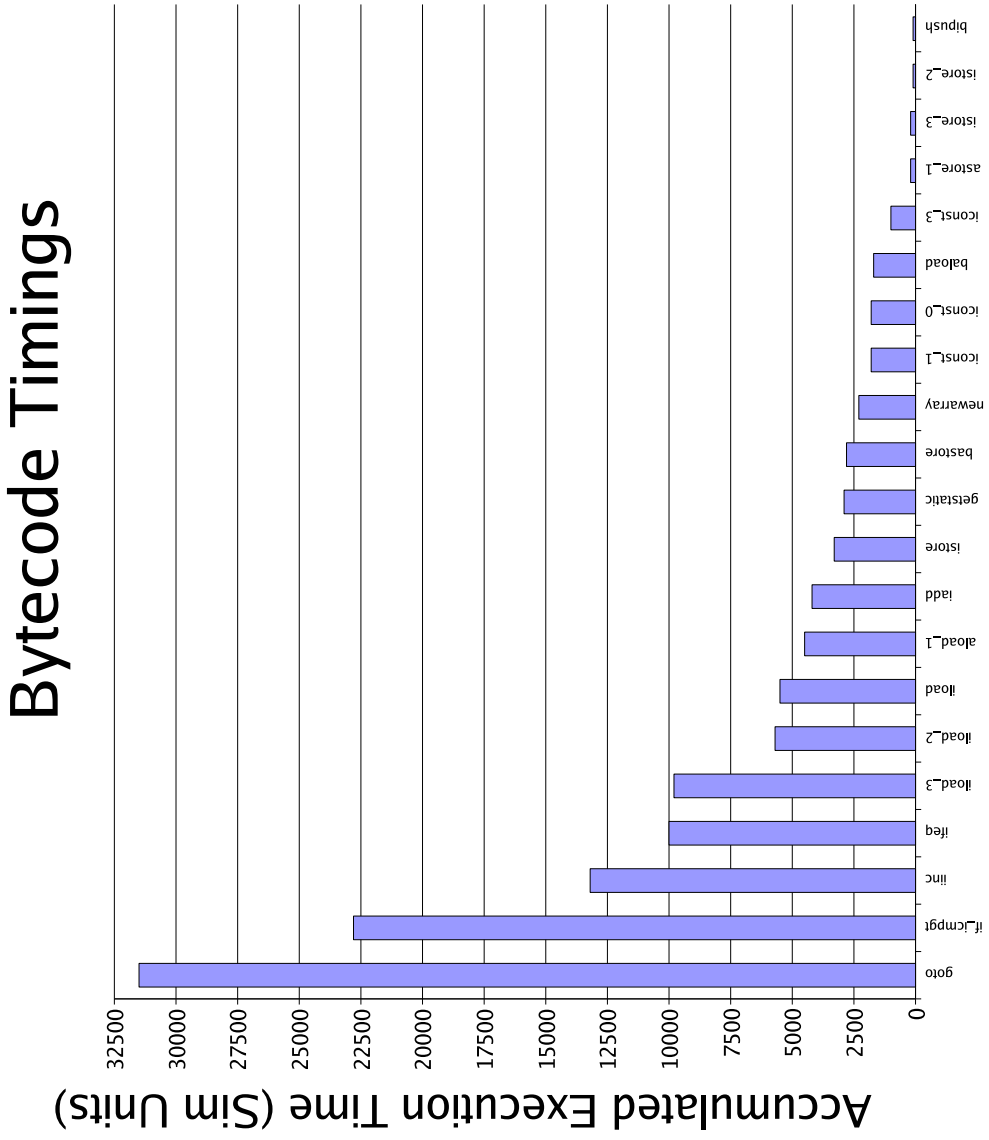


Figure A.21: Sieve Benchmark on Folding1, Single Byte-code Breakdown.

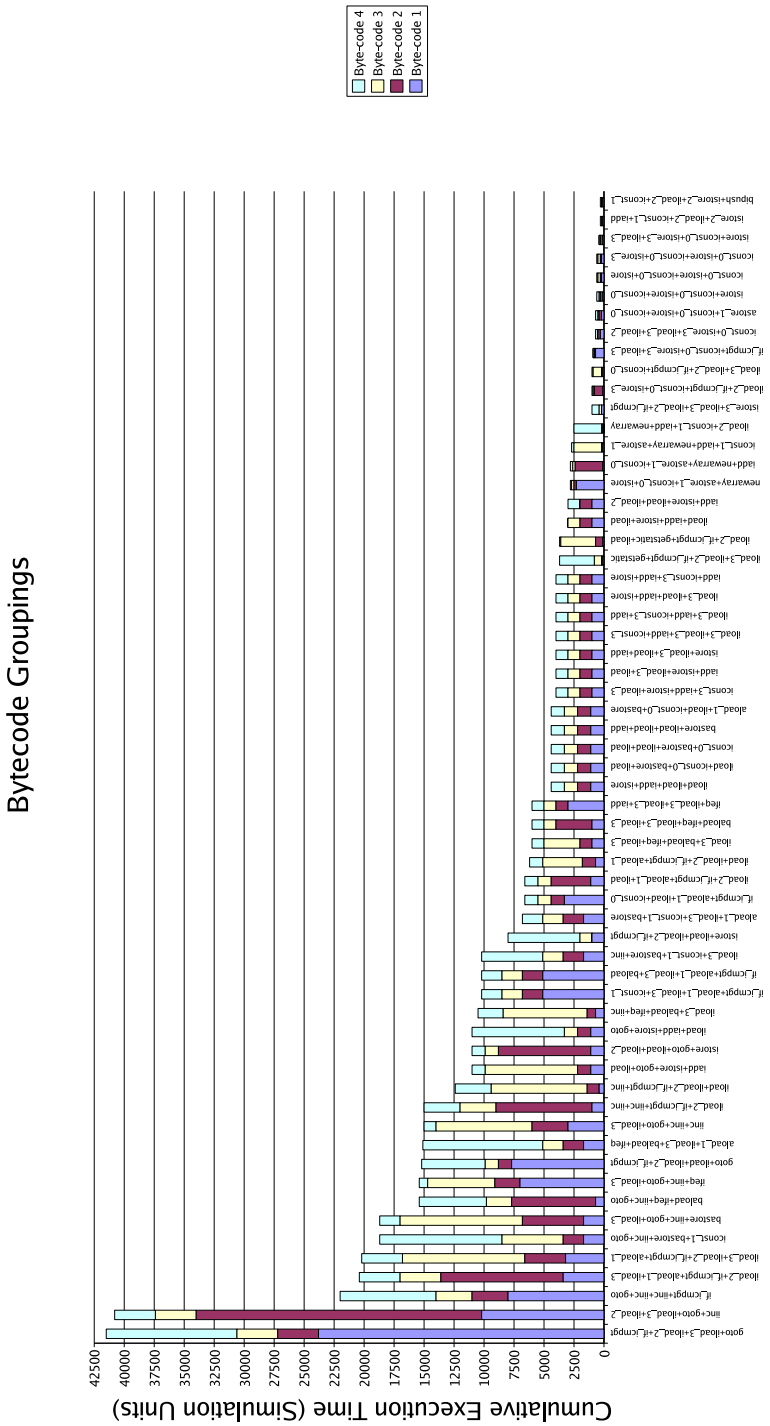


Figure A.22: Sieve Benchmark on Folding1, Grouped Byte-code Breakdown.

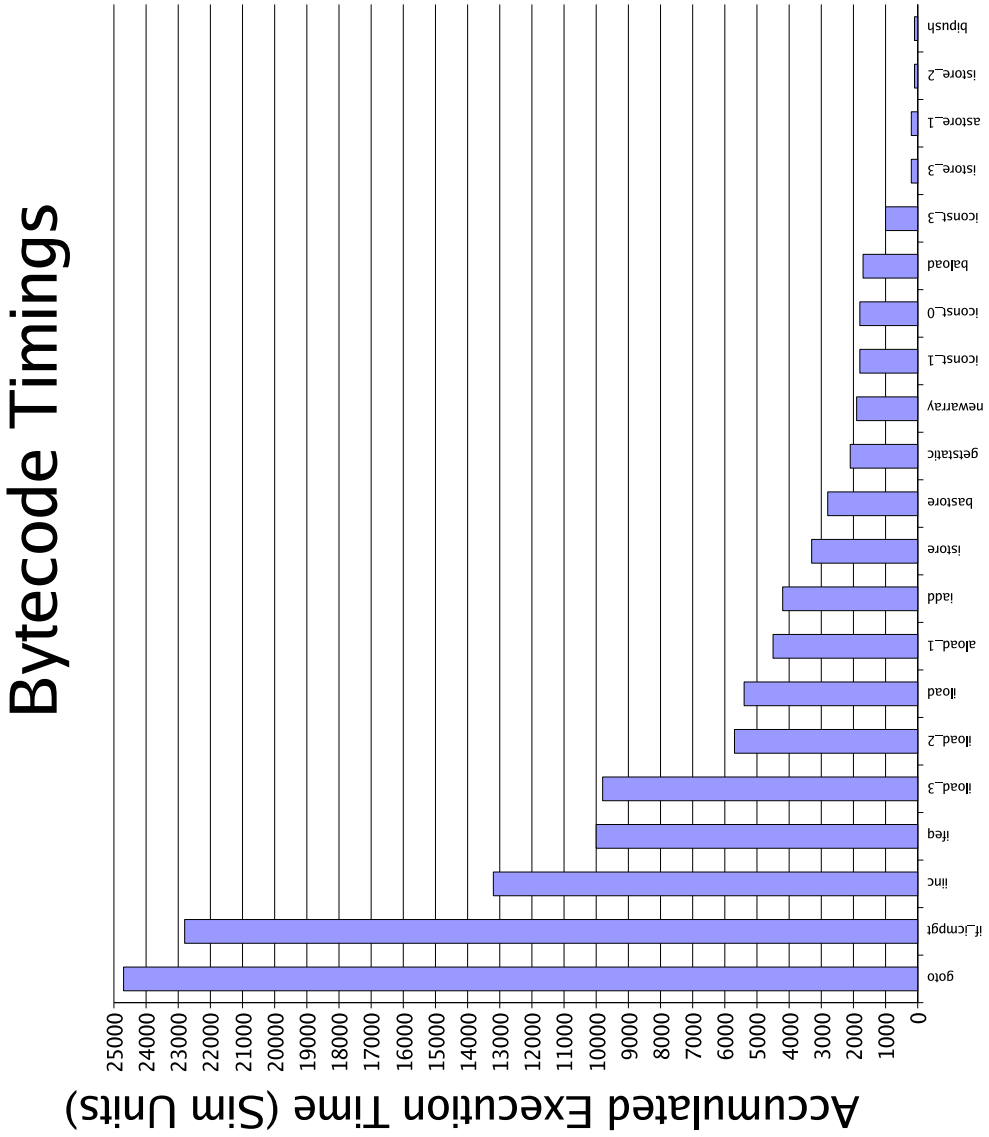


Figure A.23: Sieve Benchmark on Folding1-Branchopt, Single Byte-code Breakdown.

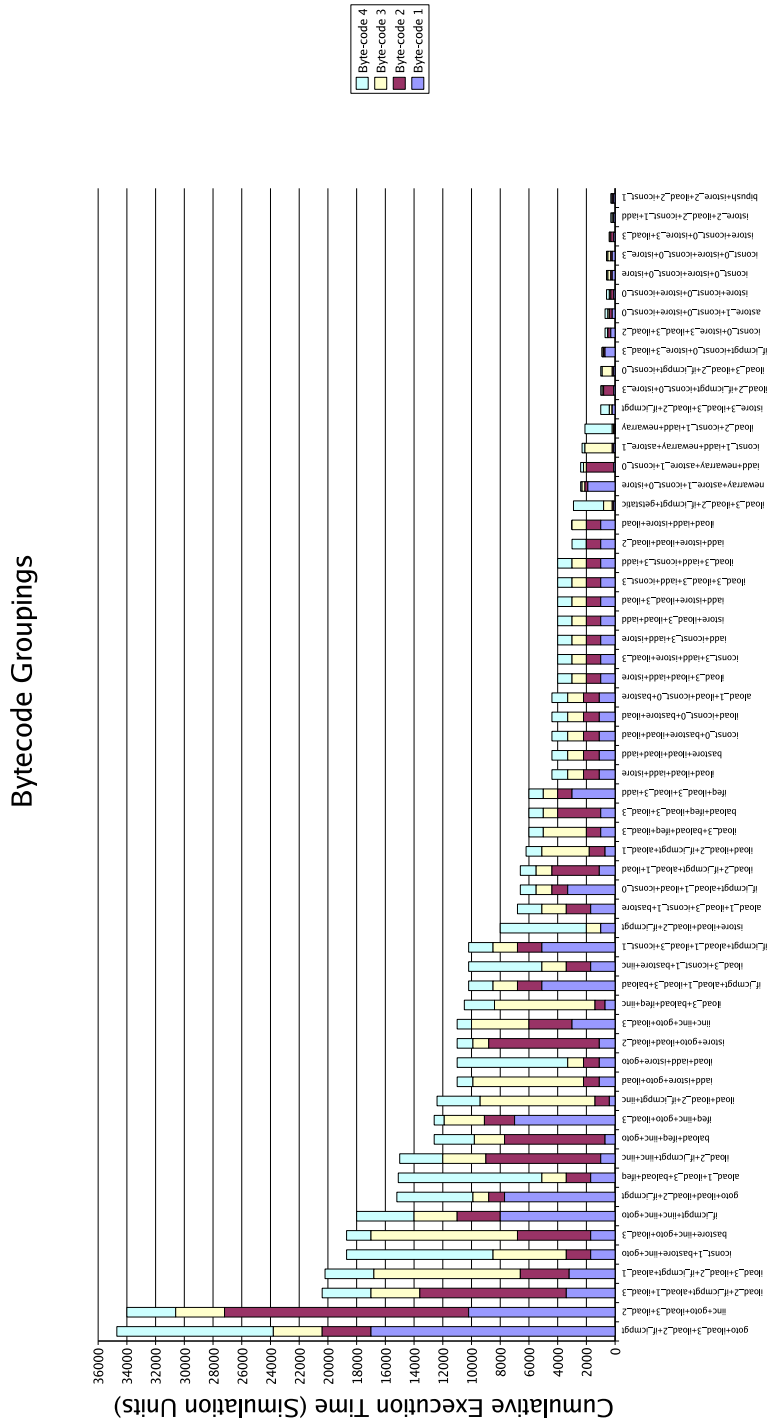


Figure A.24: Sieve Benchmark on Folding1-Branchopt, Grouped Byte-code Breakdown.

# Bibliography


[1] Accelera. *SystemVerilog Web Page*. Accelera, 2005.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, J.-D. Choi P. Cheng, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, J. R. Russell T. Ngo, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] ARM ltd. *Accelerating to Meet The Challenges of Embedded Java*, 2002.

[4] ARM Ltd. *ARM7TDMI (Rev 4) Technical Reference Manual*, 2003.

[5] E. Fetzer B. Doyle, P. Mahoney and S. Naffziger. Clock distribution on a dual-core, multi-threaded Itanium family microprocessor. In *Proceedings of International Conference on Integrated Circuit Design and Technology, 2005*, pages 1–6. IEEE Computer Society Press, may 2005. ISBN 0780390814.

[6] Andrew Bardsley. Balsa: An asynchronous circuit synthesis system. Master's thesis, Department of Computer Science, The University of Manchester, 1999.

[7] Andrew Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, The University of Manchester, 2000.

[8] Graham Birtwistle and Al Davis, editors. *Asynchronous Digital Circuit Design*, 1995.

[9] Greg Bollella and James Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[10] P. Capewell and I. Watson. A RISC hardware platform for low power java. In *"Proc VLSI Design 2005, 18th International Conference on VLSI Design"*, pages 138–143. IEEE Computer Society Press, jan 2005. ISBN 0769522645.

[11] Nazomi Communications. Nazomi communications web site. http://www.nazomi.com/, 2002.

[12] Kishinevsky M. Kondratyev A. Lavagnao L. Yakovlev A. Cortadella, J. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous

controllers. *IEICE Transactions on Information and Systems*, 80(3):315–325, March 1997.

[13] F Wieland L Blume M DiLoreto P Hontalas P Laroche K Sturdevant J Tupman V Warren J Wedel H Younger D Jefferson, B Beckman and S Bellenot. Distributed simulation and the time warp operating system. In *12th Symposium on Operating Systems Principle*, pages 77–93. IEEE Computer Society Press, 1987.

[14] Digital Equipment Corp. *Alpha Architecture Handbook*, 1992.

[15] Ahmed El-Mahdy, Ian Watson, and Greg Wright. Java virtual machine and integrated circuit architecture (JAMAICA) - choosing the instruction set. In Vijaykrishnan Narayanan and Mario L. Wolczko, editors, *Java Microarchitectures*. Kluwer, 2002.

[16] P.B. Endecott and S.B. Furber. Modelling and simulation of asynchronous systems using the LARD hardware description language. In *12th European Simulation Multiconference*, pages 39–43. Society for Computer Simulation International, June 1994.

[17] S B Furber. The return of asynchronous logic. Technical report, Department of Computer Science, University of Manchester, 1993.

[18] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference*, 1994.

[19] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, 1997.

[20] J.D. Garside, W.J. Bainbridge, , D.A Edwards, S.B. Furber, J Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Peltin, S Temple, and J.V. Woods. AMULET3i - an asynchronous system-on-chip. In *Sixth International Symposium on Asynchronous Circuits and Systems*, pages 162–175. IEEE Computer Society Press, 2000.

[21] APT Group. LARD homepage. http://www.cs.manchester.ac.uk/apt/projects/tools/lard/, 2005.

[22] Tom R Halfhill. ARM strengthens Java compilers. *Microprocessor Report*, July 2005.

[23] David Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Javatm virtual machine. In *ISORC*, pages 53–59, 2001.

[24] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard.

[25] IEEE. *VHDL Language Reference Manual, IEEE Standard 1076*. IEEE, 1988.

[26] IEEE. *Verilog Language Reference Manual, IEEE Standard 1364*. IEEE, 2001.

[27] The Open SystemC Initiative. Systemc homepage. http://www.systemc.org/, 2005.

[28] L Janin. *Simulation and Visualisation for Debugging Large Scale Asynchronous Handshake Circuits*. PhD thesis, Department of Computer Science, The University of Manchester, 2005.

[29] L Janin and D Edwards. Software visualisation techniques adapted and extended for asynchronous hardware design. In *9th International Conference on Information Visualisation, July 2005, London*. IEEE Computer Society Press, 2005.

[30] D Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (First Edition)*. Sun Microsystems, 1996.

[33] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (Second Edition)*. Sun Microsystems, 1999.

[34] Vulcan Machines Ltd. Vulcan machines ltd web site. http://www.vulcanasic.com/, 2005.

[35] Harlan McGhan and Mike O'Connor. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[36] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[37] S. M. Nowick. MIMIALIST: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical report, Computer Science Deptartment, Columbia University, 1999.

[38] L A Plana, P A Riocreux, W J Bainbridge, A Bardsley, J D Garside, and S Temple. SPA - a synthesisable Amulet core for smartcard applications. In *Eighth International Symposium on Asynchronous Circuits and Systems*, pages 201–210, 2002.

[39] Robert A Duff (eds) S Tucker Taft. Ada 95 reference manual: Language and standard libraries, international standard iso/iec 8652:1995(e). *Lecture Notes in Computer Science*, 1246, 1997.

[40] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[41] T. Fischer R. Riedlinger T.J. Sullivan S.D. Naffziger, G .Colon-Bonet and T. Grutkowski. The implementation of the Itanium 2 microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1448–1460, 2002.

[42] Velocity Semiconductor. Velocity semiconductor web site. http://www.velocitysemi.com/, 2005.

[43] Bruce D. Shriver and Bennett Smith. *The Anatomy of a High Performance Microprocessor (Interactive Book/CD-ROM): A Systems Perspective with CDROM*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[44] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[45] Sun Microsystems Computer Corporation. *The Java Language Specification*, 1995.

[46] Sun Microsystems Inc. *picoJava-II$^{TM}$ Microarchitecture Guide*, 1999.

[47] Sun Microsystems Inc. *The Java HotSpot Virtual Machine*, 2001.

[48] Sun Microsystems Inc. *Project Monty Virtual Machine*, 2002.

[49] Ajile Systems. Ajile web site. http://www.ajile.com/, 2002.

[50] Digital Communication Technologies. Lightfoot 32-bit java processor core product specification. http://www.xilinx.com/, 2001.

[51] G. Theodoropoulos and J. V. Woods. Analyzing the timing error in distributed simulations of asynchronous computer architectures. In *Proceedings 1995 EUROSIM Conference, EUROSIM95, Vienna, Austria*, pages 529–534. IEEE Computer Society Press, sep 1995. ISBN 0444822410.

[52] Georgios K. Theodoropoulos. Distributed simulation of asynchronous hardware: the program driven synchronization protocol. *J. Parallel Distrib. Comput.*, 62(4):622–655, 2002.

[53] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Systems*. Cambridge University Press, 1994.

[54] Zhongchuan Yu. *An Investigation into the Security of Self-timed Circuits*. PhD thesis, Department of Computer Science, The University of Manchester, 2004.