The DELFT-JAVA Engine

Clair Johnston Glossner

Computer Engineering Laboratory
Faculty of Electrical Engineering
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

November 5, 2001

The DELFT-JAVA Engine

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 5 november 2001 om 16.00 uur

door

Clair Johnston GLOSSNER

Master of Science in Electrical Engineering
National Technological University
geboren te Lock Haven, Pennsylvania, U.S.A.

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr. S. Vassiliadis

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG
Clair Johnston Glossner III
The DELFT-JAVA Engine

Clair Johnston Glossner III. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. – With summary in Dutch

ISBN 90-407-2255-2

Subject headings: computer arithmetic, computer design and engineering, multimedia enhancements, hardware accelerators, multimedia processors, multimedia architecture, java processor.

Cover design is “The Horsemen” of Java’s Mount Bromo. Courtesy of asiafoto.

Dedication

This dissertation is dedicated to my loving wife, Lisa, who has persevered through three advanced degrees while I worked more than full time. She has encouraged me to complete this work through job changes, physical relocations, four children, many late hours of work, and many even later hours of writing papers and this dissertation.

The DELFT-JAVA Engine

Clair Johnston Glossner

Abstract

In this dissertation, we describe the DELFT-JAVA engine - a 32-bit RISC-based architecture that provides high performance JAVA program execution. More specifically we describe a microarchitecture that accelerates JAVA execution and provide details of the DELFT-JAVA architecture for executing JAVA Virtual Machine bytecode. The basic architecture implements a Media Processor with Signal Processing capabilities. The perspective of the approach is that to maximally accelerate a compiled application, the machine language should accurately reflect the type of operations the compiler specifies. Except where JAVA Virtual Machine operations are unusually complex, we prefer to allow the compiler to optimize directly to the implementation. This is independent of any particular machine organization. The architecture is then a superset of the JAVA Virtual Machine and provides operations that are necessary for system execution (e.g., I/O, supervision, etc.). Rather than just supporting the JAVA Virtual Machine, the architecture takes a more general purpose approach in that it also is intended to be programmed from a number of additional high-level languages including C and C++. Furthermore, we introduce the concept of JAVA dynamic instruction translation, a new approach to JAVA hardware acceleration. In hardware assisted dynamic translation, JAVA Virtual Machine instructions are translated on-the-fly into the DELFT-JAVA instruction set. The hardware requirements to perform this translation are not excessive. Consequently, support for JAVA language constructs are also incorporated into the processor's Instruction Set Architecture. This technique allows application level parallelism inherent in the JAVA language to be efficiently utilized as instruction level parallelism. In addition to dynamic translation, a special Link Translation Buffer (LTB) can be used to improve the performance of dynamic linking. In addition, there are some key organization structures which we deem appropriate to provide architectural support for including: a) synchronization for multithreaded organizations, b) garbage collection, c) array bounds checking, d) real-time caches, e) multiple machines which can time-share the same datapath (e.g., the JAVA Virtual Machine and Media Processing functions), and f) vector/dsp operations. By building several models of the DELFT-JAVA engine, we were able to characterize performance metrics of kernels executing on our processor. We found that when compared to realizable stack-based machines, our techniques could improve performance by 2.7x. Furthermore, by converting stack-based dependencies into pipeline dependencies, we showed that out-of-order superscalar machines could remove up to 60% of the hazards.

Acknowledgments

I would like to thank prof.dr. Stamatis Vassiliadis for his patience, encouragement, guidance, and assistance in completing this research. His insights into high- performance processors significantly shaped the development of this dissertation. I would also like to acknowledge his wonderful abilities as a chef. His multitalented capabilities made my studies with him a complete joy.

I would like to thank prof.dr. Fred Brooks of the University of North Carolina. His love of computer architecture sparked a latent interest for me in computer architecture. The historical context in which he described machines of old and new created an environment that cultivated my interest in computer architecture.

Furthermore, I would like to thank my officemate prof.dr. Sorin Cotofana. We have had many wonderful artistic and philosophical discussions together. I really appreciate the education you gave me on European culture.

I also thank Lidwina Tromp and Mihaela Dirlea for their generous help in formatting the thesis. I am particulalry indebted to Lidwina for her help in the design of the cover of this thesis. I am also very thankful that Casper Lageweg translated the samenvatting and stellingen.

I would also like to thank and acknowledge Alexandru Berlea who contributed portions of the garbage collection work for this dissertation.

Finally, I woud like to thank dr. Jerry Pechanek. Jerry introduced me to prof.dr. Vassiliadis, mentored me in computer architecture, and imparted more wisdom to me than nearly all others whom I've come in contact with. My life is greatly enriched for the years I worked with Jerry. Especially memorable were all the great travels we shared together - especially Paris and the train.

John Glossner

Carmel, NY, May 2001

Contents

List of Figures

List of Tables

List of Programs

The only way to get something for nothing is to have previously gotten nothing for something – Fred Brooks.

Chapter 1

Introduction

JAVA is a C++ like programming language designed for general-purpose object-oriented programming[1]. An appeal for the usage of such a language is its "write once, run anywhere" philosophy [2]. This is accomplished by providing a JAVA Virtual Machine interpreter and runtime support for each platform[3]. In theory, any platform that supports the JAVA runtime environment will produce the same execution results independent of the platform. Due to its characteristics and possibilities, JAVA has been extensively used as a programming language of choice. In this dissertation we consider the possibility of executing JAVA programs efficiently and investigate the possibility of adding a specialized JAVA engine to provide high performance JAVA program execution.

High-speed communications are proliferating[4] and digital signal processors (DSPs) are accelerating this trend. DSPs have become a ubiquitous enabler for integration of audio, video, and communications[5]. Furthermore, DSPs are the driving force accelerating wireless communications. In the future world of convergence devices, efficient JAVA execution may be only one component of system performance. The DELFT-JAVA processor addresses these trends by providing facilities that enable efficient performance. Tremendous hardware and software challenges exist to realize convergence devices. First, power dissipation constraints are requiring new techniques at every stage of design - architecture, microarchitecture, software, algorithm design, logic design, circuit design, and process design. With performance requirements exploding as bandwidth demand increases, power conscious design becomes more difficult. SOC integration and low voltage process technologies will contribute to lower power system-on-a-chip (SOC) integrated circuits (ICs) but are insufficient as the only solution for streaming multimedia. Second, convergence applications are fundamentally DSP applications. In addition, these applications are becoming very complex. In wireless communications, GSM and IS-54 data rates were limited to less than 15 Kbps. Future third-generation (3G) systems may provide data rates more than 100 times the previous rates. Higher communication rates are accelerating higher processing requirements. Complexity is driving the need to program applications in high-level languages. In the past, when only small kernels were required to execute on a DSP, it was acceptable to program in assembly language. Today, resource constraints prohibit these practices. Third, JAVA may become the dominant programming paradigm for 3G systems. NTT DoCoMo recently rolled out Java-based services for its cellular subscribers and hardware solutions for efficient JAVA execution are being proposed[6]. Fourth, unlike many past developments, hardware designers will need to understand the complexities of software systems so that compilation techniques can be effective. With a large number of standards both existing and proposed for wireless communications, a programmable platform will be required for timely implementation. Fifth, embedded and DSP wireless applications have distinct requirements when compared with general purpose processors [7]. The predominant algorithmic difference is that inner loops are easily described as vectors of moderate length. A key point is that the native datatype is often fixed-point fraction. This is in distinct contrast to general purpose processors (and most high-level languages) which operate on integer datatypes. Finally, in addition to algorithmic differences, most convergence devices will be deployed in embedded environments where real-time constraints are prevalent. Real-time behavior has a dominant influence in the design of these devices[8]. Whereas general-purpose applications can often manage with variable latency response, convergence applications, in contrast, should be able to precisely guarantee the latencies within the system.

Our design includes modern DSP facilities required in convergence devices. Execution predictability in DSP systems precludes the use of many general-purpose design techniques (e.g. speculation, branch prediction, data caches, etc.). Instead, classical DSP architectures have developed a unique set of performance enhancing techniques that are optimized for their intended market. These techniques are characterized by hardware that supports efficient filtering, such as the ability to sustain three memory accesses per cycle (one instruction, one coefficient, and one data access). Sophisticated addressing modes such as bit-reversed and modulo addressing may also be provided. Multiple address units operate in parallel with the datapath to sustain the execution of the inner kernel. Examples of classical DSPs include TI's C54x [9], Lucent's 16xx[10], and IBM's Mwave DSP[11, 12].

Transitional DSP architectures have either attempted to extend existing architectures or solve a specific programming problem. The Lucent 16000 architecture extends the 1600 architecture to a dual-MAC machine while maintaining the same pipeline and programming style [10]. Likewise, TI's C55x extends the C54x to a dual-MAC machine [13]. Although these processors maintain many of the irregularities and specialized hardware of their predecessors, they provide performance gains and extend the lifetime of popular DSP families. Processors which typify transitional architectures include Infineon's Carmel DSP[14] and LSI's ZSP[15].

A special class of DSP architecture was introduced with the Media processor. Since these applications are dominated by pixel processing, an 8-bit datatype is often as important as a classical DSP's 16-bit datatype. These processors have had an influence on modern DSP architectures. Examples of media processors include IBM's Mfast [16–19], Philips' Trimedia [20], TI C80 [21], and Chromatics MPACT [22].

Another special class of processors with DSP functionality is general-purpose processors which include SIMD extensions. Examples of this include Intel's MMX[23] and PowerPC's Altivec [24]. Retrofitting DSP capability into general purpose processors has not been as successful as once envisioned. Although excellent performance can be achieved, system characteristics such as real-time constraints and power dissipation sensitivities are harder to realize on general purpose processors [25].

In classical DSP architectures, the execution pipelines were visible to the programmer and necessarily shallow to allow assembly language optimization. This programming restriction encumbered implementations with tight timing constraints for both arithmetic execution and memory access. The key characteristic that separates modern DSP architectures from classical DSP architectures is the focus on compilability. Once the decision was made to focus the DSP design on programmer productivity, other constraining decisions could be relaxed. As a result, significantly longer pipelines with multiple cycles to access memory and multiple cycles to compute arithmetic operations could be utilized. This has resulted in higher clock frequencies and higher performance DSPs.

In an attempt to exploit instruction level parallelism inherent in DSP applications, modern DSPs tend to use VLIW-like execution packets. This is partly driven by real-time requirements which require the worst-case execution time to be minimized. This is in contrast with general purpose CPUs which tend to minimize average execution times. With long pipelines and multiple instruction issue, the difficulties of attempting assembly language programming become apparent. Controlling instruction dependencies between upwards of 100 in-flight instructions is a non-trivial task for a programmer. This is exactly the area where a compiler excels. Representative examples of modern DSP architectures include Lucent/Motorola's Starcore SC140 [26], ADI's Tiger-SHARC[27, 28], TI's C64x[29], BOPS' ManArray[30–32], and Lucent's Daytona[33, 34].

Our design, combines a unique combination of modern techniques including SIMD execution and dynamically constructed VLIW or compounded instruction execution. The resulting combination provides for efficient DSP and JAVA execution. Future convergence devices will require both types of execution.

This chapter is organized as follows: In the section to follow we discuss briefly some related work. That is we consider approaches used to improve JAVA program execution time, and briefly describe original contributions of the proposals. Furthermore we identify open questions and the need to resolve the open questions left by the related work. The chapter is concluded with a brief discussion of the contents and the organization of the chapters to follow.

1.1 Related Work + Open Questions

JAVA Virtual Machine translation designers have used both software and hardware methods to execute JAVA bytecode. The advantage of software execution is flexibility. The advantage of hardware execution is performance. To try to blend the benefits of both approaches hybrid techniques have also been proposed. In this section we briefly describe existing approaches. The following have been proposed thus far:

■ **Interpretation:** Current implementations of the JAVA Virtual Machine take alternative approaches to JAVA bytecode execution. One solution is interpretation. In this approach, a software program emulates the JAVA Virtual Machine. This requires software to execute multiple machine instructions for each emulated instruction. This provides cross-platform portability but poses a number of performance issues. While this approach provides for maximum flexibility, the performance achieved can be as low as 5-10% the performance of natively compiled code [35].

■ **Just-in-time (JIT) Compilation:** For this approach translation is performed from JAVA bytecodes to native code (e.g. the machine language of the processor) just prior to executing the program. The Intel IA-32 JIT which is used in the VTune tool uses the JAVA bytecodes themselves to represent expressions rather than building an intermediate representation[36]. Using a technique called lazy code selection, native IA32 instructions are generated in a single pass with linear time complexity. They also describe lightweight implementations of several standard optimizations including common subexpression elimination, priority-based global register allocation, and array bounds check elimination. JITs have demonstrated 5-10x performance improvement over interpretation [35, 37]. However, the compilation is only resident for the current program invocation. Because they utilize processor resources, the number of optimizations that can be performed prior to execution is restricted[35]. Additionally, multiple instructions are required to implement JAVA Virtual Machine instructions and there is memory overhead to load the compiler into the runtime system.

■ **Flash Compilation:** Flash compilation is a hybrid approach in that a highly optimizing JIT compiler and a JAVA Virtual Machine are integrated into a runtime environment[37–39]. This allows code to be loaded in an already compiled application. The compiler only optimizes loops where a performance gain is likely to be obtained. The information may come from profiled bytecode execution. Stated performance improvements of 140x interpretive approaches and 13x JIT compilers have been reported.

In the Sun HotSpot compiler[38], released in 1999[40], a number of optimizations are made in addition to an optimizing compiler. The memory subsystem uses direct object pointers for objects rather than handles. C and JAVA programs share the same activation stack which allows fast calling of C routines. Garbage collection is performed using an accurate, generational copy collector which speeds object allocation and collection while reducing hard to debug memory leaks. A mark-and-compact algorithm eliminates memory fragmentation and pause-less collection ensures nearly imperceptible user-visible pauses. Special support is also provided for thread synchronization. The compiler itself does on-the-fly optimizations including method inlining, dead code elimination, loop invariant hoisting, common subexpression elimination and constant propagation. More sophisticated optimizations include null-check and range-check optimizations. Because new code can be loaded dynamically, the Sun HotSpot compiler has the ability to de-optimize (e.g. reverse the inlining process) to allow modification of the natively optimized code.

In the IBM dynamic compiler[39] a small VLIW machine with a JIT compiler hidden within the chip architecture is proposed. The first time a fragment of JAVA code is executed, the JIT compiler transparently converts the JAVA bytecodes to highly optimized RISC primitives, then parallelizes them, so multiple RISC primitives can be executed in one machine cycle. The VLIW code is saved in a portion of main memory not visible to the JAVA architecture. Subsequent executions of the same fragment do not require translation (unless cast out). They describe fast compiler algorithms for dynamic translation of JAVA bytecode to VLIW code. These algorithms parallelize across multiple paths and loop iteration boundaries. In addition, they map the JAVA stack and local variables to real registers, thereby eliminating the pushes and pops between local variables and the stack by appropriate register allocation.

■ **Off-line Compilation:** *Off-line compilers,* sometimes referred to as way-ahead-of-time compilers, translate JAVA bytecodes to machine code prior to execution. Because the scope of optimizations which can be performed in JIT compilers during JAVA execution is limited[41], an off-line compiler may devote additional time to complex optimizations. This requires that programs be distributed and installed (e.g. compiled) prior to use. Since it is assumed that the compilation is performed once and maintained on a disk, additional time may be devoted to optimizations. Except for loop information, the JAVA bytecodes contain nearly the same amount of information as the source itself[37]. Therefore, an off-line compiler should be nearly as efficient as a native JAVA compiler. A restriction on off-line compilers is that all of the class files must be present (e.g. all superclasses) to perform the compilation[36].

The Toba system first translates bytecodes to C and then compiles the C program[42]. For each JAVA method, Toba works by translating it into a C function. A C local variable is created for each JAVA local variable. Indirect jumps and exceptions are handled through a giant switch statement. Exception handling is based on the runtime program counter in the JAVA Virtual Machine. Toba simulates the program counter by assigning values to a local pc variable. Hewlett Packard has a similar system[43]. Using the Toba compiler, performance improvements nearly twice a standard interpreter have been reported for FFT signal processing functions[44, 45]. The Toba group found performance improvements of 2 to 10 times versus a standard interpreter[42].

■ **Native Compilation:** *Native compilers* use JAVA as a programming language and generate native code directly from the high-level source. Even though this approach is contrary to the JAVA philosophy of “write once, run anywhere” [2], it may provide a good opportunity for speed improvement since no information is lost during high-level compilation. A runtime system for linking the JAVA classes is still required and classes may potentially need to be resolved each time a method is invoked. Additionally, multiple instructions are required to implement JAVA Virtual Machine instructions. The gcj compiler is an example of a native compiler[46].

The previous approaches leave open the following questions:

- Is it possible to improve JAVA execution time with hardware?
- Is it possible to dynamically translate JAVA Virtual Machine instructions in hardware?
- Is it possible to accelerate JAVA dynamic linking?
- Is it possible to accelerate garbage collection in hardware?
- Is it possible to apply modern computer organizations to accelerate JAVA execution?
- Is it possible to overcome ILP limiting stack bottlenecks in JAVA bytecode?
- Is it possible to take advantage of the inherent parallelism expressed in the JAVA language?

■ **Direct Execution:** The previously mentioned questions could be possibly answered using direct JAVA execution. This approach assumes hardware capabilities that execute the JAVA Virtual Machine instruction set. Our proposal also belongs to this approach. We began in 1996 to investigate the previously mentioned questions. We envisioned that hardware approaches could significantly enhance JAVA Virtual Machine execution time. At that time only Sun Microsystems proposed similar approaches to improving JAVA performance. Sun's *picoJava* implementation directly executes the JAVA Virtual Machine Instruction Set Architecture (ISA) but incorporates other facilities that improve the system level aspects of JAVA program execution[47–49]. The picoJava chip is a stack-based implementation with a 64 entry register-based stack cache which automatically spills and fills based on high and low-water marks. Support for garbage collection, instruction optimization, method invocation, and synchronization is provided. Because the JAVA Virtual Machine does not implement an entire machine, Sun added 115 additional extended bytecodes to the picoJava core[49]. These extended bytecode are not produced by JAVA compliant compilers. Sun partitions the bytecode into simple instructions which can be directly executed, CISC-like instructions which are implemented in microcode using 2kB ROMs, and very complicated instructions which trap. Because a register-file stack cache is used, the picoJava core has access to the top 64 entries in the stack. This allows them to fold (e.g. combine) multiple stack-based operations into one execution packet. On average, Sun found about 28% of instructions executed get folded into other instructions. Researchers at National Chiao Tung University in 1997 found that instruction folding can reduce up to 84% of all stack operations and a 4-foldable Java core could improve overall program speedup by 1.34[50, 51]. Sun states that the picoJava core provides up to 5x performance improvement over JIT compilers [52].

We note that some questions discussed earlier have been investigated and results have been published[1]. At the time of our initiation, most of the open questions remained unresolved. What we thought could be a slow-paced evaluation turned out to be a frantic dash as JAVA's popularity and interest grew dramatically. The results of related investigations are also reported here. We would like to emphasize here that all published results are either at approximately the same date or later date than our preliminary results described in [53]. The following have been reported regarding direct JAVA execution[2].

- Using the Tomasulo technique[54], Munsil and Wang show that an adapted algorithm on simple benchmarks could reduce stack usage[55].
- Li et. al. from Tsignhua University also used a Tomasulo algorithm combined with a technique called virtual registers[56]. Their JAViR processor provides concurrent access to multiple stack entries. Virtual Registers are transparent to programmers and compilers (e.g. they are not architectural registers). At runtime, the dependencies of JAVA bytecode are checked and the virtual registers present the dependencies. These are then allocated to physical registers with a reference count that records the lifetime of the result. When a result is computed it is broadcast to the reservation stations. If the reference count for the virtual register is not zero, a new physical register is allocated. Using this technique they achieved an effective IPC (instructions per cycle) of 2.89 to 4.01 with a 16 and 64 entry instruction window, respectively.
- The TinyJ processor from Advancel Logic Corporation also directly executes about 60% of Java bytecode[57]. Complicated bytecode are em-

[1]Sun investigated improving Java execution with hardware. They particularly tried to overcome the problem of ILP limiting stack parallelism. Instruction folding partially resolves limitations in issuing multiple instructions from a stack-based instruction set. Results were published by Ton in 1997[51].

[2]Recently, a new approach has begun to emerge for Java execution. The new approach is based on FPGA execution.

ulated with software. They have two view of the machine - a Java Virtual Machine view and a RISC-based execution engine. They transition between the two views using a DISP (JAVA Virtual Machine dispatch instruction). They provide special hardware support for a JAVA program counter and special decode registers used to accelerated the software emulated long bytecodes. They also include rudimentary DSP multiply-accumulate instructions.

- The ShBoom PSC1000 processor from Patriot Scientific is a stack-based machine which is semantically close to the JAVA Virtual Machine [58]. It is a 32-bit processor with a peak instruction issue of 1 per cycle. To execute JAVA, a 20kB interpreter is required. This minimal memory requirement is due to the close semantic nature of the JAVA Virtual Machine instruction set architecture and the ShBoom instruction set architecture.

We note here that these approaches only partially address the previously mentioned open questions. Generally speaking, they attempt the ILP extraction without focussing on other important aspects of JAVA Virtual Machine execution. A number of such approaches have similarity to our approach. The TinyJ processor in particular uses a technique for transitioning between machines views that is close to ours. Most proposals avoid the more difficult questions of extraction of parallelism from the semantics of the JAVA language, acceleration of dynamic linking, dynamic translation of JAVA bytecode, and acceleration of garbage collection.

In this dissertation we provide answers to the open questions presented previously. We further provide techniques for significantly accelerating JAVA execution. We present results for the extraction of ILP from JAVA bytecode including an important technique of dependency collapsing. We also answer other thus far unresolved research questions such as dynamic translation, dynamic linking, and semantic extraction of JAVA parallelism from language constructs.

1.2 Framework of the Presentation

The dissertation is organized into six chapters excluding the current chapter. The chapters and discussion are organized as follows:

- Chapter 2, entitled **JVM: Brief Introduction**, provides background information. It is dedicated to a short description of the JAVA Virtual

Machine architecture. In the chapter, we introduce the reader to the stack machine concepts necessary to understand the JAVA Virtual Machine and present the memory organization, instruction formats and operations, and all concepts necessary to make the JAVA Virtual Machine operational.

- Chapter 3, entitled **Delft-Java Architecture**, provides a description of the architecture of the DELFT-JAVA processor and its organization. Instruction formats, operations, and specific hardware support for JAVA Virtual Machine instructions are described.

- Chapter 4, entitled **Microarchitecture and Java Acceleration**, provides a detailed description of how:

    1. JAVA Virtual Machine instructions are dynamically translated on-the-fly into DELFT-JAVA instructions,
    2. dynamic linking is accelerated through the use of a Link Translation Buffer, and
    3. programmer defined parallelism is exploited through multithreading.

We show that it is possible to efficiently translate JAVA Virtual Machine instructions into a native RISC-type instruction set architecture and accelerate execution. In addition, multiple instruction issue and dependency collapsing is described. This chapter shows that JAVA stack induced ILP bottlenecks can be removed from an instruction stream to allow efficient execution. In addition we show that multi-threaded architectures and machine organizations can be designed that allow for transparent extraction of JAVA parallelism.

- Chapter 5, entitled **Experimental Validation**, describes the benchmarks we used and provides an overview of the evaluation methodology. We look at performance results of dynamic translation for a number of hypothetical and realizable machines for both in-order and out-of-order organizations. We characterize IPC and speedup to determine the success of turning stack-induced hazards into pipeline hazards. The pipeline hazards are then removed through superscalar techniques. Next we characterize the speedup of organizational techniques for accelerating dynamic linking. We characterize a number of workloads and their performance improvement. Next we investigate signal processing applications and their suitability to the JAVA programming language. Specifically, we use tensor algebra to optimize the algorithms to the JAVA Virtual Machine and characterize the performance delta between C code and JAVA code. Finally, we summarize our experiments with garbage collection and identify opportunities where hardware may accelerate garbage collection.

- Chapter 6, entitled **Conclusions**, presents a short description of what has been achieved and discusses the overall significance of our research by describing specific contributions. Finally, it discusses directions for future investigation.

Knowledge brings power. – Stamatis Vassiliadis.

Chapter 2

JVM: Brief Introduction

As indicated in the introduction, JAVA is a C++-like programming language designed for general-purpose object-oriented programming[1]. The language includes a number of useful programming features including programmer defined parallelism support in the form of threads with synchronization, strong typing, garbage collection, classes, inheritance, and dynamic linking. The JAVA Virtual Machine (JVM) is a stack-based instruction set designed to efficiently transport programs across the Internet and allow register poor processors to efficiently execute JAVA bytecodes[2]. Instructions are not confined to a fixed length however all of the opcodes in the JAVA Virtual Machine are 8-bits[3]. This allows for efficient decoding of instructions while not requiring all instructions to be 32-bits or longer.

Generally speaking, the JAVA language supports many of the basic data representation types[1]. In contrast to C, their values are not implementation dependent[59]. In addition, the JAVA language also supports `char` which is a 16-bit unsigned Unicode character and a true `boolean`[2] for relational and logical operators. While JAVA does not allow operations on C-style pointers, it does have the concept of a reference type. There are three kinds: `class`, `interface`, and `array` types. These objects are created on a dynamically allocated heap. Multiple references may exist to the same object. The reference values are handles (e.g. pointers) to the object. The distinction is that a reference can not be operated on arithmetically as is often done in C-style pointers. Operations in the JAVA Virtual Machine are strongly typed. The 8-bit opcode imposes the availability of only 256 opcodes, this results in

[1]e.g. `byte`, `short`, `int`, `long`, `float`, and `double`.

[2]Note that while the JAVA language supports a `boolean` datatype, the JAVA Virtual Machine does not support it.

the trade-off that nearly all operations are performed as integers or IEEE-754 floating point. An interesting JAVA definition is that the JAVA Virtual Machine does not indicate overflow or underflow during operations on integer data types[3]. There are also load and store instructions which move values from memory locations to the operand stack in a very RISC-like manner. In addition to standard operations, there is direct support for method invocation, synchronization, exceptions, and arrays. Of the more unusual instructions, the `iinc` is a memory-to-memory instruction that increments the contents of a local variable location by a signed constant. There are two variable length instructions[3] - `tableswitch and lookupswitch`.

This Chapter is dedicated to a more in-depth description of the various concepts incorporated in the JAVA Virtual Machine. The chapter is organized as follows: Given that the JAVA Virtual Machine is a stack-based machine, we first describe the operation of a stack machine. Next we describe the JAVA Virtual Machine storage organization. We then describe the operations available in the JAVA Virtual Machine . Finally, we describe instruction execution.

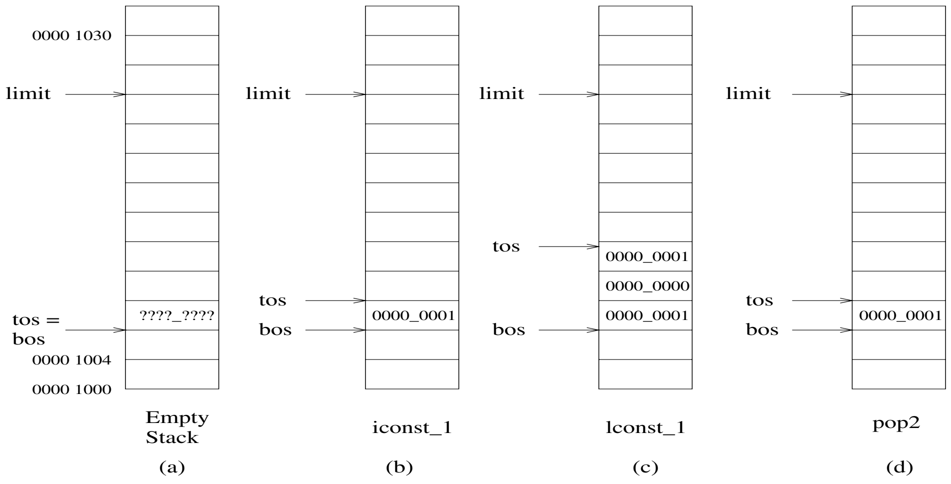


Figure 2.1: Stack Example.

[3]This is in contrast with instructions of variable length where the length of the instruction is fixed but variable.

2.1 Stack Machines and Java Execution

The JAVA Virtual Machine is based on the stack concept. A *stack* is an ordered set of elements [60]. Elements contained within the stack are accessed one at a time. The point of access is called the *top* of the stack. The number of elements in the stack is known as the *length* of the stack. Items may only be added to or deleted from the top of the stack. Figure 2.1 shows the basic stack mechanism. An area of memory is displayed beginning at byte address 0000_1000[4]. The bottom of stack pointer (bos) in 2.1a is equal to the top of stack pointer (tos) and is referencing address 0000_0008. By convention, we describe the stack as growing upward in memory[5]. In most stack architectures, a *push* operation moves an operand onto the top of the stack. Because the JAVA Virtual Machine is strongly typed, a special instruction for each datatype is required for push operations. As shown in the figure 2.1b, an *iconst_1* instruction (a push operation) pushes the 32-bit integer constant 0x1 onto the stack. Figure 2.1c shows the result of a 64-bit long integer constant 0x1 that is pushed onto the stack. In JAVA , the JAVA Virtual Machine specification does not mandate how two words of a 64-bit word are represented[3]. We choose to represent 64-bit quantities in big endian[6] notation utilizing two 32-bit stack locations. In addition to the iconst instructions, the JAVA Virtual Machine also contains *bipush* and *sipush* instructions. The *bipush* instruction pushes a sign extended byte onto the top of the stack. The *sipush* instruction pushes a sign extended 16-bit short word onto the stack. The JAVA Virtual Machine also has a traditional *pop* instruction. Figure 2.1 depicts the result of executing the pop2 instruction. This instruction pops the top two locations off of the stack[7].

[4]We adopt the following notation; values written as 0, 1, 2, etc. are considered to be integer representations and values written as 0x0, 0x1, 0xA, 0x0000_0001, or 0000_0001 are considered to be in hexidecimal notation. Specifically, when the number of bytes is important, we use the 0000_0001 or 0x00000001 format where each digit represents 4-bits. Additionally, a ? represents an undefined value of variable length depending upon the context in which it is used. If the length is not obvious, we specify the number of bits the ? occupies.

[5]To allow for C linkage, our actual implementation grows downward in memory. However, it is conceptually simpler to consider a stack as growing upward in memory.

[6]Endian refers to which bytes are most significant in multi-byte data types. In big-endian architectures, the leftmost bytes (those with a lower address) are most significant. In little-endian architectures, the rightmost bytes are most significant.

[7]Since the maximum size of the stack can be statically ascertained[59], a JAVA compliant compiler should never produce code that causes a stack underflow or overflow. However, in the case of malicious code that is not properly verified by the JAVA Virtual Machine, it may be necessary to provide a mechanism to ensure stack integrity. In Figure 2.1, a JAVA Virtual Machine should indicate a stack underflow condition if the top-of-stack (*tos*) pointer is ever decremented below the bottom-of-stack *bos* pointer. Likewise, if the *tos* pointer is incremented

```
class Add {
  public static void main( String argv[] ) {
    int i;
    int result = 0;
    for( i=0; i<10; i++ ) {
      result = result + 1;
    }
  }
}
```

Program 2.1: A Simple JAVA Program.

```
  Byte    Text          Comments
 Address
--------  ----          ---------------------
       0  iconst_0      ; push into 0x0 onto the stack
       1  istore_2      ; store tos into LocalVar[2] = result
       2  iconst_0      ; push int 0x0 onto the stack
       3  istore_1      ; store tos into LocalVar[1] = i
       4  goto 14       ; jump to address 14
       7  iload_2       ; load LocalVar[2] = result
       8  iload_1       ; load loop count i
       9  iadd          ; add them
      10  istore_2      ; store result in LocalVar[2]
      11  iinc 1 1      ; Memory increment LocalVar[1] + 1 = i
      14  iload_1       ; Load LocalVar[1] = 1
      15  bipush 10     ; Push byte 0xA onto stack
      17  if_icmplt 7   ; if( i < 10 ) goto 7
      20  return        ; return control to invoker
```

Program 2.2: Simple JAVA Bytecodes

To comprehend how JAVA programs operate consider Figures 2.2 and 2.3. These figures correspond to the execution of the bytecodes in Program 2.1 and 2.2. Program 2.1 depicts a simple JAVA program which adds 1 to a variable for 10 iterations of a loop. When compiled, the bytecode shown in Program 2.2 is produced. The notation represents the Local Variables memory area as LocalVar. Addresses in the LocalVar memory start at 0 at the bottom of the box and increment by 1 towards the top of the page. The width of the Local Variables memory space is 32-bits. In Figure 2.2a, the Loop Count and Result locations

beyond the limit register, a stack overflow condition should be indicated.

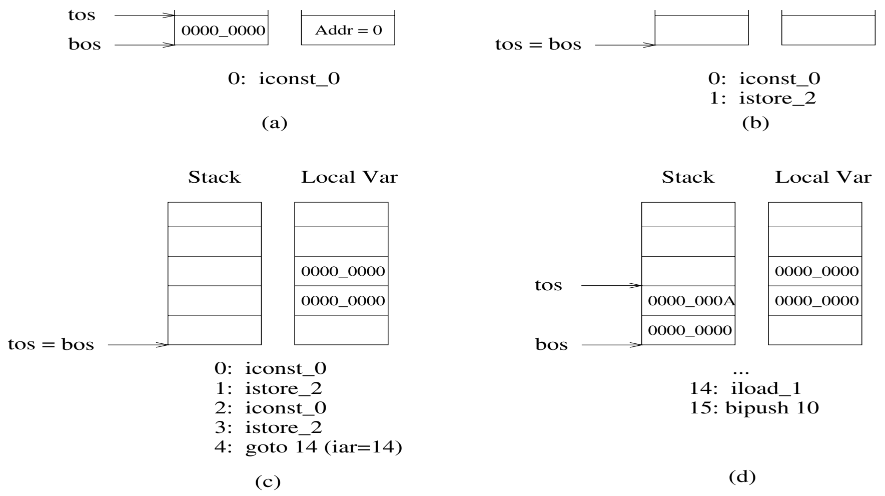


Figure 2.2: Stack Execution Example.

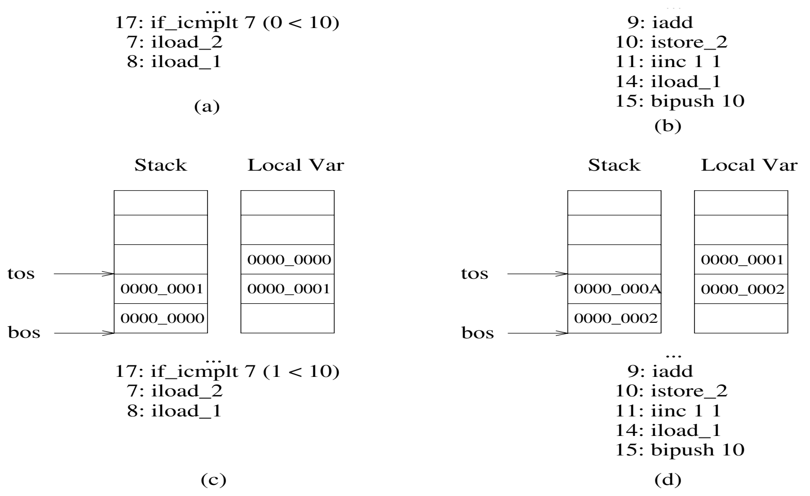


Figure 2.3: Bytecode Stack (cont.).

are symbolically labeled at Local Variables address 1 and 2, respectively. This represents the high-level JAVA variables i and result of Program 2.1. The state of the stack and local variables spaces are depicted at the completion of the last instruction in the list. For Figure 2.2a, this implies that the iconst_0 instruction has completed. Similarly, in Figure 2.2c, the state is depicted after the execution of goto 14 where 14 is the byte address of the instruction to branch to. In Figure 2.2a, the assembled program begins with an iconst_0 instruction. This instruction pushes an immediate 0x0 value onto the operand stack. This causes the top of stack pointer (tos) to be incremented. The Local Variables memory is shown but has not yet had any operations performed on it. Figure 2.2b shows the state of both the operand stack and the Local Variables memory area after the execution of the istore_2 instruction. The istore_2 instruction pops the 0x0 that was placed on the operand stack by the iconst_0 instruction execution of Figure 2.2a. The _2 notation denotes that the machine is to place the value in the Local Variables memory space at address 0x2. At the end of this operation the stack is empty. Figure 2.2c shows the state of the operand stack and local variable memory space after executing the sequence of instructions at addresses 0 through 4. The instructions at addresses 2 and 3 perform the same operations as Figures 2.2a and 2.2b. The instruction at method area byte address 4 is an unconditional branch. When the branch is executed, control is passed to instruction address 14. This is effected by writing the instruction address register (IAR) with the method area address 14. In Figure 2.2d, the ellipsis (...) implies that all of the previous instructions in Figures 2.2a-c have completed. The execution state is shown after the completion of the iload_1 and bipush 10 instructions. The iload_1 instruction loads the integer contents of the Local Variables memory space at address 1 onto the operand stack. The bipush 10 instruction pushes an immediate, sign extended value 10, represented in decimal notation, onto the stack. The first value corresponds to the loop count i in the high-level JAVA code shown in Figure 2.1. Similarly, the decimal value 10 represents the loop termination count.

2.2 Memory (Storage) Organization

This section is dedicated to the description of the JAVA Virtual Machine memory[8] organization. There are a number of run-time memory areas that are defined in the JAVA Virtual Machine[61]. The *Heap* is the run-time data area where all objects are dynamically allocated. There is one heap and it is shared among all threads. The heap is required to be garbage collected. The *Java Stack* is a private area created for each thread. It performs the same function as the stack in C-like languages. It can not be directly manipulated except to push and pop frames. A *Frame* is created each time a method, described

[8]In this presentation we will use memory and storage to have the same meaning except when explicitly stated.

later, is invoked and is destroyed when the method completes. It is used to store data, partial results, aid in dynamic linking, method results, and exception handling information. Only one frame can be active at any point for a particular thread. The *Operand Stack* is contained within the JAVA frame. It is defined to be 32-bits[62] with 64-bit values occupying two locations on the stack. The *Local Variables* area is allocated for each JAVA frame. It contains 32-bit variables addressed as word offsets from the base address of the area. Any 64-bit datatypes are considered to occupy two logically contiguous local variable locations. The *Method Area* is the location where the bytecode text is loaded. In addition, the method area also contains the *Constant Pool*. Constant Pool entries include numeric constants as well as symbolic references to dynamically loadable objects. More specifically, the following describes the JAVA Virtual Machine storage structure:

2.2.1 Spaces

Instructions and their operands must be obtained from a storage space or an input source; the results are placed in a storage space or an output sink. From a machine-language view, I/O can be treated as a specialized type of storage read-only or write-only. We distinguish three types of spaces: Memory, Working Store, and Control Store. We also discuss runtime memory areas which are not distinct spaces since no operations are provided. However, the runtime areas are useful in understanding how a JAVA Virtual Machine is implemented.

Memory Area	Space Limitations
Heap	Unbounded
Method Area	2^{16} bytes per method
Constant Pool	2^{16} entries per class
Local Variables	2^{16} 32-bit entries per method
Operand Stack	2^{16} (class file restriction)

Table 2.1: JVM Memory Spaces.

■ **Memory:** The *memory space* is the storage space from which programs are directly executed. This may be augmented with auxiliary store or secondary store. The JAVA Virtual Machine does not have provision for accessing auxiliary store. Embedding allows the same space to be accessed in two syntactically distinct ways. An example is mapping the register file into the memory space. The JAVA Virtual Machine architecture has no embedding of address spaces. The JAVA Virtual Machine specifies the following types of memory:

Heap: The *heap* is a runtime data area that is shared among all threads. It is garbage collected and contains object instances and array allocations. The JAVA Virtual Machine specification does not require the heap to be a fixed size. If the physical memory capacity of the heap is exceeded, an `OutOfMemory-Error` is reported.

Method Area: The *method area* is a single space that is shared among all threads. It contains the constant pool, field and method data, and the code for methods and constructors. According to the JAVA Virtual Machine specification, it may be of fixed or variable size. The memory area is not required to be contiguous. If enough memory can not be allocated, an `OutOfMemoryEr-ror` is reported.

Constant Pool: The *Constant Pool* is a per-class or per-interface runtime data area that contains numeric literals and symbolic names of classes that are dynamically linked. The JAVA specification states that this area is allocated from the method area's space. The constant pool is created when a class or interface specified in a JAVA class file has successfully been loaded. If the physical memory capacity of the method area is exceeded, an `OutOfMemo-ryError` is thrown. Each index into the Constant Pool references a variable length structure.

■ **Working Store:** The *working store* is the set of concisely specifiable locations that temporarily contain operands or results of the operations.

Operand Stack: The primary JAVA Virtual Machine working store is a stack, termed the *Operand Stack*, of 32-bit words, placed as a variable-length, variable-location segment in memory[9]. The *Operand Stack* is logically part of the *JavaFrame* that is allocated on method invocation. Most instructions operate on the current frame's operand stack and return results to it. The operand stack is also used to pass arguments to methods and receive method results. A `long` or `double` is considered to occupy two stack locations. All operations on the operand stack are strongly typed and must be appropriate to the type being operated upon.

Local Variables: The *Local Variables* are logically part of the *JavaFrame* that is allocated on method invocation. There are up to 2^{16} local variable locations per method invocation. Each location is 32-bits wide, placed as a fixed-length,

[9]The Burroughs B5500, circa 1963, used a similar organization[63]. In the JAVA Virtual Machine, this area of the memory space may be implemented with a register file. It is particularly convenient because the stack is not global but is local to each executing thread.

variable-location segment in memory. A `long` or `double` is considered to take two local variable locations. The local variables hold the formal parameters for the method and partial results during a computation.

■ **Control Store:** The *control store* is the storage that contains the status of the processor, and the information to control the syntactic and semantic interpretation process. The JAVA Virtual Machine provides no directly accessible control registers. For changes in control flow (if_<cond>,jsr, etc.), all transfers are relative to the current bytecode. This implies an instruction (or bytecode) address register. However, all operations on this register are indirect side effects.

Instruction Address Register (IAR)[10]: The *Instruction Address Register (IAR)* controls the flow of program execution. Branch conditions and other control instructions may indirectly modify the IAR. Upon completion of the current instruction, the IAR is indirectly incremented by the bytelength of the current instruction.

■ **Runtime Memory Areas:** The runtime memory areas are not distinct spaces and no operations are provided that operate on them. We describe them here for completeness. The *JavaStack*[11] is a per thread memory area created to store frames. It is equivalent to the stack of a conventional language and hold local variables, partial results, method invocation parameters, and return parameters. The memory is not required to be contiguous and may be either fixed or dynamically varying in size. If the maximum stack size is exceeded by a computation in a thread, a `StackOverflowError` is thrown. If enough physical memory is not available to create the *JavaStack*, an `OutOfMemory-Error` is thrown.

A *JavaFrame* is per method space allocated from the *JavaStack*. It stores data and partial results, performs dynamic linking, returns values for methods, and dispatches exceptions. A new frame is created each time a JAVA method is invoked. It is destroyed when the method terminates whether it is normal or abnormal termination. Each frame may contain a local variable area and an operand stack[12]. Since the size of all these areas is known at compile time, the size of the frame data structure depends only upon the implementation [3].

[10]Note that in the JAVA Virtual Machine literature, this is often incorrectly described as the *Program Counter (pc)*. In fact, it does not count programs. Therefore, we prefer to give it the proper term - Instruction Address Register.

[11]The *JavaStack* is not to be confused with the Operand Stack.

[12]We note that the implementation of the Frame is not mandated. We may therefore store a pointer to the actual Local Variables and Operand Stack memory spaces.

The *NativeMethodStack* is allocated on a per thread basis and is used to support `native` methods which are written in a language other than JAVA. It is not required to be implemented. If it is implemented, it may be of either fixed or variable size. If the maximum stack size is exceeded by a computation in a thread, a `StackOverflowError` is thrown. If enough physical memory is not available to create the Native Method Stack, an `OutOfMemoryError` is thrown.

2.2.2 Memory (Storage) Access

In this section we describe the storage access characteristics of the JAVA Virtual Machine. *Storage space* is a two-dimensional array of bits, whose rows are addressable units and whose columns are the bits within each unit.

■ **Address Space:** The one-dimensional vector of addresses possible in a storage space is its *address space* or *name-space*. An *address* is a storage element's unique name. The *name-spaces* of a language are the disjoint sets into which the names of objects are grouped. Each object has a unique name or address within its name-space. Two names are defined to be in the same name-space if either can be substituted for the other in any machine-language statement without altering that statement's syntactic correctness.

In the JAVA Virtual Machine, a set of successive integers as addresses is assigned as the name-space of specific objects. This provides an isomorphic mapping between the set of all possible n-bit names and the set of binary integers from 0 to $2^n - 1$. This constitutes a dense, ordered, and measured set[13]. This allows the same mechanisms used for operations on data to be used for comparisons and additions desirable for names.

Address-Set Structure: The *address-set* structure is linear with detection of addresses beyond the ends of the installed segment. This ensures that an increase of memory will not affect correct execution of programs.

Address Resolution: The minimal *memory address resolution* is an 8-bit byte. The byte ordering convention is *big-endian*. Big-endian is chosen because of the logical convention of treating the whole storage space as one stream of bits. Bits, bytes, half-words, etc. are numbered from left to right.

Information-Unit Alignment: Memory is not required to be aligned with the memory space being manipulated. It is recommended that compilers generate code that is aligned with the datatype being manipulated. Some language sys-

[13]By measured we mean that the successor of a name can be calculated by addition.

tems (e.g. FORTRAN) forbid alignment. However, it is highly recommended to align values where ever possible.

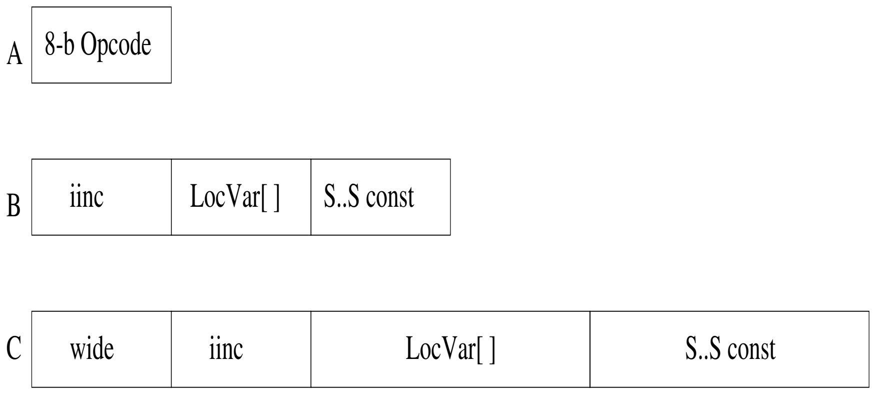


Figure 2.4: JVM Operand Address Formats

■ **Names and Addresses:** In programming languages, objects are identified by names. The *name* may refer to a single object but most names refer to groups of data and instructions. The *address* is the corresponding machine-language name.

Binding: The *memory location* is the place where a programmer defined name is stored. *Binding* is the process of mapping a programmer defined name to an address. The address is then interpreted at execution time to refer to the memory location. Because the set of programs whose data are simultaneously in memory may change, it is desirable not to have a fixed correspondence between the programmer defined name and the object's actual location in memory.

■ **Address Modification:** The *address calculation* computes an element of an array or matrix by taking the array name and adding an index. This is called *address modification* when the address calculation takes place as part of instruction execution. The result of the calculation is the *effective address*.

Address Components: The *base address* specifies the location of the array in memory. The *element address* specifies an element within a data structure. This is relative to the base address. The *displacement* determines the location of an item relative to the current element address. The JAVA Virtual Machine provides for all three components but not in a traditional manner. For simple data accesses (e.g. load, store, load_<n>, store_<n>), the JVM provides the element address directly from an index. JAVA Arrays provide a base and element address. For example, the base address in the iaload instruction is an 32-bit array reference. An index is also provided to supply the element address. The JAVA Virtual Machine also provides for dynamic binding of data locations. The putfield and getfield instructions use a modified form of element addressing.

Effective Address Calculation: For simple data accesses, there is no address modification and the direct address is the effective address. For JAVA Arrays, the components are added to form the effective address. For dynamic binding of locations, an object reference to the class of the field requested is used as a type of base address[14]. An index into the Constant Pool of the class requesting the field is used to return the name of the field request. The effective address is then formed by resolving, loading, and linking the requested field name[1].

Location of Address Components: For simple data accesses, the element address is provided directly by an address field of the instruction. For Array accesses, the base address (e.g. array reference) and element address (e.g. index) are provided in the operand stack. For dynamic binding of location, the object reference is provided on the operand stack and the index into the current class' Constant Pool is provided by an address field in the instruction.

■ **Index Arithmetic:** The JAVA Virtual Machine index arithmetic is described by the following:

General Index Operations: Index arithmetic takes place in either the Operand Stack or the Local Variables memory (using the iinc instruction). All integer operations available for normal computations are available for index arithmetic.

Stack Addressing: The JAVA Virtual Machine provides a generalized Operand Stack. The maximum size of the stack is computed statically at compile time. A current Class File restriction limits the maximum number of stack locations to 2^{16} entries. Checks for overflow and underflow of the stack are not enforced at runtime but may be optionally verified statically during class loading. The stack is logically in memory although the top of the stack may be cached or contained within a register file.

Incrementing: All incrementing, whether for general index operations or stack operations takes place on the stack or through the Local Variables memory (using the iinc instruction). The index for the stack is called the *stack*

[14]Note that the object reference does not specify an array in memory but specifies a reference to the class which contains the field.

pointer. There is no directly accessible stack pointer register that can be manipulated as part of the JAVA Virtual Machine instruction set. However, an implied stack pointer is indirectly modified as the result of operations. The stack is not defined to grow in a particular manner. We adopt the arbitrary convention that the stack grows from high to low memory addresses. The stack pointer then points to the top element. A pop operation becomes a Read followed by an increment corresponding to the size of the data just read. This is a postincrement. A push is a write preceded by a decrement, the predecrement.

■ **Address Levels:** Addresses which refer directly to the machine-language names for data are called *direct addresses*. An address which refers to another address rather than the machine-language name is called *indirect address*. There are no indirect addresses in the JAVA Virtual Machine architecture[15]. An *immediate address* is an address that does not name a data item but is itself used as the data item. The JAVA Virtual Machine provides only the proper operation of using immediate addressing for loading the stack.

2.3 Operations

An *information unit* is the information that an operation stores and operates upon as a single entity. The JAVA Virtual Machine basic unit system is defined to be an 8-bit long byte. As shown in Table 2.2, operations are provided for standard datatypes as well as Unicode characters, Object references, and return addresses. In contrast to their C counterparts, JAVA values are not implementation dependent[59].

Operations are provided for integer, logical, and floating point datatypes. The following definitions are from [64]. A *datatype* consists of a *referent set* (the set of concepts represented) and a *representation* (a set of bit patterns and the encoding and allocation that define the correspondence). A processor's datatypes are divided into a couple of categories.

1. those used to produce bit patterns with new meaning (data in the narrow sense)

[15]The convention is adopted that for an address to be indirect it must first retrieve a pointer from memory and then use that as the direct address. Often, the term *register indirect* is used where the address given refers to an abbreviated register reference. We consider this a direct address.

Type	Interpretation
$byte$	1-byte signed 2's complement integer
$short$	2-byte signed 2's complement integer
int	4-byte signed 2's complement integer
$long$	8-byte signed 2's complement integer
$float$	4-byte IEEE 754 single-precision floating point
$double$	8-byte IEEE 754 double-precision floating point
$char$	2-byte unsigned Unicode character
$reference$	4-byte reference to a JAVA object
$array$	Array of another type
$returnAddress$	4-byte reference used with jsr, ret, jsr_w, ret_w

Table 2.2: JAVA Virtual Machine Datatypes.

2. instructions and other status words used to control the computing system itself.

Each datatype has a set of operations proper to it. A datatype is made up of several subparts, which may themselves be datatypes (e.g. vectors and matrices). Often, one kind of representation can be embedded in another (e.g. addresses are usually encoded as integers).

■ **Logical Data:** A logical (or Boolean) datum has two possible states - true and false. It can be represented by a single bit. The JAVA language provides a true boolean type. However, the JAVA Virtual Machine does not directly support this single-bit type. The allocation for logical operations is a 32-bit vector of booleans. A logical datum is encoded as a binary value with true and false states. The JAVA Virtual Machine adopts the C convention that a false value is represented by zero and any non-zero value is true.

■ **Fixed Point Numbers:** The representation of fixed point numbers includes the number system choice, the allocation of elements, and the element representation.

Number System: The JAVA Virtual Machine uses a positional representation for the number system. Two types of fixed-point numbers can be represented: positive whole numbers which include zero (termed positive integers or unsigned integers) and integers. The notation for positive integers uses a binary radix with an implied sign of zero in the most significant (hidden) bit. The position of the radix point is just to the right of the digits.

For integers, the notation of negative numbers uses *radix complement*. The radix value is binary. This is commonly referred to as 2's complement. The position of the radix point is just to the right of the digits (e.g. integer notation). Because the high-order digit (which denotes the sign) participates fully in arithmetic, multiple precision low-order numbers can be treated as all digits.

The implied position of the radix point matters only for complement notations. The multiplication of two n-bit numbers (with sign) words yields $2n - 2$ bits of product and a sign. In the complement notations, the sign is treated as a digit.

⧫ **Allocation of elements:** An allocation of elements is comprised of the number of digits and a sign.

Number of digits: For a binary radix, each digit requires 1 bit. The number of digits is fixed based on the datatype. For addresses (e.g. object references), all lengths are 32-bits. The length of data is implicitly given by the operation code with the instruction.

Sign: For unsigned integers, the sign is hidden in the most significant bit (the left-most bit) and implied to be 0 (e.g. positive). For integers, the sign is encoded as required for radix complement notations (0 for positive and 1 for minus). The sign bit is left aligned into the most significant bit.

⧫ **Element Representation:** The allocation is one-to-one in that the number of digits determines the length of the number in bits. Digits are encoded as 0 for zero and 1 for one.

■ **Floating Point Numbers:** All floating-point notation follows the IEEE-754-1985 standard. In fixed-point arithmetic, the basic arithmetic laws can be preserved provided some indication mechanism is used when the result overflows the representation range. In floating-point arithmetic, however, preserving exact results would entail corrective action. This would reintroduce programmed scaling and therefore defeat the purpose of floating-point arithmetic. Therefore, floating-point arithmetic usually produces an approximation of the result. The key arithmetic law violated is the associative law. As a result, the distributive law also fails as do cancellation and solvability.

■ **Arrays:** An array's elements are placed in groups of equal size. The number of elements in a group is called that group's *dimension*. The *rank* of an array is the number of dimensions of the array. A vector has rank 1. A matrix has rank 2. A scalar has rank 0. The JAVA Virtual Machine has support for arrays of rank up to 255 (limited by an 8-bit dimension field in the instruction format). Additionally, all arrays are *homogeneous* - all elements are considered to be of the same datatype for all operations. All element types must be one of the primitive datatypes.

2.3.1 Operation Specification

An *operation* is the elementary independent step carried out by a machine. An *operation code* is the encoded specification of an operation. It may specify not only the action to be taken but also the datatype of the operand. It may also specify the instruction format and the format for the operation code itself. An *instruction* is the elementary independently reorderable unit of a program. We group the operations in a similar manner as Stallings[60] with additional categories for JAVA specific operations.

■ **Secondary Operations:** A *secondary operation* is one implied by an explicitly specified operation; the operand and result locations are usually implied as well. An example of this is the sign test. Secondary operations create a dependence between previously independent functions - they violate orthogonality. The JAVA Virtual Machine does not provide secondary operations (e.g. indicators). All bits participate in the computation. All arithmetic is modulo its native datatype.

■ **Operation Encoding:** A *homogeneous encoding* contains no internal structure to the correspondence between code point and the representant (e.g. no significant subsets of the representands can be discerned by examining the codes). A fully decomposed specification provides n bits for each of the n actions. The JAVA Virtual Machine uses a partially decomposed specification. Operations are decomposed into *actions* - (e.g. Add, Logic, etc.) and *modifiers*. Operations are specified using a sequence of variable length with an 8-bit basic sequence. The operation code is always found within the same bits of all the instruction formats[16] and is always 8-bits in length.

2.3.2 JAVA Virtual Machine Operation List

■ **Data Handling:** Data-handling operations are those that do nothing else - no arithmetic or logic. The data movement operations change neither allocation nor coding. An unchanged bit pattern is copied from one place to another.

[16]The *wide* instruction could be considered to be an exception. Because it may not be the target of a branch operation, it can also be considered a delimiter instruction which modifies the behavior of the following instruction. Using the later interpretation allows the JVM to treat the instruction independently.

The format transformation operations change allocation but not encoding. As with data-handling operations, the format transformations are code independent (e.g. decoding and encoding steps are not involved). The third class of data-handling operations is code transformation, which changes coding and in general requires reallocation.

Data Movement: Data movement corresponds to the programming-language assignment primitive. The JAVA Virtual Machine provides data movement operations via working store in the form of stack manipulations, Load, and Store operations. Table 2.3 summarizes the JAVA Virtual Machine data movement instructions.

Instruction Name	Function/Syntax	Size/ Opcode
pop	Remove top word from operand stack *pop*	1 87
pop2	Remove top two words from operand stack *pop2*	1 88
dup	Duplicate top word on operand stack *dup*	1 89
dup_x1	Duplicate top word on operand stack and put two down *dup_x1*	1 90
dup_x2	Duplicate top word on operand stack and put three down *dup_x2*	1 91
dup2	Duplicate top two words on operand stack *dup2*	1 92
dup2_x1	Duplicate top two words on operand stack and put three down *dup2_x1*	1 93
dup2_x2	Duplicate top two words on operand stack and put four down *dup2_x2*	1 94
swap	Swap top two words on operand stack *swap*	1 95

Table 2.3: JVM Data Movement Instructions

Instruction Name	Function/Syntax	Size/ Opcode
bipush	Push (Load) sign-extended imm8 onto stack *bipush (S..S)imm8*	2 16
sipush	Push (Load) operand stack with sign-extended imm16 *sipush (S..S)imm16*	3 17
ldc	Load operand stack with 32-bit ConstantPool[(0..0)imm8] value *ldc (0..0)cp_index_imm8*	2 18
ldc_w	Load operand stack with 32-bit ConstantPool[imm16] value *ldc (0..0)cp_index_imm16*	3 19
ldc2_w	Load operand stack with 64-bit ConstantPool[imm16] value *ldc2_w (0..0)cp_index_imm16*	3 20
aconst_null	Push (Load) operand stack with null reference *aconst_null*	1 1
iconst_<i>	Push (Load) operand stack with int constant i *iconst_m1; iconst_0; iconst_1; etc.*	1 2-8
lconst_<l>	Push (Load) operand stack with long constant l *lconst_0; lconst_1*	1 9-10
fconst_<f>	Push (Load) operand stack with float constant f *fconst_0; fconst_1; fconst_2*	1 11-13
dconst_<d>	Push (Load) operand stack with double constant d *dconst_0; dconst_1*	1 14,15

Table 2.4: JVM Load Constant Instructions

Note that in Table 2.4, when using the ldc instruction it is also possible for a String to be loaded. In this case, a String Class object is created on the heap and the value loaded is a reference to the String.

Instruction Name	Function/Syntax	Size/ Opcode
iload	Load operand stack with int in LocalVariable[(0..0)imm8] *iload lv_index_(0..0)imm8*	2 21
iload_<n>	Load operand stack with int in LocalVariable[n] *iload_0; iload_1; etc.*	1 26-29
lload	Load operand stack with long in LocalVariable[index],[index + 1] *lload lv_index_(0..0)imm8*	2 22
lload_<n>	Load operand stack with long in LocalVariable[n],[n+1] *lload_0; lload_1; etc.*	1 30-33
fload	Load operand stack with float in LocalVariable[(0..0)imm8] *fload lv_index_(0..0)imm8*	2 23
fload_<n>	Load operand stack with float in LocalVariable[n] *fload_0; fload_1; etc.*	1 34-37
dload	Load operand stack with double in LocalVariable[index],[index + 1] *lload lv_index_(0..0)imm8*	2 24
dload_<n>	Load operand stack with long in LocalVariable[n],[n+1] *dload_0; dload_1; etc.*	1 38-41
aload	Load operand stack with reference in LocalVariable[(0..0)imm8] *aload lv_index_(0..0)imm8*	2 25
aload_<n>	Load operand stack with reference in LocalVariable[n] *aload_0; aload_1; etc.*	1 42-45

Table 2.5: JVM Load Constant Instructions

In Table 2.4, the iload, lload, fload, dload, and aload instructions, the Local Variable index (imm8) can be extended to an imm16 using the wide instruction. In the lload and dload instructions, the order of LV[(0..0)imm8] and LV[(0..0)imm8 + 1] may be reversed provided the data returned from the two addresses results in the stack state: ...,word1,word2.

Instruction Name	Function/Syntax	Size/ Opcode
istore	Store operand stack containing int to LocalVariable[(0..0)imm8] *istore lv_index_(0..0)imm8*	2 54
istore_<n>	Store operand stack containing int in LocalVariable[n] *istore_0; istore_1; etc.*	1 59-62
lstore	Store operand stack containing long in LocalVariable[index],[index + 1] *lstore lv_index_(0..0)imm8*	2 55
lstore_<n>	Store operand stack containing long in LocalVariable[n],[n+1] *lstore_0; lstore_1; etc.*	1 63-66
fstore	Store operand stack containing float in LocalVariable[(0..0)imm8] *fstore lv_index_(0..0)imm8*	2 56
fstore_<n>	Store operand stack containing float in LocalVariable[n] *fstore_0; fstore_1; etc.*	1 67-70
dstore	Store operand stack containing double in LocalVariable[index],[index + 1] *dstore lv_index_(0..0)imm8*	2 57
dstore_<n>	Store operand stack containing long in LocalVariable[n],[n+1] *dstore_0; dstore_1; etc.*	1 71-74
astore	Store operand stack containing reference in LocalVariable[(0..0)imm8] *astore lv_index_(0..0)imm8*	2 58
astore_<n>	Store operand stack containing reference in LocalVariable[n] *astore_0; astore_1; etc.*	1 75-78

Table 2.6: JVM Store Constant Instructions

In Table 2.6 the istore, lstore, fstore, dstore, and astore instructions, the Local Variable index (imm8) can be extended to an imm16 using the wide instruction. In the lload and dload instructions, the order of LV[(0..0)imm8] and LV[(0..0)imm8 + 1] may be reversed provided the data returned from the two addresses results in the stack state: ...,word1,word2.

Format Transformation: Format Transformations reallocate one machine field into another without changing the encoding. Operations are provided to convert datatypes to smaller or larger lengths while maintaining their encoding (e.g. formatting a $byte$ to int, $short$ to $long$, or int to $byte$). In some cases the bits are truncated. In other cases, they are zero- or sign-extended. Format transformations are accomplished by restricting the domains and types of the general code transformation instructions.

In formatting a double to a single, if the represented value is too small it is formatted as a zero of the same sign and if it is too large, it is formatted as an infinity of the same sign. Table 2.7 summarizes the available JAVA Virtual Machine format transformations.

Instruction Name	Function/Syntax	Size/ Opcode
i2l	Format int (w32) to long (w64) *i2l*	1 133
i2b	Format int (w32) to byte (w8) by trucation and sign-extension *i2b*	1 145
i2s	Format int (w32) to short (w16) by truncation and sign-extension *i2s*	1 147
l2i	Format long (w64) to int (w32) by truncating high-order 32-bits *l2i*	1 136
f2d	Format float (f32) to double (f64) *f2d*	1 141
d2f	Format double (f64) to float (f32) using round-to-nearest *d2f*	1 144

Table 2.7: JVM Format Transformation Instructions

Code Transformation: One data representation in a machine language can be transformed to another representation if the concept sets that the two represent intersect. The JAVA Virtual Machine architecture allows general transformations between floating point, integer, and unsigned integer representations.

When converting from types float or double to types int or long, the following rules are used:

1. if the floating point representation is NaN, the resultant is zero
2. if the value is an infinity, the maximum representable integer of the appropriate sign is used
3. Otherwise, the operand is converted to an integer using round-to-zero. If this is too small to be represented, the smallest representable integer is used.

Table 2.8 summarizes the available JAVA Virtual Machine code tranformation operations.

Instruction Name	Function/Syntax	Size/ Opcode
i2f	Convert int to float using IEEE-754 round-to-nearest *i2f*	1 134
i2d	Convert int to double *i2d*	1 135
i2c	Convert int to unsigned char using zero-extension and truncation *i2c*	1 146
l2f	Convert long to float using IEEE-754 round-to-nearest *l2f*	1 137
l2d	Convert long to double using IEEE-754 round-to-nearest *l2d*	1 138
f2i	Convert float to int *f2i*	1 139
f2l	Convert float to long *f2l*	1 140
d2i	Convert double to integer *d2i*	1 142
d2l	Convert double to long *d2l*	1 143

Table 2.8: JVM Conversion Instructions

■ **Logic:** All logical operations apply to vectors of bits. There is no direct boolean datatype in the JAVA Virtual Machine although the JAVA language contains a boolean type. There are however logical vectors.

Connectives: The dyadic operations upon a single pair of bits are called *connectives*. The JAVA Virtual Machine architecture supplies 3 connectives on integer and long types. Connectives are inherently unsigned. Table 2.9 summarizes the available JAVA Virtual Machine logical connectives.

Instruction Name	Function/Syntax	Size/ Opcode
iand	int logical boolean bitwise AND (conjunction) *iand*	1 126
ior	int logical boolean bitwise inclusive OR *ior*	1 128
ixor	int logical boolean bitwise exclusive OR *ixor*	1 130
land	long logical boolean bitwise AND (conjunction) *land*	1 127
lor	long logical boolean bitwise inclusive OR *lor*	1 129
lxor	long logical boolean bitwise exclusive OR *lxor*	1 131

Table 2.9: JVM Logical Connective Instructions

Composite Functions: A *composite logical function* applies a dyadic scalar function among the elements of a bit vector. The JAVA Virtual Machine contains no composite instructions.

Shift Operations: A *shift* operation is used in data handling for field selection; in logic, for bit inspection, and in arithmetic, for programmed scaling, multiply, divide, and floating point. The JAVA Virtual Machine provides a rudimentary set of logical and arithmetic shift functions. The shift displacement is passed on the stack using the low-order 5 bits for an integer and the low-order 6-bits for a long (e.g. shiftval = $2^{low\_order\_bits}$). Table 2.10 summarizes the available JAVA Virtual Machine shift instructions.

Instruction Name	Function/Syntax	Size/ Opcode
ishl	int arithmetic shift left *ishl*	1 120
ishr	int arithmetic shift right ($\lfloor \frac{operand}{2^s} \rfloor$) *ishr*	1 122
iushr	int logical shift right *iushr*	1 124
lshl	long arithmetic shift left *lshl*	1 121
lshr	long arithmetic shift right ($\lfloor \frac{operand}{2^s} \rfloor$) *lshr*	1 123
lushr	long logical shift right *lushr*	1 125

Table 2.10: JVM Shift Instructions

Bit Manipulations: The JAVA Virtual Machine does not contain any bit manipulation instructions.

■ **Fixed-Point Arithmetic:** Computer arithmetic operations are best understood as consisting of two steps. First, the operation as defined in mathematics is applied to the interpretations of the operand representations. Next, the mathematical result is changed by a domain function, if necessary, such that it can be represented in the datatype's domain. Computer arithmetic does not have closure because of finite domains and ranges. For fixed-point representation, the domain function usually does not change the mathematical result. The JAVA Virtual Machine does not contain any flags to indicate overflow. All arithmetic is modulo. The *representation range* is the range of integers that can be represented by the integer datatypes. Datatypes are not properly mixed and therefore must undergo a format transformation prior to arithmetic operations.

Addition and Subtraction: When operands and results have equal lengths, an overflow can always be represented by 1 bit. The JAVA Virtual Machine does not store the overflow bit and the results is truncated to fit within the datatype (e.g. modulo arithmetic). However, if overflow occurs in a two's-complement encoding, the sign of the result will not be the same as the sign of the mathematical sum of the two values. In negation, because the range of values is not symmetric in 2's-complement encoding, when the most negative number is negated, the result is again the most negative number. No exception is thrown in this case. Table 2.11 summarizes the available JAVA Virtual Machine addition and subtraction instructions.

Instruction Name	Function/Syntax	Size/ Opcode
iadd	int add (truncate) *iadd*	1 96
isub	int subtract (truncate) *isub*	1 100
ineg	int arithmetic negation *ineg*	1 116
iinc	Increment LocalVariables[(0..0)imm8] by (S..S)imm8. (No change to stack) *iinc lv_index_(0..0)imm8 (S..S)imm8*	3 132
ladd	long add (truncate) *ladd*	1 97
lsub	long subtraction (truncate) *lsub*	1 101
lneg	long arithmetic negation *lneg*	1 117

Table 2.11: JVM Add and Subtract Instructions

Multiplication and Division: For multiplication, the number of product digits is just less than or equal to the total number of digits of the multiplier and the multiplicand. The JAVA Virtual Machine considers the resultant destination to be the same length as the operands. This is as per the JAVA (and C/C++) language convention. If an overflow occurs, the least significant bits of the mathematical product are used (e.g. truncation).

For division, if the dividend is the largest representable number and the divisor is -1, then the result is equal to the dividend (e.g. overflow occurred). No exception is thrown. The quotient's magnitude is the largest possible while satisfying $|divisor * quotient| \leq |dividend|$. A divisor of 0 throws an ArithmeticException.

For remainder, the result is $dividend - (\frac{dividend}{divisor}) * divisor$. The identity $\frac{a}{b} * b + a\%b = a$, always holds. Moreover, the result of the remainder operation will have the sign of the dividend. A divisor of 0 throws an ArithmeticException. Figure 2.12 summarizes the available JAVA Virtual Machine multiplication and division instructions.

Instruction Name	Function/Syntax	Size/ Opcode
imul	int multiply (truncate) *imul*	1 104
idiv	int division (round toward 0) *idiv*	1 100
irem	int remainder *irem*	1 112
lmul	long multiply (truncate) *lmul*	1 105
ldiv	long division (round toward 0) *ldiv*	1 109
lrem	long remainder *lrem*	1 113

Table 2.12: JVM Multiplication and Division Instructions

■ **Floating Point:** All floating point operations correspond exactly to the IEEE-754-1985 standard including denormalized numbers and gradual underflow. Inexact results must be rounded to the representable value nearest to the infinitely precise result. Round-towards-zero (the default rounding) effectively truncates the mantissa. Floating point operations produce no exceptions. An overflow produces a signed infinity, an underflow produces a signed zero, and an indefinite result produces NaN.

Addition and Subtraction: For addition, the following IEEE rules apply:

1. if either value is NaN, the result is NaN
2. the sum of two infinities of opposite sign is NaN
3. the sum of two infinities of the same sign is the infinity of that sign
4. the sum of infinity and a finite value is infinity
5. the sum of two zeroes of opposite sign is positive zero
6. the sum of two zeroes of the same sign is the zero of that sign
7. the sum of a zero and a nonzero finite value is equal to the non-zero value
8. the sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.

Moreover, all sums are computed and rounded using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign.

For subtraction, the result of $a-b$ is the same as $a+(-b)$ except for the case of subtraction of 0.0. For example, $0.0-0.0 = +0.0$ while $0.0+(-0.0) = -0.0$.

For negation, the following additional IEEE rules apply:

1. the negation of an infinity is the infinity of opposite sign
2. the negation of a zero is the zero of opposite sign.

Table 2.13 summarizes the available JAVA Virtual Machine addition and subtraction operations.

Instruction Name	Function/Syntax	Size/ Opcode
fadd	float add (round-to-nearest) *fadd*	1 98
fsub	float subtract (round-to-nearest) *fsub*	1 102
fneg	float arithmetic negation *fneg*	1 118
dadd	double add (round-to-nearest) *dadd*	1 99
dsub	double subtraction (round-to-nearest) *dsub*	1 103
dneg	double arithmetic negation *dneg*	1 119

Table 2.13: JVM Floating Point Add and Subtract Instructions

Multiplication and Division: For multiplication, the following IEEE-754 rules apply:

1. if either operand is a NaN, the result is NaN
2. if neither operand is NaN, the sign of the result is positive if both operands have the same sign, and negative the the operands have different signs
3. multiplication of infinity by a zero is NaN
4. multiplication of an infinity by a finite operand produces an infinity following the sign rule given above.

Moreover, the product is computed and rounded using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign.

For division, the following IEEE-754 rules apply:

1. if either operand is a NaN, the result is NaN
2. if neither operand is NaN, the sign of the result is positive if both operands have the same sign, and negative the the operands have different signs
3. division of infinity by infinity is NaN
4. division of infinity by a finite value produces an infinity following the sign rule given above
5. division of a finite value by an infinity produces a zero following the sign rule given above
6. division of a zero by a zero results in NaN
7. division of zero by a finite value produces a zero following the sign rule given above
8. division of a nonzero finite value by a zero produces an infinity following the sign rule given above.

Moreover, the quotient is produced using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign. For remainder, the definition is *not* the same as the IEEE-754 version. The IEEE version uses rounding division while the JAVA Virtual Machine uses truncating division (to more closely approximate the integer behavior). The following JAVA Virtual Machine rules apply:

1. if either operand is a NaN, the result is NaN
2. if neither value is NaN, the sign of the result is equals the sign of the dividend
3. if the dividend is an infinity, or the divisor is a zero, the result is NaN
4. if the dividend is finite and the divisor is an infinity, the result equals the dividend
5. if the dividend is a zero and the divisor is finite, the result equals the dividend.

Moreover, the result is $dividend - intof((\frac{dividend}{divisor}) * divisor)$. The function $intof()$ rounds toward the nearest integer, or towards the nearest even integer if the number is half way between two integers.

Table 2.14 summarizes the available JAVA Virtual Machine multiplication and division instructions.

Instruction Name	**Function/Syntax**	**Size/ Opcode**
fmul	float multiply (round-to-nearest) *fmul*	1 106
fdiv	float division (round-to-nearest) *fdiv*	1 110
frem	float remainder *frem*	1 114
dmul	double multiply (round-to-nearest) *dmul*	1 107
ddiv	double division (round-to-nearest) *ddiv*	1 111
drem	double remainder *drem*	1 115

Table 2.14: JVM Floating Point Multiplication and Division Instructions

Relational Operations: Relational operations test a specified relation among operands and produce a result that is true or false. A relation can formally be considered to be a mapping from an input domain, consisting of all possible values of the datum, to an output space or range, consisting of one point for each category. Generally, a compare instruction is provided for this purpose. The JAVA Virtual Machine provides comparisons for types long, float, and double. The kind of comparison used is a ranked comparison (e.g. <,=,> membership). The domain of the comparison is the operand stack. The results are recorded explicitly on the operand stack. Table 2.15 shows the result of a comparison:

Comparison	**Result**
$>$	1
$=$	0
$<$	-1
NaN	1 (fcmpg)
NaN	-1 (fcmpl)
$+0.0 = -0.0$	0

Table 2.15: JVM Comparison Results

For floating point relational operations, the following JAVA Virtual Machine rules apply:

1. positive zero and negative zero are equivalent
2. negative infinity is less than positive infinity
3. a NaN is unordered and can be determined from a combination of fcmpg and fcmpl instructions.

Table 2.16 summarizes the available JAVA Virtual Machine relational instructions.

Instruction Name	**Function/Syntax**	**Size/ Opcode**
lcmp	Compare long (op1 <> op2) *lcmp*	1 148
fcmpg	Compare float (NaN = 1) *fcmpg*	1 150
fcmpl	Compare float (NaN = -1) *fcmpl*	1 149
dcmpg	Compare double (NaN = 1) *dcmpg*	1 152
fcmpl	Compare double (NaN = -1) *fcmpl*	1 151

Table 2.16: JVM Comparison Instructions

■ **Array Operations:** The JAVA Virtual Machine provides for operation on arrays. JAVA arrays are created dynamically and are a type of Object (e.g. all methods of class Object may be invoked on an array). An array object may contain zero or more variables (called *components*). The *length* of an array is the number of components it contains. Non-negative integer index values are used to access arrays. An array of zero components is not equivalent to a null reference. All components within an array must be of the same type (which, however, may be an array type). The *element type* is the component type of the components which are not array types. If the element type is of type Object, then it is possible to have a case where the element type is actually an array type. Moreover, an array of type char is not a string. Neither is a String an array of char. Arrays may also contain an abstract class component type.

A variable of array type holds a reference to an object but the array object is created using an array creation instruction. The length of an array is not part of its type. Once an array is created, its length never changes. All arrays are 0-origin. They are indexed using int values. Array access bounds are checked at runtime. An index greater than or equal to the array length causes an ArrayIndexOutOfBoundsException to be thrown.

Array Creation: Arrays are created by specifying the number of elements (passed on the operand stack) and the element type. Valid element types are shown in Table 2.17. Unfortuantely, Sun's JAVA Virtual Machine architecture does not provide a way to disambiguate a byte from a boolean (e.g. the same instruction is used to load and store both types). Therefore, even though Sun suggests other implementations may implement packed boolean arrays, this is not architecturally possible (although a software interpreter may be able to accomplish it). A special instruction (anewarray) is used to create an array of type reference. All arrays are allocated to the garbage collected heap. If the number of elements is less than zero, a *NegativeArraySizeException* is thown. Table 2.17 summarizes the available JAVA Virtual Machine array types which may be created. Table 2.18 summarizes the available JAVA Virtual Machine array creation instructions.

Array Type	aType	Array Type	aType
bool	4	byte	8
char	5	short	9
float	6	int	10
double	7	long	11

Table 2.17: JVMArray Element Types

Instruction Name	Function/Syntax	Size/ Opcode
newarray	Create a new array for non-object types *newarray imm8_aType*	2 188
anewarray	Create a new array for objects *anewarray imm16_CPindex*	3 189
multianewarray	Create a new multidimensional array *multianewarray imm16_CPindex, imm8_dim*	4 197

Table 2.18: JVM Array Creation Instructions

Array Data Movement: The JAVA Virtual Machine provides instructions to load and store values in Arrays. Table 2.19 and Table 2.20 summarizes the available JAVA Virtual Machine array Load and Store instructions, respectively.

Instruction Name	**Function/Syntax**		**Size/ Opcode**
<t>aload	Load type <t> from heap to operand stack *<t>aload*	$< t >= i, f, a$ $result = int, float, ref$	1 46,48,50
<t>aload	Load type <t> from heap to operand stack *<t>aload*	$< t >= l, d$ $result = long, double$	1 47,49
<t>aload	Load type <t> from heap to operand stack *<t>aload*	$< t >= b, s$ $result = (S..S)byte, (S..S)short$	1 51,53
<t>aload	Load type <t> from heap to operand stack *<t>aload*	$< t >= c$ $result = (0..0)char$	1 52

Table 2.19: JVM Array Load Instructions

When datatypes smaller than 32-bits are stored to the heap, the values are truncated to the size of the type even though they are maintained on the stack as 32-bit quantities.

Instruction Name	**Function/Syntax**		**Size/ Opcode**
<t>astore	Store type <t> from operand stack to heap *<t>astore*	$< t >= i, f, a$ $operand = int, float, ref$	1 79,81,83
<t>astore	Store type <t> from operand stack to heap *<t>astore*	$< t >= l, d$ $operand = long, double$	1 80,82
<t>astore	Store type <t> from operand stack to heap *<t>astore*	$< t >= b, c, s$ $operand = (Trunc)byte, char, short$	1 84-86

Table 2.20: JVM Array Store Instructions

Miscellaneous Array Instructions: The JAVA Virtual Machine provides an instruction that returns the length of an Array. The length is pushed onto the stack as an int. In JAVA, all arrays are 0-origin. An array with length n can be indexed by the integers 0 through $n - 1$. All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an *ArrayIndexOutOfBounds* exception to be thrown. Table 2.21 summarizes the JAVA Virtual Machine miscellaneous array instructions.

Instruction Name	**Function/Syntax**	**Stack State Prior/Post**	**Size/ Opcode**
arraylength	Get length of an array *arraylength*	...,arrayref ...,length	1 190

Table 2.21: JVM Array Miscellaneous Instructions

■ **Object Operations:** The JAVA Virtual Machine provides for operation on objects including accessing fields, casts, comparison, etc. JAVA allocates all object on the garbage collected heap. Objects can never reside on the Java Stack.

Object Creation: All JAVA Virtual Machine objects are created on the garbage collected heap. Table 2.22 summarizes the available JAVA Virtual Machine object creation instruction.

Instruction Name	Function/Syntax	Stack State Prior/Post	Size/ Opcode
new	Create new object *new imm16_CPindex*	... ...,objectref	3 187

Table 2.22: JVM Object Creation Instructions

Object Manipulation: JAVA Virtual Machine objects may access fields which are members of class objects (static fields) or instance objects. Operations are provided for moving values to/from fields and to/from the operand stack. The type of the field is determined from the Constant Pool index and the appropriate number of bytes are pushed or poped. Table 2.23 summarizes the available JAVA Virtual Machine object manipulation instructions.

Instruction Name	Function/Syntax	Size/ Opcode
getfield	Get field from object *getfield imm16_CPindex*	3 180
putfield	Put field into object *putfield imm16_CPindex*	3 181
getstatic	Get static field from class *getstatic imm16_CPindex*	3 178
putfield	Put static field into class *putstatic imm16_CPindex*	3 179

Table 2.23: JVM Object Manipulation Instructions

Miscellaneous Object Operations: The JAVA Virtual Machine provides an operation that ensures a type cast between objects is valid. It also provides an operation to determine if a reference is an instance of a particular class. If a cast is made to an improper object type, an *ClassCastException* is thrown. The *instanceof* operation does not throw an exception but places an int 1 on the operand stack if a proper instance is found. Otherwise, a int 0 is pushed onto the operand stack. Table 2.24 summarizes the available JAVA Virtual Machine miscellaneous object operations.

Instruction Name	Function/Syntax	Size/ Opcode
checkcast	Check whether object is a given type *checkcast imm16_CPindex*	3 192
instanceof	Check whether object is a given type *instanceof imm16_CPindex*	3 193

Table 2.24: JVM Miscellaneous Object Instructions

■ **Miscellaneous Operations:** The JAVA Virtual Machine provides some additional operations.

The *athrow* instruction searches the current frame for a catch clause that catches the class that threw the exception. If it is found, the instruction address register is reset to that location, the operand stack of the current frame is cleared, the object reference is pushed back onto the stack, and execution continues. Otherwise, the frame is popped, the frame of the invoker is reinstated, and the exception for that object reference is rethrown. Ultimately, if no catch clause is found, the current thread exits. If the object reference is null, a *NullPointerException* is thrown instead of the object reference exception.

Two opcodes are specifically reserved for JAVA Virtual Machine implementations and are guaranteed by Sun not to be used in future implementations of the JVM. One of these opcodes is used by the DELFT-JAVA processor to transition between JAVA Virtual Machine execution and normal DELFT-JAVA execution. Table 2.25 summarizes the available JAVA Virtual Machine miscellaneous instructions.

Instruction Name	Function/Syntax	Size/ Opcode
nop	No Operation *nop*	1 0
wide	Extend Local Variable Index *wide*	1 196
athrow	Raise an exception *athrow*	1 191
breakpoint	Reserved for Debuggers	1 202
rsv	Implementation Dependent	1 254,255

Table 2.25: JVM Miscellaneous Instructions

2.4 Instruction Execution

In this section we describe JAVA Virtual Machine instructions and their execution.

■ **Instructions:** A sequence of instructions is defined as a *procedure*. The procedure and the data to which it applies are jointly called a *program.*

Sequence Specification: The status variable that indicates the location of the instruction to be fetched and executed next is the *instruction address.* Because most instructions are stored consecutively in memory, the address is typically incremented each time an instruction is executed. Because of the incrementation, the instruction address is often misnamed the *instruction counter* or the *program counter*.

Instruction Format: The *instruction format* is the summary document for CPU architecture.

Status Format: *Status* is information that controls the interpretation process, in contrast to instructions that define this process and data that determine the result of this process. The status must be saved as part of the context saving during program switching. Similarly, part of the status is saved during subroutine call and return. The memory format for saved status is called the *status format*. Such a unit of control information is called a *status word* or *control word*.

■ **Instruction Sequencing:** A sequence requires a specification and a normal sequence. The specification was previously treated. A normal sequence requires selecting a Continuity and a Choice. The Continuity is partitioned into a linear sequence and a delegation. The choice can be a decision or iteration.

Linear Sequence: The simplest structure is the linear sequence - or vector arrangement - of instructions. When instructions are arranged in a vector, each can be identified by the vector index of its position in memory - its address. The address where an instruction resides is called its *location*. The design choices for a linear sequence include Dependence, Next Location, and Completion.

Functional Independence: Normally, it is desirable that all instructions to be executed constitute independent syntactical units. The JAVA Virtual Machine places semantic dependencies between instructions with the use of a *wide* instruction prefix. It is used to extend memory indicies or constant values.

Instruction Location: The next instruction is placed linearly in memory, using an implied instruction address, incremented by the length of the instruction.

Completion: A program typically has a well-defined end. A *wait* instruction is generally provided for this purpose. Because the JAVA Virtual Machine assumes a higher level run-time interpreter or operating system, it does not require a program end. When all JAVA methods have terminated, control returns to the run-time system.

Decision: The design choices for Decision include the Condition and an Alternative Action.

Condition: The condition is decomposed into a general computation with explicit condition followed by a general condition test and the target selection. Indicators that are recorded as the result of a general computation are termed condition *codes* because they encode secondary operations. A problem with condition codes is that they constitute state. The JAVA Virtual Machine does not use hidden condition codes but rather places the results of a comparison (e.g. lcmp, fcmp, etc.) on the stack. The condition for a branch is calculated as part of the total computation.

Alternative Action: Having specified the condition, one must indicate which action corresponds to each of its values. The entails specifying a Branching factor, and a Target Address. Since the JAVA Virtual Machine supports a CASE statement, branching factors may be greater than two. The exclusive use of 16-bit signed relative branches (e.g. Branch Target Address = iar + simm16) allows a section of program to be relocated without modification. Instructions are aligned on byte boundaries except for the tableswitch and lookupswitch instructions which are padded to align on a 4-byte boundary from the beginning of a method. All branch target addresses must be within the current executing method.

Tables 2.26 and 2.27 summarize the JAVA Virtual Machine conditional branch instructions.

Instruction Name	**Function/ Syntax**	**Size/ Opcode**
if_icmp<cond>	Branch relative if int datum2$<$ $cond$ $>$datum1 $cond =$ *if_icmp<cond> simm16* eq, ne, lt, le, gt, ge	3 159-164
if<cond>	Branch relative if int datum$<$ $cond$ $>$0 $cond =$ *if<cond> simm16* eq, ne, lt, ge, gt, le	3 153-158

Table 2.26: JVM Integer Conditional Branch Instructions

Instruction Name	Function/ Syntax	Size/ Opcode
if_acmp<cond>	Branch relative if reference datum2$<cond>$datum1 $cond = eq, ne$ *if_acmp<cond> simm16*	3 165-166
ifnonnull	Branch relative if reference is not null *ifnonnull simm16*	3 199
ifnull	Branch relative if reference is null *ifnull simm16*	3 198

Table 2.27: JVM Reference Conditional Branch Instructions

The JAVA Virtual Machine supports unconditional branching. The Branch Target Address is a relative signed 16-bit or 32-bit offset from the current instruction address. All branch target addresses must be within the current executing method. Note that due to class file restrictions, a JAVA method is limited to 2^{16} instructions. Table 2.28 summarizes the JAVA Virtual Machine unconditional branch instructions.

Instruction Name	Function/ Syntax	Size/ Opcode
goto	Jump relative by short offset *goto simm16*	3 167
goto_w	Jump relative by int offset *goto_w simm32*	5 200

Table 2.28: JVM Unconditional Branch Instructions

The JAVA Virtual Machine supports two special instructions which support case statements. Each instruction pops an int key from the operand stack. The key is compared with all the case values. If a match is found, the branch offset associated with the case value is taken. Otherwise, the default case value is taken. The number of case comparison/offset pairs is unsigned 32-bit value given in the instruction format.

Instruction Name	Function/ Syntax	Size/ Opcode
lookupswitch	Access jump table by key match *lookupswitch <0-3 byte pad> simm32_default, simm32_npairs, match-offset pairs*	variable 171
tableswitch	Access jump table by index *tableswitch <0-3 byte pad> simm32_default, simm32_low, simm32_high, offsets*	variable 170

Table 2.29: JVM Switch/Case Instructions

Iteration: Iteration involves a scope (what is to be iterated) and a termination condition (when iteration should be stopped). In the JAVA Virtual Machine, all iteration is done through conditional branching.

Delegation: Delegation of control allows a recurring function to be detailed only once and to be called from many places. The JAVA Virtual Machine provides delegation instructions for method invocation and subroutines.

Method Invocation: The JAVA Virtual Machine provides support for instance methods and class methods. Instance methods are dynamic and use late binding. Class methods are static (e.g. its type is known at compile time) and may use early binding. Each occurence of a non-native method invocation creates a new stack frame within the executing thread. The stack frame contains space for all the state of the virtual machine including local variables and the operand stack (both of whose size are determinable at compile time).

An interface method invocation must create a hash table to map from a specific class to a structure containing that class's implementation of a specific interface. For a class invocation, the offset of the method will be the same in the method table regardless of the actual class or the object. For an interface reference, the method may occupy different locations for different classes that implement the same interface.

All method invocations complete with a strongly typed return instruction. Table 2.30 summarizes the JAVA Virtual Machine method invocation instructions while Table 2.31 summarizes the return instructions.

Instruction Name	Function/ Syntax	Size/ Opcode
invokeinterface	Invoke Interface Method *invokeinterface imm16_CP, imm8_nargs, 0x00*	5 185
invokespecial	Invoke superclass, private, or constructor method invocation *invokespecial imm16_CP*	3 183
invokestatic	Invoke a class method *invokestatic imm16_CP*	3 184
invokevirtual	Invoke an instance method based on dynamic class type *invokevirtual imm16_CP*	3 182

Table 2.30: JVM Method Invocation Instructions

Instruction Name	Function/ Syntax	Size/ Opcode
return	Return from a method *return*	1 177
<t>return	Return result from a method $< t >= i, f, a$ *<t>return* $int, float, ref$	1 172,174,176
<t>return	Return result from a method $< t >= l, d$ *<t>return* $long, double$	1 173,175

Table 2.31: JVM Return Instructions

Subroutines: Subroutines in the JAVA Virtual Machine are used only to support exception handling. They are used to implement the finally clause of the JAVA language. All subroutines are local to the body of method. Upon a subroutine invocation, the instruction address to return to is pushed onto the stack. Execution then branches relative to the instruction address plus a signed-immediate 16-bit offset. The *ret* instruction intentionally retrieves the instruction address from a local variable location and not from the stack. If more than 256 Local Variables are required for the ret instruction, a wide prefix may be used to modify the local variable index.

Instruction Name	**Function/ Syntax**	**Size/ Opcode**
jsr	Jump to subroutine *jsr simm16*	3 168
jsr_w	Jump to subroutine (wide offset) *jsr_w simm32*	5 201
ret	Return from subroutine *ret imm8_LVindex*	2 169

Table 2.32: JVM Subroutine Instructions

Parameter Passing: Parameters are passed either through the operand stack or Local Variables depending upon the type of call/return sequence. For method invocations, parameters are passed and returned on the operand stack. For the intra-method subroutine call, the instruction address is passed on the stack but it is returned through a local variable location.

State Preservation: If a method or subroutine saves state in its own space, it is no longer a pure procedure and cannot be used reentrantly and recursively. Therefore, it is preferable for the caller to furnish a save area or activation record such as a stack. Passing the address of the stack to the subroutine allows an effective callee-save strategy. All state preservation in the JAVA Virtual Machine is implicit. Method invocations store all required state in the Java Stack Frame.

2.4.1 Supervision and I/O

Supervision is necessary for efficiency and reliability. Efficiency requires that the resources of the system - such as memory space, processor time, and peripheral devices - be used by a program no more and no longer than necessary.

Reliability requires that the result of a program be correct in the presence of malfunction. The essential architectural requirement for the supervisor is the ability to seize control from a user program. Because the JAVA Virtual Machine is not intended to be a full physical machine, support for most supervisory functions is absent.

■ **Interlocks:** In a multiprogrammed uniprocessor, the critical-section problem can be solved by disabling the interruption system upon entering the section and re-enabling it upon exiting the section. The semantics of the JAVA language state that in the in the absence of explicit synchronization, a JAVA implementation is free to update the main memory in any order[1].The JAVA Virtual Machine does provide support for explicit syncrhonization for critical section integrity. These instructions are only used within a method. If an entire method is required to be locked, the JAVA Virtual Machine implicitly aquires the lock during resolution and invocation based on information contained within the Constant Pool. Each time a lock is aquired, a lock count is incremented. Each time a lock is released, the lock count is decremented. When the count becomes zero, the current thread releases the monitor. If an exception is thrown while a monitor is aquired, the lock is released. Table 2.33 summarizes the available JAVA Virtual Machine concurrency instructions.

Instruction Name	Function/Syntax	Size/ Opcode
monitorenter	Aquire monitor lock for object *monitorenter*	1 194
monitorexit	Release monitor lock for object *monitorexit*	1 195

Table 2.33: JVM Interlock Instructions

Integrity: The JAVA Virtual Machine contains no support for privileged operations or protected spaces.

Control Switching: The JAVA Virtual Machine contains no support for interruption, humble access or dispatching.

State Saving: The JAVA Virtual Machine contains no support for context switching.

Control: The JAVA Virtual Machine has no concept of a clock or other control mechanisms.

I/O: The JAVA Virtual Machine contains no support for input or output operations.

2.4.2 Conclusion

In this chapter we have given a brief overview of the JAVA Virtual Machine architecture. We have described how stack machines operate in general and also how the stack-based JAVA Virtual Machine operates. We showed how a simple JAVA program is translated to JAVA Virtual Machine bytecode and how it executes. We gave an architectural overview of the JAVA Virtual Machine. We described the storage organization and each of the spaces the JAVA Virtual Machine can access. We described how this memory is accessed and how index arithmetic is computed. We also described all JAVA Virtual Machine operations and the data upon which they operate. Finally, we described how instructions execute and their control structures.

This chapter gave the background for the work which we will describe in the following chapters. We will show how our RISC-based architecture can efficiently translate JAVA Virtual Machine instructions into our instruction set. This chapter also laid the foundation to understand why certain techniques which we have developed to accelerate JAVA execution. The following chapters will explain in detail the techniques we have developed as a result of our research.

If you believe that something is impossible, do not disturb the person who is doing it. – Albert Einstein.

Chapter 3

Delft-Java Architecture

In the previous chapters we gave an introduction to the JAVA Virtual Machine and discussed previous research on JAVA acceleration. This chapter is dedicated to the description of the DELFT-JAVA architecture - a 32-bit RISC-based architecture. More specifically we describe how we accelerate JAVA execution and provide details of the DELFT-JAVA architecture for executing JAVA Virtual Machine bytecodes. Before we begin our discussion we briefly describe the design philosophy underlying our approach. The basic architecture implements a Media Processor with Signal Processing capabilities. The architecture takes the perspective that to maximally accelerate a compiled application, the machine language should accurately reflect the type of operations the compiler specifies. Except where JAVA Virtual Machine operations are unusually complex, we prefer to allow the compiler to optimize directly to the implementation. This is independent of any particular organization. The architecture is then a superset of the JAVA Virtual Machine and provides operations that are necessary for system execution (e.g. I/O, supervision, etc.). Rather than just supporting the JAVA Virtual Machine, the architecture takes a more general purpose approach. While it continues to support JAVA Virtual Machine specific constructs, it also is intended to be programmed from a number of additional high-level languages including C and C++.

Dynamic instruction translation, a new approach to JAVA hardware acceleration, is also used in the DELFT-JAVA processor. In hardware assisted dynamic translation, JAVA Virtual Machine instructions are translated on-the-fly into the DELFT-JAVA instruction set. The hardware requirements to perform this translation are not excessive. Consequently, support for JAVA language constructs are also incorporated into the processor's ISA. This technique allows application level parallelism inherent in the JAVA language to be efficiently utilized as instruction level parallelism while providing support for other common programming languages such as C/C++. In addition to dynamic translation, a special Link Translation Buffer (LTB), discussed in the next chapter, can be used to improve the performance of dynamic linking.

In addition to the basic RISC design philosophy, there are some key organization structures which we deem appropriate to provide architectural support for. In particular, we support the following important categories:

- Synchronization for multithreaded organizations
- garbage collection
- array bounds checking
- real-time caches
- multiple machines which can time-share the same datapath (e.g. the JAVA Virtual Machine and Media Processing functions) and
- vector operations

The chapter is organized in a manner consistent with Chapter 2, JVM : Brief Introduction. First we describe our storage organization including index arithmetic operations. Next we describe the operations which our machine can perform. We then describe how instructions execute on our machine. Finally, we present some conclusions.

3.1 Memory (Storage) Organization

Instructions and their operands must be obtained from a storage space or an input source; the results are placed in a storage space or an output sink. From a machine-language view, I/O can be treated as a specialized type of storage - read-only or write-only. We distinguish three types of spaces: memory, working store, and control store.

3.1.1 Spaces

■ **Memory Space:** The *memory space* from which programs are directly executed is augmented with auxiliary store or secondary store which appears jointly in the DELFT-JAVA architecture as a single contiguous memory space. The DELFT-JAVA architecture has no embedding of address spaces.

■ **Working Store:** In the DELFT-JAVA RISC-style instructions, the *working store* are the following registers.

1. General Purpose Register File (r): This space is a 32 entry by 32-bit register file. An 8-bit datatype occupies bits 7..0. A 64-bit entry occupies an odd/even register pair with the odd register containing the MSB.

2. Instruction Address Base Registers ($ibase$): This file contains 32 entries of a $u32$ datatype. The purpose of an instruction address base register is to allow sharing of common subroutines between tasks and to provide namespace protection among tasks. They can only be written under supervisory control. $ibase0$ always reads zero and may only be accessed in privileged mode.

3. Data Address Base Registers ($dbase$): This file contains 32 entries of a $u32$ datatype. The purpose of a data address base register is to allow sharing of common data between tasks and to provide namespace protection among tasks. They can only be written under supervisory control. $dbase0$ always reads zero and may only be accessed in privileged mode.

■ **Control Store:** The *control store* is the storage that contains the status of the DELFT-JAVA processor, and the information to control the syntactic and semantic interpretation process.

1. Control Register File ($ctl0$): This space is a 32 entry by 32-bit register file. The file holds control information necessary for the proper operation of the machine. The file is only accessible by the supervisor. Some fields available within the Control Registers are:

   - The *Instruction Address Register (IAR)* controls the flow of program execution. Branch conditions and other control instructions may modify the IAR.
   - The Processor Status Word (PSW) .
   - Stack Pointers
   - Limit Registers
   - Clock Counts

3.1.2 Storage Access

In the DELFT-JAVA processor, a set of successive integers as addresses is assigned as the name-space of specific objects. This provides an isomorphic mapping between the set of all possible n-bit names and the set of binary integers from 0 to $2^n - 1$. This constitutes a dense, ordered, and measured set[1]. This allows the same mechanisms used for operations on data to be used for comparisons and additions desirable for names. The *address-set* structure is linear with detection of addresses beyond the ends of the installed segment. This ensures that an increase of memory will not affect correct execution of programs. The minimal *memory address resolution* is an 8-bit byte. The byte ordering convention is *big-endian.* Big-endian is chosen because of the logical convention of treating the whole storage space as one stream of bits. Bits, bytes, half-words, etc. are numbered from left to right. The memory *alignment* is aligned with the memory space being manipulated. It is recommended that compilers generate code that is aligned with the datatype being manipulated. Some language systems (e.g. FORTRAN) forbid alignment. However, it is highly recommended to align values where ever possible.

■ **Address Format:** The architecture permits generalized use of (both indirect and direct) a *three-address* operand format. In general, a three-address operand format is the most natural since most operations are dyadic. However, the three-address format is costly in bits even though the operands are in abbreviated address register files. Because memory addresses are costly in bits, the working store is used as the source and destination of an operation.

■ **Datatypes:** As shown in Table 3.1, the DELFT-JAVA architecture offers a number of useful datatypes. Each instruction must specify the type of its operands. An expression remains syntactically correct if an operand is replaced by another operand from the same set and/or if the operation is replaced by an operation of the same set. One datatype may be a subset of another if the corresponding representands are a subset of the other and/or if the corresponding operation set is a subset of the other. The architecture determines the datatype of the operands in a specific manner. The operation code gives the operation to be performed. An extension field gives the type of the operation, and whether a vector datatype is specified. The architecture's basic unit system is the 8-bit byte.

[1]By measured we mean that the successor of a name can be calculated by addition.

Type	Interpretation	Packed Form	Java Type
$u8$	1-byte unsigned integer	$u8[8]$	n/a
$w8$	1-byte signed 2's complement integer	$w8[8]$	$byte$
$u16$	2-byte unsigned integer	$u16[4]$	$char$
$w16$	2-byte signed 2's complement integer	$w16[4]$	$short$
$u32$	4-byte unsigned	$u32[2]$	$reference$
$w32$	4-byte signed 2's complement	$w32[2]$	int
$u64$	8-byte unsigned		n/a
$w64$	8-byte signed 2's complement		$long$
$f32$	4-byte IEEE 754 single-precision	$f32[2]$	$float$
$f64$	8-byte IEEE 754 double-precision float		$double$

Table 3.1: DELFT-JAVA Datatypes.

■ **Data Length:** The architecture is comprised of a collection of fixed-length datatypes. The data length is specified in an instruction explicitly by the type field, the operation performed, the working store that is used, and the machine view selected.

■ **Address Phrase:** The fundamental address phrase in the architecture is Base, Offset, Displacement addressing. In addition, a stack addressing mode is provided for state saving. A software stack can, of course, be built from the basic address components. The state-saving stacks provide object-code compatible mechanisms that scale with main memory bandwidth. Very low cost addressing can be achieved noting that $dbase0$ always reads zero. This allows privileged code to provide non-additive register direct addresses.

■ **Address Modes:** The architecture provides the following addressing modes:

- Base + Register Offset + Register Displacement; Offset += Displacement
  - If Base=$dbase0$, this is a true register direct address.
- Base + Register Offset + Register Displacement
  - If Base=$dbase0$, this is a true register direct address.
- Base + Register Offset + Immediate Displacement; Offset += Register Displacement
  - If Base=$dbase0$, this is a true direct address.

- Base + Register Offset + Immediate Displacement
  - If Base=$dbase0$, this is a true direct address.
- Base + Register Offset; Offset += Immediate Displacement
  - If Base=$dbase0$, this is a true direct address.
- Stack State Saving using push/pop range.

■ **Address Length:** The memory name-space size of the DELFT-JAVA architecture is 2^{32}.

■ **Names:** In programming languages, objects are identified by names. The *name* may refer to a single object but most names refer to groups of data and instructions. The *address* is the corresponding machine-language name. The *memory location* is the place where a programmer defined name is stored. *Binding* is the process of mapping a programmer defined name to an address. The address is then interpreted at execute time to refer to the memory location. Because the set of programs whose data are simultaneously in memory may change, it is desirable not to have a fixed correspondence between the programmer defined name and the object's actual location in memory. The *address calculation* computes an element of an array or matrix by taking the array name and adding an index. This is called *address modification* when the address calculation takes place as part of instruction execution. The result of the calculation is the *effective address*.

Address Components: The architecture provides all three base, element, and displacement addresses.

Effective Address Calculation: All components of the effective address are added.

Location of Address Components: The base component is a separate quantity and in a supervisor protected space. The element address resides in a general purpose register. The displacement never changes and is properly placed in the address field of the instruction. Because of the number of bits required for large displacements, the architecture has a provision to place the displacement in a register as well as an option for a small displacement in the address field.

■ **Index Arithmetic:** Index arithmetic takes place in either of the general purpose register domains. All integer operations available for normal computations are available for index arithmetic. The architecture provides state-saving stack addressing. In addition, a software stack can be obtained by the general addressing mechanisms. The stack is logically in memory. The top of the stack and its maximum extent are specified by pointers. An exception (trap) is thrown when an attempt is made to pop from an empty stack or to push onto a full one.

Incrementing: All incrementing, whether for general index operations or stack operations takes place in registers. The index for the stack is called the *stack pointer.* The stack is implied to grow from high to low memory addresses. The stack pointer points to the top element. A pop operation becomes a Read followed by an increment corresponding to the size of the data just read. This is a postincrement. A push is a write preceded by a decrement, the predecrement.

■ **Address Levels:** Addresses which refer directly to the machine-language names for data are called *direct addresses.* An address which refers to another address rather than the machine-language name is called *indirect address.* There are no indirect addresses in the architecture. An *immediate address* is an address that does not name a data item but is itself used as the data item. Immediate addressing is only proper for loading registers.

Operation Types	**Example Instructions**
Data Transfer	load, store, mv
Arithmetic	add, add.sat, add.w16[4].sat
Logical	and, or
Transfer of Control	beq, bgt, fbeq
I/O	ior, iow
Conversion	i2f, d2l, f2d, i2l
System Control	scall, wait
Synchronization	csa
Dynamic Linking	invokevirtual, getfield
Object Allocation	new, newarray
Exceptions	athrow

Table 3.2: DELFT-JAVA Operation Categories

3.2 Operations

In a general purpose machine, there is little merit in specifying operations that compilers can not generate. The compiler is most likely to use an operation repertoire that satisfies the requirements of generality, orthogonality, and parsimony. The architecture makes allowances for certain application-specific operations that accelerate JAVA Virtual Machine operations (e.g. method invocation, synchronization, garbage collection, etc.). It also makes allowances for DSP-specific operations (e.g. general bit shifting, extract, rotate) and Multimedia processing (SIMD operations). Because JAVA does not require extended precision, our architecture does not provide secondary operations (e.g. indicators such as carry and overflow). For DSP-specific functions, a saturating type is provided for both signed and unsigned integers so that secondary operations are not required. The category of operations in our architecture is shown in Table 3.2 along with some representative sample instructions.

Our architecture uses a partially decomposed specification. Operations are decomposed into *actions* - (e.g. Add, Logic, etc.) and *modifiers* are used to select datatypes. All operations are specified using a fixed-length sequence of 32-bits. However, certain JAVA specific instructions are greater than 32-bits. These instructions trap and the instruction address is updated to reflect the actual length This allows for the instruction address to be updated prior to decoding any particular instruction. In all formats, the operation code is a fixed-length of 7-bits. Furthermore, the operation code is always found within the same bits of all the instruction formats.

We note that throughout this section any instruction which references a direct register (e.g. r_t) may also use it's indirect form (e.g. $[idx_n]$ i_t) where $[idx_n]$ is the index register offset file and i_t is the target index. For example, in Table 3.3 and Table 3.4, the ld instruction could be re-written as $[idx_n]$ {iT3.} i_t = Mem[$dbase_n$ + i_{xo} + i_{yd}] i_x += i_y. This instruction would then be interpreted as select the indirect register file location $[idx_n]$ where n can be from 0 to 7. Assign the target register that is dereferenced by idx_n to the contents of the effective address formed by adding the $dbase_n$ registers with the i_{xo} and the i_{yd} registers. Following the formation of the effective address, the indirect offset register is incremented contents of the indirect displacement register. The {iT3.} specifies that 3-bits of type information are available. The curly braces mean a value is optional. If no value is specified, a default value is selected.

■ **Data Handling:** Data-handling operations are those that do nothing else -

no arithmetic or logic. That is register-to-register moves, loads from memory, etc.

Data movement corresponds to the programming-language assignment primitive. The DELFT-JAVA architecture provides data movement operations via working store in the form of register to register moves, Load and Store operations. There is also a special form which allows a group of registers to be moved to a stack. A register to register move is accomplished by restricting the datatypes in the conversion functions. The supervisor may also move control registers. The 32-bit instructions can access up to 32 locations. All offset and displacement registers are defined to be in the general purpose register file r and may be accessed as direct or indirect register references. The type is $u32$. Up to 32 $dbase$ registers may be specified. A conditional move operation is provided in the DELFT-JAVA architecture. The conditions are the same as in Table 3.21. Tables 3.3 and 3.4 summarize the available data movement instructions.

Name	**Function**
ld	Load base + r[offset] + r[disp]; off += disp
ld.rdisp	Load base + r[offset] + r[disp]
ld.disp	Load base + r[offset] + imm16 displacement
ld.off	Load base + r[offset]; off += disp
ldi	Load imm16
st	Store base + r[offset] + r[disp]; off += disp
st.rdisp	Store base + r[offset] + r[disp]
st.disp	Store base + r[offset] + simm16 displacement
st.off	Store base + r[offset]; off += disp
mv	Move register to register
cmv	Conditional Move register to register
mvc	Move Control Register

Table 3.3: DELFT-JAVA Load and Store Operations Function.

Name	Assembly Syntax
ld	$\{$iT3.$\}r_t$ = Mem[$dbase_n$ + r_{xo} + r_{yd}] r_x += r_y
ld.rdisp	$\{$iT3.$\}r_t$ = Mem[$dbase_n$ + r_{xo} + r_{yd}]
ld.disp	$\{$iT3.$\}r_t$ = Mem[$dbase_n$ + r_{xo} + #imm16]
ld.off	$\{$iT3.$\}r_t$ = Mem[$dbase_n$ + r_{xo}] r_x += r_y
ldi	$\{$iT4.$\}r_t$ = #imm16
st	Mem[$dbase_n$ + r_{xo} + r_{yd}] = $r_t\{$.iT3$\}$; r_x += r_y
st.rdisp	Mem[$dbase_n$ + r_{xo} + r_{yd}] = $r_t\{$.iT3$\}$
st.disp	Mem[$dbase_n$ + r_{xo} + #imm16] = $r_t\{$.iT3$\}$
st.off	Mem[$dbase_n$ + r_{xo}] = $r_t\{$.iT3$\}$; r_x += r_y
mv	r_t = r_x
cmv	(if cond) then $\{$iT3.$\}r_t$ = r_x
mvc	r_t = ctl or ctl = r_t

Table 3.4: DELFT-JAVA Load and Store Operations Syntax.

Format Transformation: Reallocating operators transform one machine field into another without changing the encoding. Operations are provided to convert datatypes to smaller or larger lengths while maintaining their encoding (e.g. converting a $u8$ to $u32$, $w16$ to $w64$, or $u16$ to $u8$). In some cases the bits are truncated. In other cases, they are zero- or sign-extended. Format transformations are accomplished by restricting the domains and types of the general code transformation instructions. Table 3.5 summarizes the available format transformation instructions.

Name	Function	Assembly Syntax
cvtii	Convert Integer to Integer	$\{$iT3.$\}r_t$ = $r_x\{$.iT3$\}$
cvtff	Convert Floating Point to Floating Point	$\{$fT2.$\}r_t$ = $r_x\{$.fT2$\}$

Table 3.5: DELFT-JAVA Format Transformation Operations.

Code Transformation: One data representation in a machine language can be transformed to another representation if the concept sets that the two represent intersect. The architecture allows general transformations between floating point, integer, and unsigned integer representations. Table 3.6 summarizes the available code transformation instructions.

Name	**Function**	**Assembly Syntax**
cvtif	Convert Integer to Floating Point	{fT2.}r_t = r_x{.iT3}
cvtfi	Convert Floating Point to Integer	{iT3.}r_t = r_x{.fT2}

Table 3.6: DELFT-JAVA Data Code Transformation Operations.

■ **Logic:** A logical (boolean) datum has two possible states - true and false. It can be represented by a single bit. In the architecture, a logical datum is encoded as a binary value with a true state represented by a 1 and false by 0. The allocation for logical operations is as a vector of booleans. The architecture representation includes the Encoding and Allocation. Logical data is represented by a vector. All logical operations apply to vectors of bits. There is no direct boolean datatype of length 1-bit. There are however logical vectors.

Connectives: The dyadic operations upon a single pair of bits are called *connectives*. The architecture supplies 8 connectives (versus the generic 16). A ninth (not) can be synthesized from the nAorB connective with the B value being an immediate containing zero. Connectives are inherently unsigned. Note, however, that the immediate fields are sign extended. This is to provide an all 1's mask. Table 3.7 summarizes the available logic connective instructions.

Name	**Function**	**Assembly Syntax**
and	$A \wedge B$	{uT2.}r_t = r_y & $r_x$$\vee$#simm5
nab	$\overline{A} \wedge B$	{uT2.}r_t = ˜r_y & $r_x$$\vee$#simm5
nand	$\overline{A \wedge B}$	{uT2.}r_t = ˜(r_y & $r_x$$\vee$#simm5)
naOrb	$\overline{A} \vee B$	{uT2.}r_t = ˜r_y | $r_x$$\vee$#simm5
nor	$\overline{A \vee B}$	{uT2.}r_t = ˜(r_y | $r_x$$\vee$#simm5)
or	$A \vee B$	{uT2.}r_t = r_y | $r_x$$\vee$#simm5
xnor	$\overline{A \oplus B}$	{uT2.}r_t = ˜(r_y ˆ$r_x$$\vee$#simm5)
xor	$A \oplus B$	{uT2.}r_t = r_y ˆ$r_x$$\vee$#simm5
not	`naOrb(` A, #0 `)`	{uT2.}r_t = ˜r_y | #0

Table 3.7: DELFT-JAVA Logical Connective Operations.

Composite Functions: A composite logical function applies a dyadic scalar function among the elements of a bit vector. A population count which is an add reduction is a popular composite function which can be used as a primitive along with other logical functions to find the first bit set. A scan function is provided to give this result directly. Rather than shifting the number of bits set, a normalize function is provided. Table 3.8 summarizes the available logical composite functions.

Name	Function	Assembly Syntax
scan.0	Count Leading Zeros	$\{$uT2.$\}r_t$ = clz(r_x)
scan.1	Count Leading Ones	$\{$uT2.$\}r_t$ = clo(r_x)
norm	Normalize	$\{$uT2.$\}r_t$ = norm(r_x)

Table 3.8: DELFT-JAVA Composite Logic Operations.

Shift Operations: The shift is used in data handling for field selection; in logic, for bit inspection, and in arithmetic, for programmed scaling, multiply, divide, and floating point. The architecture provides a rich set of shift functions including arithmetic shift, logical shift, rotate, and rotate with carry. The shift displacements can either be an immediate field or contained within another register. The results may also be saturated. Table 3.9 summarizes the available shift instructions.

Name	Function	Assembly Syntax
sll	Shift Logical Left	$\{$uT2.$\}r_t = r_y <<< r_x \vee$#imm8
sll.sat	Shift Logical Left with Saturate	$\{$uT2.$\}r_t$ = sat($r_y <<< r_x \vee$#imm8)
slr	Shift Logical Right	$\{$uT2.$\}r_t = r_y >>> r_x \vee$#imm8
sal	Shift Arithmetic Left	$\{$wT2.$\}r_t = r_y << r_x \vee$#imm8
sal.sat	Shift Arithmetic Left and Saturate	$\{$wT2.$\}r_t$ = sat($r_y << r_x \vee$#imm8)
sar	Shift Arithmetic Right	$\{$wT2.$\}r_t = r_y >> r_x \vee$#imm8
sar.sat	Shift Arithmetic Right and Saturate	$\{$wT2.$\}r_t$ = sat($r_y >> r_x \vee$#imm8)
ror	Rotate Right	$\{$uT2.$\}r_t = r_y$ ^> $r_x \vee$#imm8
rol	Rotate Left	$\{$p==z$\}\{$uT2.$\}r_t = r_y$ ^< $r_x \vee$#imm8

Table 3.9: DELFT-JAVA Shift Operations.

Bit Manipulations: A rich set of bit manipulation operations exist for DSP-specific operations. Bit manipulations are defined only for the $u32$ datatype with some limited support for the $u16$ datatype. While this is an impropriety, it is the best choice that can be implied for a signal processing datatype. The bit fields are allowed to be set, cleared, or extracted from a starting and ending position. The bits to be operated upon are specified by two 5-bit immediate fields within the instruction formats. Bit testing is also provided for. Table 3.10 summarizes the available bit manipulation instructions.

Name	**Function**	**Assembly Syntax**
bic	Clear bit field	$\{$uT.$\}r_t$ = bic(r_x, r_t, start, stop)
bis	Set bit field	$\{$uT.$\}r_t$ = bis(r_x, r_t, start, stop)
bit	Test bit field	r_t = bit(r_x, start, stop)
bix	Extract bit field	$\{$uT.$\}r_t$ = bix(r_x, start, stop, r_t, start, stop)

Table 3.10: DELFT-JAVA Bit Operations.

■ **Fixed-Point Arithmetic:** The representation of fixed point numbers includes the number system choice, the allocation of elements, and the element representation. The architecture uses a positional representation for the number system. Two types of fixed-point numbers can be represented: positive whole numbers which include zero (termed positive integers or unsigned integers) and integers. The notation for positive integers uses a binary radix with an implied sign of zero in the most significant (hidden) bit. The position of the radix point is just to the right of the digits. For integers, the notation of negative numbers uses *radix complement*. The radix value of our representation is 2 and our notation is 2's complement. The position of the radix point is just to the right of the digits - integer notation. Because the high-order digit (which denotes the sign) participates fully in arithmetic, multiple precision low-order numbers can be treated as all digits. The implied position of the radix point matters only for complement notations. In the complement notations, the sign is treated as a digit. For a radix 2 system, each digit requires 1 bit. The number of digits is fixed based on the datatype. For addresses, all lengths are 32-bits. The length of data is explicitly given by an instruction field. For unsigned integers, the sign is hidden in the most significant bit (it is implicit and preceding the left-most bit) and implied to be 0 (e.g. positive). For integers, the sign is encoded as required for radix complement notations (0 for positive and 1 for minus). The sign bit is left aligned into the most significant bit. The allocation is one-to-one in that the number of digits determines the length of the number in bits. Digits are encoded as 0 for zero and 1 for One. Scalar operations are provided for absolute value, add, subtract, and multiply.

Absolute Value: Table 3.11 summarizes the absolute value instruction.

Name	**Function**	**Assembly Syntax**
abs	Absolute Value	$\{$wT2.$\}r_t$ = abs(r_x)

Table 3.11: DELFT-JAVA Absolute Value Operation.

Addition and Subtraction: When operands and results have equal lengths, an overflow can always be represented by 1 bit. However, the architecture does not provide support for obtaining overflow information (since both JAVA and C/C++ specify modulo arithmetic). For DSP operations, the result may be saturated. The instructions can specify rounding (including convergent rounding) as well as saturation and type information. A special butterfly operation is provided which is useful in certain signal processing applications. To be able to perform polynomial arithmetic, a double width result may be specified on selected instructions (provided the maximum result width is not greater than 64-bits). Table 3.12 and Table 3.13 summarizes the available addition and subtraction instructions.

Name	Function
add	Add
add.rnd	Add and Round
add.sat	Add and Saturate
sub	Subtract
sub.rnd	Subtract and Round
sub.sat	Subtract and Saturate
sub2	Negate / Sub
sub2.rnd	Negate / Sub and Round
sub2.sat	Negate / Sub and Saturate
bfly	Butterfly
bfly.rnd	Butterfly and Round
bfly.sat	Butterfly and Saturate

Table 3.12: DELFT-JAVA Add Operations Function.

Name	Assembly Syntax
add	$\{$2x.$\}\{$T5.$\}r_t$ = r_y + $r_x\vee$#imm8
add.rnd	$\{$2x.$\}\{$T5.$\}r_t$ = rnd(r_y + $r_x\vee$#imm8)
add.sat	$\{$2x.$\}\{$T5.$\}r_t$ = sat(r_y + $r_x\vee$#imm8)
sub	$\{$2x.$\}\{$T5.$\}r_t$ = r_y - $r_x\vee$#imm8
sub.rnd	$\{$2x.$\}\{$T5.$\}r_t$ = rnd(r_y - $r_x\vee$#imm8)
sub.sat	$\{$2x.$\}\{$T5.$\}r_t$ = sat(r_y - $r_x\vee$#imm8)
sub2	$\{$2x.$\}\{$T5.$\}r_t$ = $r_x\vee$#simm8 - r_y
sub2.rnd	$\{$2x.$\}\{$T5.$\}r_t$ = rnd($r_x\vee$#simm8 - r_y)
sub2.sat	$\{$2x.$\}\{$T5.$\}r_t$ = sat($r_x\vee$#simm8 - r_y)
bfly	$\{$2x.$\}\{$T5.$\}r_t$H = r_y + $r_x\vee$#imm8; r_tL = r_y - $r_x\vee$#imm8
bfly.rnd	$\{$2x.$\}\{$T5.$\}r_t$H = rnd(r_y + $r_x\vee$#imm8); r_tL = rnd(r_y - $r_x\vee$#imm8)
bfly.sat	$\{$2x.$\}\{$T5.$\}r_t$H = sat(r_y + $r_x\vee$#imm8); r_tL = sat(r_y - $r_x\vee$#imm8)

Table 3.13: DELFT-JAVA Add Operations Syntax.

Multiplication and Division: For multiplication, the number of product digits is just less than or equal to the total number of digits of the multiplier and the multiplicand. Most instructions in the architecture consider the resultant destination to be the same length as the operands. This is as per the C language convention. However, a double width result may be specified on selected instructions (provided the maximum result width is not greater than 64-bits). The result can also be saturated. The 3-address formats automatically expand the result to $2n$ bits. Tables 3.14 and 3.15 summarize the available multiplicative instructions.

Name	**Function**
mpy	Multiply
mpy.rnd	Multiply and Round
mpy.sat	Multiply and Saturate
div	Divide
mac	Multiply and Accumulate
mac.rnd	Multiply and Accumulate and Round
mac.sat	Multiply and Accumulate and Saturate
rem	Remainder

Table 3.14: DELFT-JAVA Multiply/Divide Operations.

Name	**Assembly Syntax**
mpy	{2x.}{T4.}r_t = r_y * r_x∨#imm8
mpy.rnd	{2x.}{T4.}r_t = rnd(r_y * r_x∨#imm8)
mpy.sat	{2x.}{T4.}r_t = sat(r_y * r_x∨#imm8)
div	{2x.}{T4.}r_t = r_y / r_x∨#imm8
mac	{2x.}{T4.}r_t = r_t + r_y * r_x∨#imm8
mac.rnd	{2x.}{T4.}r_t = rnd(r_t + r_y * r_x∨#imm8)
mac.sat	{2x.}{T4.}r_t = sat(r_t + r_y * r_x∨#imm8)
rem	{2x.}{T4.}r_t = rem(r_y / r_x∨#imm8)

Table 3.15: DELFT-JAVA Multiply/Divide Operations.

■ **Floating Point:** All floating-point operations follow the IEEE-754-1985 standard as per the JAVA Virtual Machine specification. Inexact results must be rounded to the representable value nearest to the infinitely precise result. Round-towards-zero (the default rounding) effectively truncates the mantissa. Floating point operations produce no exceptions. An overflow produces a signed infinity, an underflow produces a signed zero, and an indefinite result produces NaN.

Addition and Subtraction: For addition, the following IEEE rules apply:

1. if either value is NaN, the result is NaN
2. the sum of two infinities of opposite sign is NaN
3. the sum of two infinities of the same sign is the infinity of that sign
4. the sum of infinity and a finite value is infinity
5. the sum of two zeroes of opposite sign is positive zero
6. the sum of two zeroes of the same sign is the zero of that sign
7. the sum of a zero and a nonzero finite value is equal to the non-zero value
8. the sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.

Moreover, all sums are computed and rounded using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign.

For subtraction, the result of $a - b$ is the same as $a + (-b)$ except for the case of subtraction of 0.0. For example, $0.0 - 0.0 = +0.0$ while $0.0 + (-0.0) = -0.0$.

For negation, the following additional IEEE rules apply:

1. the negation of an infinity is the infinity of opposite sign
2. the negation of a zero is the zero of opposite sign.

Table 3.12 summarizes the available DELFT-JAVA floating point instructions. They are a superset of the JAVA Virtual Machine addition and subtraction operations.

Multiplication and Division: For multiplication, the following IEEE-754 rules apply:

1. if either operand is a NaN, the result is NaN
2. if neither operand is NaN, the sign of the result is positive if both operands have the same sign, and negative the operands have different signs

3. multiplication of infinity by a zero is NaN
4. multiplication of an infinity by a finite operand produces an infinity following the sign rule given above.

Moreover, the product is computed and rounded using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign.

For division, the following IEEE-754 rules apply:

1. if either operand is a NaN, the result is NaN
2. if neither operand is NaN, the sign of the result is positive if both operands have the same sign, and negative the operands have different signs
3. division of infinity by infinity is NaN
4. division of infinity by a finite value produces an infinity following the sign rule given above
5. division of a finite value by an infinity produces a zero following the sign rule given above
6. division of a zero by a zero results in NaN
7. division of zero by a finite value produces a zero following the sign rule given above
8. division of a nonzero finite value by a zero produces an infinity following the sign rule given above.

Moreover, the quotient is produced using round-to-nearest mode. If an overflow occurs, the result is the infinity of the appropriate sign. If an underflow occurs, the result is a zero of the appropriate sign. For remainder, the definition is *not* the same as the IEEE-754 version. The IEEE version uses rounding division while the DELFT-JAVA uses truncating division (to more closely approximate the integer behavior). This was required to be consistent with the JAVA Virtual Machine rules for division.

The following rules apply:

1. if either operand is a NaN, the result is NaN

2. if neither value is NaN, the sign of the result is equals the sign of the dividend
3. if the dividend is an infinity, or the divisor is a zero, the result is NaN
4. if the dividend is finite and the divisor is an infinity, the result equals the dividend
5. if the dividend is a zero and the divisor is finite, the result equals the dividend.

Moreover, the result is $dividend - intof((\frac{dividend}{divisor}) * divisor)$. The function $intof()$ rounds toward the nearest integer, or towards the nearest even integer if the number is half way between two integers.

Table 3.14 summarizes the available multiplication and division instructions. They are a superset of the JAVA Virtual Machine addition and subtraction operations.

■ **Relational Operations:** Relational operations test a specified relation among operands and produce a result that is true or false. A relation can formally be considered to be a mapping from an input domain, consisting of all possible values of the datum, to an output space or range, consisting of one point for each category. Generally, a compare instruction is provided for this purpose. The architecture provides comparisons for types all types. Table 3.17 shows the result of a comparison:

Name	**Function**	**Assembly Syntax**
cmp	Compare	$\{$T5.$\}r_t \leftarrow$ result, $r_y <> r_x \vee$#imm8

Table 3.16: DELFT-JAVA Compare Operation.

Comparison	**Result**
$>$	0x01
$=$	0x00
$<$	0xFF
NaN	0x10
$<=$	0x0F
$>=$	0xF1
$+0.0 = -0.0$	0x02

Table 3.17: DELFT-JAVA Comparison Results

The architecture uses a ranked comparison (e.g. <,=,> membership). Comparisons can be made on any of the general purpose registers r domain. Results are recorded in working store.

For floating point relational operations, the following rules apply:

1. positive zero and negative zero are equivalent
2. negative infinity is less than positive infinity
3. a NaN is unordered and can be determined from a combination of fcmpg and fcmpl instructions.

Table 3.16 summarizes the available relational instructions.

■ **Numeric SIMD Operations:** All arithmetic on the r domain can be considered to apply to a Single Instruction Multiple Data (SIMD) datatype. The maximum length of the SIMD array is fixed architecturally at 64-bits. SIMD arrays logically take up an even/odd register pair. For all SIMD operations, the destination operand must be distinct from the source operand. Any arithmetic instruction can be made a SIMD instruction by prepending a v (e.g. vadd, vsub, vmpy, vcvtii, etc.). There is one SIMD-only operation. The permute function rearranges the datatype within the array by a specified function. The vector length is determined by the type specified in the instruction format. The valid permutations are $P^{vlen}_{2^{imm4}}$ with an $imm4 = 000$ being interpreted as $I_{\frac{vlen}{4}} \otimes P^4_2$ operation. The permute function may be applied to any SIMD datatype but for the purposes of the operation, the data is treated as unsigned. To avoid a large internal storage requirement, the source and destination vectors must be distinct.

Name	**Function**	**Assembly Syntax**
perm	Permute Vector	`{utype.}`r_t `= P(imm3,`r_x`)` `{cex=z}`

Table 3.18: DELFT-JAVA Vector Permute Operation.

3.3 Instruction Execution

The architecture increments the *instruction address* by 4-bytes each cycle. In the architecture, all instructions formats are 32-bits. However, certain JAVA specific instructions are greater than 32-bits. These instructions trap and the instruction address is updated to reflect the actual length. The status word of the architecture consists of uniquely named registers.

■ **Instruction Sequence:** An instruction sequence requires a specification and a normal sequence. A normal sequence requires selecting a Continuity and a Choice. The Continuity is partitioned into a linear sequence and a delegation. The choice can be a decision or iteration.

Linear Sequence: The simplest structure is the linear sequence - or vector arrangement - of instructions. When instructions are arranged in a vector, each can be identified by the vector index of its position in memory - its address. The address where an instruction resides is called its *location*. The design choices for a linear sequence include Dependence, Next Location, and Completion. Normally, it is desirable that all instructions to be executed constitute independent syntactical units. The JAVA Virtual Machine contains a wide instruction which places semantic dependency upon multiple instructions. In the DELFT-JAVA architecture this instruction causes a trap. The wide instruction is summarized in Table 3.19.

Name	Function	Assembly Syntax
wide	Wide prefix	wide

Table 3.19: DELFT-JAVA Wide Prefix.

Instruction Location: The next instruction is placed linearly in memory, using an implied instruction address, incremented by the length of the instruction.

Completion: A program has a well-defined end. A wait instruction which is summarized in Table 3.20 is provided for this purpose.

Name	Function	Assembly Syntax
wait	Wait (privileged operation)	wait

Table 3.20: DELFT-JAVA Program Completion.

■ **Decision:** The design choices for Decision include the Condition and an Alternative Action. Tables 3.21 and 3.22 summarize the available integer decision instructions. Tables 3.23 and 3.24 summarize the available floating point decision instructions. Tables 3.25 and 3.26 summarize the available trap instructions.

Condition: The condition is decomposed into a general computation with explicit condition followed by a general condition test and the target selection. Indicators that are recorded as the result of a general computation are termed condition codes because they encode secondary operations. A problem with condition codes is that they constitute state. The architecture contains no condition codes. The condition is calculated as part of the total computation.

Alternative Action: Having specified the condition, one must indicate which action corresponds to each of its values. The entails specifying a Branching factor, and a Target Address. Since a CASE statement is not supported directly, all branching factors are two[2]. The exclusive use of user mode relative branches allows a section of program to be relocated without modification. Use of an absolute branch address is available only in privileged mode. All instructions must be aligned on 4-byte boundaries and relative branches are implied 4-byte boundaries. Any valid type may be specified.

[2]A CASE statement (tablelookup) is provided but only so that a trap may occur and the actual instruction specified in the JAVA Virtual Machine may be emulated.

Op	Function	Cond
eq	Branch Relative Equal Zero	0000
nz	Branch Relative Not Zero	0001
gu	Branch Relative Greater Unsigned	0010
cu	Branch Relative Carry Unsigned	0011
geu	Branch Relative Greater or Equal Unsigned	0100
	rsv	0101
	rsv	0110
	rsv	0111
	rsv	1000
n	Branch Relative Never	1001
g	Branch Relative Greater	1010
l	Branch Relative Less	1011
ge	Branch Relative Greater or Equal	1100
le	Branch Relative Less or Equal	1101
	rsv	1110
	rsv	1111

Table 3.21: DELFT-JAVA Relative Branch Conditions with Compare.

Op	Assembly Syntax	Cond
eq	br.eq{.T4} r_y- r_x, $r_t\vee$#simm6[iar]	0000
nz	br.nz{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	0001
gu	br.gu{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	0010
cu	br.cu{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	0011
geu	br.geu{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	0100
		0101
		0110
		0111
		1000
n	br.n{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	1001
g	br.g{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	1010
l	br.l{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	1011
ge	br.ge{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	1100
le	br.le{.T4} r_y- r_x, $r_t\vee$#simm11[iar]	1101
		1110
		1111

Table 3.22: DELFT-JAVA Relative Branch Conditions with Compare.

Name	**Function**	**Cond**
fbeq	Branch Equal Zero	0000
fbnz	Branch Not Zero	0001
fbu	Branch Unordered	0010
fbug	Branch Unordered or Greater	0011
fbul	Branch Unordered or Less	0100
fbue	Branch Unordered or Equal	0101
fbuge	Branch Unordered or Greater or Equal	0110
fbule	Branch Unordered or Less or Equal	0111
fbo	Branch Ordered	1000
		1001
fbg	Branch Greater	1010
fbl	Branch Less	1011
fbge	Branch Greater or Equal	1100
ble	Branch Less or Equal	1101
		1110
		1111

Table 3.23: DELFT-JAVA Floating Point Branch Conditions.

Name	**Assembly Syntax**	**Cond**
fbeq	fbeq $r_y - r_x$, $r_t \vee$#simm6	0000
fbnz	fbnz $r_y - r_x$, $r_t \vee$#simm6	0001
fbu	fbu $r_y - r_x$, $r_t \vee$#simm6	0010
fbug	fbug $r_y - r_x$, $r_t \vee$#simm6	0011
fbul	fbul $r_y - r_x$, $r_t \vee$#simm6	0100
fbue	fbue $r_y - r_x$, $r_t \vee$#simm6	0101
fbuge	fbuge $r_y - r_x$, $r_t \vee$#simm6	0110
fbule	fbule $r_y - r_x$, $r_t \vee$#simm6	0111
fbo	fbo $r_y - r_x$, $r_t \vee$#simm6	1000
		1001
fbg	fbg $r_y - r_x$, $r_t \vee$#simm6	1010
fbl	fbl $r_y - r_x$, $r_t \vee$#simm6	1011
fbge	fbge $r_y - r_x$, $r_t \vee$#simm6	1100
ble	ble $r_y - r_x$, $r_t \vee$#simm6	1101
		1110
		1111

Table 3.24: DELFT-JAVA Floating Point Branch Conditions.

A branch with implied absolute target address is called a *trap*. Traps are used only for exception handling and debug. The address of the trap handler can be specified through an unsigned immediate field or placed in a register. When a trap occurs, the instruction address to return to is atomically pushed onto the supervisor stack.

Name	**Function**	**Assembly Syntax**	**Cond**
teq	Trap Equal Zero	teq r_y, $r_t \vee$#imm12	0000
tnz	Trap Not Zero	tnz r_y, $r_t \vee$#imm12	0001
tgu	Trap Greater Unsigned	tgu r_y, $r_t \vee$#imm12	0010
tcu	Trap Carry Unsigned	tcu r_y, $r_t \vee$#imm12	0011
tgeu	Trap Greater or Equal Unsigned	tgeu r_y, $r_t \vee$#imm12	0100
	rsv		0101
	rsv		0110
	rsv		0111
ta	Trap Always	trap r_y, $r_t \vee$#imm12	1000
debug	Trap Debug	debug r_y, $r_t \vee$#imm12	1001
tg	Trap Greater	tg r_y, $r_t \vee$#imm12	1010
tl	Trap Less	tl r_y, $r_t \vee$#imm12	1011
tge	Trap Greater or Equal	tge r_y, $r_t \vee$#imm12	1100
tle	Trap Less or Equal	tle r_y, $r_t \vee$#imm12	1101
			1110
			1111

Table 3.25: Trap Instructions.

Name	**Function**	**Assembly Syntax**
reti	Return From Interrupt	reti

Table 3.26: DELFT-JAVA Return from Trap/Interrupt Instruction(16-bit).

■ **Iteration:** Iteration involves a scope (what is to be iterated?) and a termination condition (when should iteration be stopped?). In the architecture all iteration is done through conditional branching.

■ **Delegation:** Delegation of control allows a recurring function to be detailed only once and to be called from many places. The architecture provides a *call* instruction and a *ret* (return) instruction for user level subroutine invocation. Supervisory functions are also provided for with the privileged *svc* and *sret* instructions. This provides for fully protected subroutines that only the supervisor can access. Tables 3.27 and 3.28 summarize the available delegation instructions.

Name	Function	Assembly Syntax
call	Call and Link	call i_{base}, $r_y \vee$#imm20
svc	User Request for Supervisor Program	call i_{base}, $r_y \vee$#imm20

Table 3.27: Delegation Operations.

Name	Function	Assembly Syntax
ret	Return From Subroutine	ret
sret	Return From Supervisor call	sret

Table 3.28: DELFT-JAVA Return Operations.

Parameter Passing: Parameters are passed either through registers or memory. When a *call* occurs, the instruction address to return to is atomically pushed onto the user stack which is logically in memory. In addition, a supervisor stack is provided which atomically copies the instruction address to return to onto the supervisor stack whenever an *scall* instruction is executed. Push and pop instructions are provided to pass parameters to a separate supervisor stack. When an interrupt or trap occurs, the instruction address to return to is atomically pushed onto the supervisor stack.

State Preservation: If a subroutine saves state in its own space, it is no longer a pure procedure and cannot be used reentrantly and recursively. Therefore, it is preferable for the caller to furnish a save area or activation record such as a stack. Passing the address of the stack to the subroutine allows an effective callee-save strategy.

3.4 Supervision and I/O

Supervision is necessary for efficiency and reliability. Efficiency requires that the resources of the system - such as memory space, processor time, and peripheral devices - be used by a program no more and no longer than necessary. Reliability requires that the result of a program be correct in the presence of malfunction. The essential architectural requirement for the supervisor is the ability to seize control from a user program. The DELFT-JAVA architecture assumes the presence of a supervisor and provides privileged instructions.

■ **Interlocks:** In a multiprogrammed uniprocessor, the critical-section problem can be solved by disabling the interruption system upon entering the section and re-enabling it upon exiting the section. The architecture provides an atomic Compare And Swap instruction for critical section integrity. Table 3.29 and Table 3.30 summarize the available atomic instructions.

Name	**Function**
csa	Compare and Swap Atomic B/O/D
csa.off	Compare and Swap Atomic Base + Offset
csa.disp	Compare and Swap Atomic Base + ImmDisp

Table 3.29: DELFT-JAVA Atomic Operations Function.

Name	**Assembly Syntax**
csa	{T3.}cmpReg = csa(dbase + off/dispReg, swapReg)
csa.off	{T3.}cmpReg = csa(dbase + offReg, swapReg)
csa.disp	{T3.}cmpReg = csa(dbase + #simm6, swapReg)

Table 3.30: DELFT-JAVA Atomic Operations.

■ **Privileged Operations:** *Privileged operations* are operations that exercise control and are reserved for the supervisor. Capabilities imply that the operands that are capable of control are accessible only to the supervisor. By definition, privileged operations can be invoked only by the supervisory program. The computer must know whether or not a privileged operation is allowed. The mode in which the supervisor has control is termed *privileged mode*. This is distinguished by a bit in the processor status word. Tables 3.31 and 3.32 summarize the available privileged instructions.

Name	**Function**
scall	Supervisor Subroutine Call
rets	Return from Supervisor
wait	Program Completion
mvr2c	Move Register to Control Register

Table 3.31: DELFT-JAVA Privileged Operations.

Name	Function
trap	Trap
reti	Return From Interrupt
mvc2r	Move Control Reg To Register (except PSW)

Table 3.32: DELFT-JAVA Privileged Operations.

In addition to privileged operations, there are also registers which only the supervisor may access. These include the entire set of control registers. Tables 3.33 and 3.34 summarize the DELFT-JAVA privileged registers.

reg	**Function**	**reg**	**Function**
0	$ctrH$ - Cycle Ctr High	16	psw
1	$ctrL$ - Cycle Ctr Low	17	iar
2	$clk0$ - 8 kHz	18	$nextiar$
3	$clk1$ - 9.6 kHz	19	ssp
4	$clk2$ - 44.1 kHz		
5	$mclk$ - Master Clock		
6			
7			

Table 3.33: DELFT-JAVA Control Registers I.

reg	Function	reg	Function
32	$ibase0$	48	$dbase0$
33	$ibase1$	49	$dbase1$
34	$ibase2$	50	$dbase2$
35	$ibase3$	51	$dbase3$
36	$ibase4$	52	Supervisor $sibase$
37	$ibase5$	53	$umemL$ - User Mem limit low
38	$ibase6$	54	$umemH$ - User Mem limit high
39	$ibase7$	55	

Table 3.34: DELFT-JAVA Control Registers II.

■ **Control Switching:** A supervisory program requires three types of control switches: 1) Dispatch where the supervisor gives control to the user, 2) Humble Access where the user yields control, and 3) Interruption where the supervisor takes control.

Interruption: An *interruption* is a control switch away from the program under execution to another program - almost invariably the supervisor.

Dispatching: A program is said to be *active* if it has requested system use. It is said to be *inactive* otherwise. A program that is actually placed in memory is called *entered*; the others are called *not entered*. In a uniprocessor, only one of the programs that are entered is executing at a given moment; its state is called *executing*. The others may be in a *ready* state where they are waiting for the processor or they may be in a *not ready* state waiting for input/output. The supervisor *dispatches* a program on a processor when it changes that program's state to executing. A switch from the supervisor to a user program is always initiated by the supervisor. This is accomplished by the privileged load of the Instruction Address Register in the Control Registers.

Humble Access: The user program yields control to the supervisor through the Supervisor Call (SVC) instruction. It is important that the supervisor - not the user program - specify the point at which the supervisor starts execution.

■ **State Saving:** The cause of an asynchronous interruption is independent of the program that is in execution. The *state* of a program is defined by the content of its storage spaces - the used parts of memory, working, and control store. A *program context* is all the storage spaces that are time-shared when one program is switched to executing another program.

Context Switching: Context switching overhead is alleviated by using push and pop range commands.

■ **Control:** The clock controls the operation of the processor and provides the master control signal which synchronizes all events.

3.5 Java Specific Operations

The DELFT-JAVA architecture supports some JAVA Virtual Machine specific instructions. These instructions are inherent in the DELFT-JAVA architecture. Thus, it allows the machine to maintain the high-level information contained in the operation and either emulate or execute the instruction based on a particular implementation's performance requirements. Instructions which are more than 4 bytes in length trap and are architecturally defined to be emulated in a trap handler. Table 3.35 summarizes the available DELFT-JAVA specific JAVA Virtual Machine instructions.

In addition to JAVA Virtual Machine specific instructions, additional support is provided for microarchitectural features that are useful to accelerate JAVA language constructions.

■ **Link Translation Buffer**

The Java Virtual Machine [3] contains support for run-time bound method invocation. Because the Delft-Java processor incorporates invocation instructions directly into it's ISA, architecturally transparent techniques can be used to accelerate dynamic linking and method invocation. In this section we briefly describe method resolution and invocation. We then introduce an architecturally transparent technique, the Link Translation Buffer, and explain its operation.

Method Invocation: In a Java program, the Constant Pool contains the names of the methods to be invoked. These are stored as strings and function much like a symbol table. When a method is invoked, the Java Virtual Machine searches the Constant Pool for the name of the method to invoke. Then, based on the run-time type of the object invoking the method, it determines if the method has already been resolved [1]. If the method has not been resolved, the runtime system searches for the method. If the method is found, the address of where the method is loaded is returned and execution continues from the

Name	Function
anewarray	create array of reference
arraylength	get length of an array
athrow	throw exception
checkcast	check if object is of given type
getfield	get field from an object
getstatic	get static field from a class
goto_w	branch always wide *(traps)*
instanceof	determine if object is of given type
invokeinterface	invoke interface method
invokespecial	invoke instance method (superclass, etc.)
invokestatic	invoke class (static) method
invokevirtual	invoke instance method
jsr_w	jump subroutine wide *(traps)*
lookupswitch	jump table match by key *(traps)*
monitorenter	enter monitor for object
monitorexit	exit monitor for object
multianewarray	create multdimensional array
new	create new object
newarray	create new array
putfield	set field in object
putstatic	set static field in class
tableswitch	jump table match by index *(traps)*
wide	extend local variable index *(traps)*

Table 3.35: DELFT-JAVA JVM specific Instructions

new address. If the method has been resolved, the name contained within the constant pool can be associated with a physical location in memory for each object.

LTB Supported Operations: Some architectural support is provided for LTB Operation. In particular, the ability to lock/unlock the cache, or to flush it is provided. Table 3.36 summarizes the available DELFT-JAVA Link Translation Buffer instructions.

Name	Function
ltbLock	lock an LTB entry
ltbUnlock	unlock an LTB entry
ltbFlush	flush the LTB

Table 3.36: DELFT-JAVA LTB specific Instructions

3.6 Conclusion

In this chapter we have described the DELFT-JAVA architecture in general terms. We presented the Memory (Storage) spaces the processor operates from. We also presented the complete set of operations the processor is able to perform and identified a number of JAVA Virtual Machine specific instructions for which special support is provided. Without the special instruction support, many cycles may be required to emulate the operations. A key point of our architecture was introduced - where it is easy to dynamically translate JAVA Virtual Machine bytecode into DELFT-JAVA instructions, no instruction set support is provided. For those JAVA Virtual Machine bytecode which are highly complex, instruction set support is provided to allow acceleration of the function through microarchitectural support.

In the following chapters we will show how the DELFT-JAVA instruction set architecture can be used to accelerate JAVA program execution. The next chapter presents a microarchitecture for high-performance JAVA execution. By using microarchitectural techniques and specific organizations which accelerate various aspects of the JAVA Virtual Machine, performance improvements can be realized using hardware to perform the acceleration.

If one woman can have a baby in 9 months it doesn't imply that 9 women can have a baby in 1 month. – Fred Brooks.

Chapter 4

Microarchitecture and Java Acceleration

In the previous chapters we gave an introduction to the JAVA Virtual Machine, discussed previous research on JAVA acceleration, and provided an architectural introduction to the DELFT-JAVA processor. This chapter is dedicated to describing the organization of our processor. We provide microarchitectural support for dynamic translation, dynamic linking, multiple thread units, multiple instruction issue, dependency collapsing, and other features common to modern superscalar processors. These techniques take advantage of key JAVA language properties to transparently extract parallelism without programmer intervention. The presentation is as follows: First we describe our hardware support for JAVA Virtual Machine dynamic translation. We describe how indirect access to the register file provides the basic mechanism required to dynamically translate JAVA Virtual Machine instructions. Then we provide an example of the translation process. Next we describe special hardware features we incorporated to assist in translation. Finally, we list instructions which are not translated. Second, we describe how we support dynamic method invocation. We provide background on dynamic method invocation. Next we describe the Link Translation Buffer (LTB) and its operation including enhancements which can be made to the LTB. Finally, we describe our concurrent multithreaded organization and describe how multiple thread units and multiple instruction issue efficiently accelerate JAVA program execution. We also briefly describe how the indirect registers and Link Translation Buffer operates within the description of the microarchitecture. We then describe some features of the architecture that are not primarily for JAVA execution but allow speedup of native methods. Finally we describe so related work on multithreaded architectures and present some conclusions.

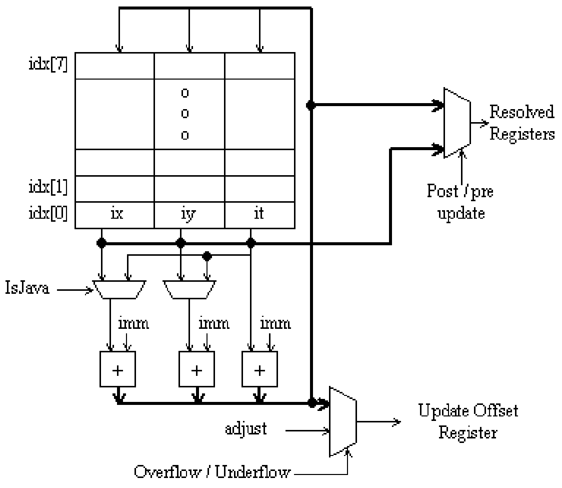


Figure 4.1: Indirect Register Access..

4.1 Dynamic Translation

The architecture supports the same basic datatypes as the JAVA Virtual Machine. We dynamically translate JAVA Virtual Machine instructions into DELFT-JAVA instructions by providing indirect access into the register file. Figure 4.1 shows a set of index registers. Each index (e.g. ix, iy, and it) is 5-bits wide with separate entries for each source and destination operand. Every indirect operation accesses the index register file to obtain the last previously allocated register. An immediate field within the instruction format can be used to specify offsets from the original index value. In addition, a pre/post-increment field specifies whether the index uses a pre-incremented or post-incremented value to resolve the register reference. For most translated JAVA instructions this can be inferred from the operation. For general indirect instructions, which are useful in vector operations, it is beneficial to directly specify a pre or post increment. Once the operands are transformed from an indirect address to a direct register reference, they are placed in the instruction window for dispatch. If an overflow or underflow of the register file is detected by the hardware, the offset register which maps the register file into main memory must be adjusted.

In addition, the register file may be configured to act as a memory cache. In this case, a base register indicates the starting memory address being cached. Valid and modified bits control the write-back to memory when overflow or underflow is detected.

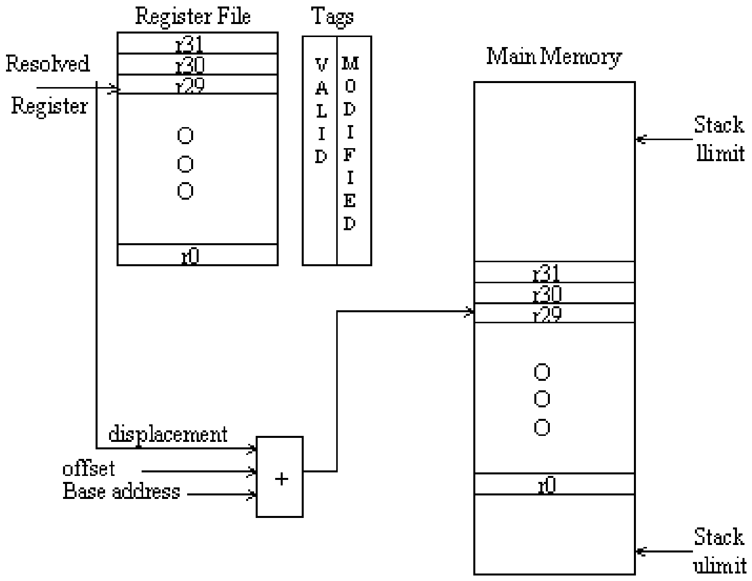


Figure 4.2: Indirect Register Mapping.

To illustrate how these operations are performed, consider the code show in Program 4.1.

```
add            r2, r0, r1
addi [idx7]    ++it, 2-ix, iy
storei [idx7]  base0 + #3, it++
```

Program 4.1: Indirect Instructions.

In Program 4.1, a typical RISC-style instruction is shown in line 1. The add mnemonic specifies the operation, r2 is the destination (target) register. Registers r0 and r1 are the source operands. When no type is explicitly specified, a $w32$ (signed integer 32-bits) type is implied. In line 2, addi specifies that an *indirect* add will occur. The idx[7] implies that the 8-th index register is to be selected. The source operand 2+ix implies that an immediate value of 2 (which is specified in the instruction format) is pre-updated with the contents of idx[ix][7] to determine the source operand. In line 3, a memory store operation is performed. The target operand is a memory location addressed by base register base0 with an immediate displacement of 3. To calculate the source operand, the value contained in idx[it][7] is used. In practice, the only way for this to happen is to be in JAVA translation mode (which provides for locked indexing using it). Since it+1 contains the +1 on the right hand side of the expression, it implies that idx[it][7] is post-incremented by 1. For JAVA Virtual Machine bytecodes, the pre/post increment values can be implied from the JAVA Virtual Machine instruction.

Figure 4.2 shows the indirect mapping translation. The resolved register address from Figure 4.1 is used as an index into the register file. This address is also used as a displacement which maps the register file into Main Memory. A 32-bit base address is set by the DELFT-JAVA processor to point to the starting memory location. A 32-bit offset is added to provide the current mapping of the register file to the stack main memory. If the amount of required stack storage exceeds the register file limit, a signal is sent to the DELFT-JAVA processor and the offset is adjusted as needed. The tags control whether all the data is written back on an overflow or underflow. It is possible to be continually updating main memory in the background while bytecode execution proceeds.

4.2 Example Translation

In this section we present the translation of a Vector Multiply. Program 4.2 shows a rudimentary JAVA program that reads an element of a vector from array a[], multiplies it with a fully disambiguated array b[], and stores the result in another independent array c[]. The JAVA language specifies that array memory is allocated on the heap. The operations take place on an element by element basis.

■ **Inner Loop Bytecode:** When compiled with -O optimization using Sun's Java JDK 1.1, the bytecodes produced for the inner loop of Program 4.2 (e.g. c[i]=a[i]*b[i}) are shown in Program 4.3. To be able to load a single element from an array, the address of the array is pushed onto the stack (Program 4.3

```
class VectorMultiply {
  public static final int MAXVEC = 100;
  public static void main( String[] args ) {
     int[] a,b,c;
    a = new int[MAXVEC];
    b = new int[MAXVEC]
    c = new int[MAXVEC];
    for( int i=0; i<MAXVEC; i++ ) { // init arrays
       a[i] = i; b[i] = 2*i; c[i] = 0;
    }
     for( int i=0; i<MAXVEC; i++ ) {
        c[i] = a[i] * b[i};
} } }
```

Program 4.2: Vector Multiply Example.

line 1) followed by the index to load (Program 4.3 line 2). Previously (not shown in Program 4.3), each array was allocated on the heap. As a result of executing the instruction *"newarray int"*, the heap address is returned on the stack. This address was immediately stored into a Local Variables location (e.g. LV[1], LV[2], and LV[3] for a[], b[], and c[] respectively).

```
1    aload_3    ; address of c[0] on heap
2    iload 5    ; index into c[]
3    aload_1    ; address of a[0]
4    iload 5    ; index into a[]
5    iaload     ; load element from a[index]
6    aload_2    ; address of b[0]
7    iload 5    ; index into b[]
8    iaload     ; load element from b[index]
9    imul       ; multiply a[i]*b[i]
10   iastore    ; store it into c[index]
```

Program 4.3: Compiled Inner Loop Bytecode.

■ **Translated Bytecode:** Program 4.4 shows the vector multiply inner loop bytecode translated into DELFT-JAVA indirect instructions. Because instructions are being translated from JAVA, all operand indirect references are with respect to the target location. When a program is about to begin execution

Opc		**Indirect Register**
load	[idx7]	–it, base_LV + #3
load	[idx7]	–it, base_LV + #5
load	[idx7]	–it, base_LV + #1
load	[idx7]	–it, base_LV + #5
load	[idx7]	++it, ++it + it
load	[idx7]	–it, base_LV + #2
load	[idx7]	–it, base_LV + #5
load	[idx7]	++it, ++it + it
mpy	[idx7]	++it, it, ++it
store	[idx7]	2+it + 1+it, it

Program 4.4: Translation Bytecode.

of JAVA bytecodes, a *"branchJVM"* instruction is executed by a DELFT-JAVA processor. As shown in Figure 4.1, this configures the IsJava control switch to use the *"it"* reference. The *"base_LV"* name is a symbolic name for one of the DELFT-JAVA base registers. As shown in Program 4.4 line 1, loading a JAVA array reference from a local variable is translated as an indirect load with base register plus displacement. Notice that after the translation most of the type information contained within the JAVA instruction is removed. It is therefore important for a separate program to verify the bytecodes prior to execution if security is an issue.

	Opc	**Direct Register**
		// initial value of idx[7][it] = 24
i_1	load	r23 $\Leftarrow$ Mem[base_LV + #3]
i_2	load	r22 $\Leftarrow$ Mem[base_LV + #5]
i_3	load	r21 $\Leftarrow$ Mem[base_LV + #1]
i_4	load	r20 $\Leftarrow$ Mem[base_LV + #5]
i_5	load	r21 $\Leftarrow$ Mem[r21 + r20]
i_6	load	r20 $\Leftarrow$ Mem[base_LV + #2]
i_7	load	r19 $\Leftarrow$ Mem[base_LV + #5]
i_8	load	r20 $\Leftarrow$ Mem[r20 + r19]
i_9	mpy	r21 $\Leftarrow$ r20 * r21
i_{10}	store	Mem[r23 + r22] $\Leftarrow$ r21

Program 4.5: Final DELFT-JAVAInstructions.

■ **Executed Bytecode:** Program 4.5 shows the operation code mnemonic and the final resolved instruction. For this example, we assume that the value contained in *idx[7][it]* is 24. Of notable observation is the large number of Memory accesses required. However, it should be noted that most of these are not global memory accesses but rather Local Variable accesses which may be cached locally or even stored in small buffer. The JAVA language currently allows up to 2^{16} local variables. Implementations which do not store this much memory locally (e.g. when the Local Variables are allocated to registers) must dynamically allocate spill memory to accommodate a particular program's requirements.

4.3 Hardware Support

In order to perform JAVA translation, the DELFT-JAVA machine has a number of special registers which control the dynamic translator. When the processor transitions to JAVA-mode using a *branchJVM* instruction, the programmer views the processor as a JAVA Virtual Machine and translation is automatically enabled. In any of the privileged modes, the translator is disabled. When dynamic translation is enabled, the register file caches the top of the JAVA stack. This is accomplished by using architected base and offset/displacement registers within the architecture. During normal JAVA execution, the register file can cache up to 32 stack entries. In addition, the actual top of the stack may be offset from the memory location that points to it to allow for delayed write-back. The JAVA language specifies that in the absence of explicit synchronization, a JAVA implementation is free to update the main memory in any order[1]. Therefore, each context may maintain a set of register file status bits that allow a more balanced utilization of bandwidth constrained resources.

To ensure proper sequencing of instructions during JAVA translation, all instructions are assumed to be stored as JVM bytecode. To transition to kernel-mode, a special reserved JAVA Virtual Machine instruction is used. The JAVA Virtual Machine specification states that 3 opcodes will permanently be reserved for implementation dependent purposes[3]. The DELFT-JAVA processor utilizes one of these instructions to transition a context between JAVA Virtual Machine execution and general DELFT-JAVA execution. When the context is executing in kernel-mode, instructions are assumed to be stored as 32-bit DELFT-JAVA instructions. This allows the branch decode logic to operate correctly without modifying JAVA compilers while compilers specific to our architecture can take advantage of hardware-specific features. Additionally, it is not necessary for all DELFT-JAVA instructions to execute in kernel mode. A security scheme may be implemented using a supervisor invoked transition to native user-mode DELFT-JAVA execution.

anewarray	invokeinterface[1]	multianewarray
arraylength	invokespecial	new
athrow	invokestatic	newarray
checkcast	invokevirtual	putfield
getfield	jsr_w[1]	putstatic
getstatic	lookupswitch[1]	tableswitch
goto_w[1]	monitorenter	wide
instanceof	monitorexit	[1](traps)

Table 4.1: Instructions with Special Support.

■ **Non-translated Instructions:**Primarily, we dynamically translate arithmetic and data movement instructions. In addition to the translation process, the architecture provides direct support for a) synchronization, b) array management, c) object management, d) method invocation, e) exception handling, and f) complex branching operations. The JAVA instructions shown in Table 4.1 have special support in our architecture. These instructions are dynamically translated but only the parameters which are passed on the stack are actually translated. The high-level JAVA Virtual Machine operations are translated to equivalent high-level operations in the DELFT-JAVA architecture. In addition, four instructions which are greater than the 32-bit DELFT-JAVA instruction format width trap.

■ **Conclusion:** In this section we have described the dynamic translation of JAVA Virtual Machine instructions into DELFT-JAVA instructions. The basic mechanism for accomplishing the translation is indirect access to the register file. Special hardware is utilized to accomplish the translation. In addition, certain JAVA Virtual Machine reserved opcodes are used to transition between kernel and JAVA execution. All but a handful of instructions are dynamically translated. Those that are not have special instruction set support provided in the native DELFT-JAVA ISA. A very few number of instructions have architectural support but trap due to their variable length.

In the next section we will describe an organization of a DELFT-JAVA processor and show how JAVA execution can be further accelerated at the microarchitecture level. We will introduce the Link Translation Buffer for accelerating dynamic method invocation which is required in the JAVA language. Then, we will show how multiple thread units, multiple instruction issue, and dependency collapsing can further accelerate JAVA execution.

4.4 Link Translation Buffer

An important consideration in accelerating JAVA 's dynamic linking is the Link Translation Buffer (LTB)[65]. The LTB acts as a global repository for dynamically resolved names. During dynamic linking, the name of the class or field to be resolved is contained in the constant pool. After a process called resolution [1], the name contained within the constant pool can be associated with a physical location in memory. This association is placed in the Link Translation Buffer. If the control unit finds the constant pool address in the LTB and the requesting class has access permissions to the data, then the control unit very quickly returns the resolved address. There is still a potential problem that the LTB may hold data that is stale. To diminish the impact of this, the control processor regularly re-resolves addresses when it is not busy performing other tasks. A program may also completely disable the LTB or more judiciously issue flushLTB instructions.

■ **Background:** Dynamic method invocation is a technique whereby a program may invoke a method with the same name and parameters but execute a different sequence of code depending upon the run-time type of the object invoking the procedure. The JAVA programming language supports generalized use of dynamic method invocation. The C++ language also supports a more limited form of this behavior through the virtual keyword. As in C++, JAVA method invocation generally involves an indirection through a method dispatch table.

In Program 4.6 and Program 4.7, both the C++ and JAVA statements for msc.instanceMethod() invoke MySubClass:: instanceMethod(). In the C++ version, the virtual keyword informs the compiler that the method instanceMethod() will have runtime binding behavior. The capability of calling different methods at runtime is known as *late binding* because the method to invoke is not known until the program executes. In C or Pascal, by contrast, a function call always resolves to a specific, compile-time known, location. This is known as early binding because physical addresses can be associated with the

```
class MyClass {
 public:
   virtual void instanceMethod() {};
};
class MySubClass : public MyClass {
 public:
   virtual void instanceMethod() {};
};
void main() {
 MyClass mc = MyClass();
 MyClass msc = MySubClass();
 mc.instanceMethod();
 msc.instanceMethod();
}
```

Program 4.6: C++ Method Invocation

```
public class MyClass {
 void instanceMethod() {}
}
public class MySubClass extends MyClass {
 void instanceMethod() {}
}
class Test {
 public static void main(String args[]) {
 MyClass mc = new MyClass();
 MyClass msc = new MySubClass();
 mc.instanceMethod();
 msc.instanceMethod();
}}
```

Program 4.7: JAVA Method Invocation

function during compilation and linking. The advantage of early binding is that the only run-time overhead is argument passing, performing the function call, and cleaning up the frame stack. The advantage of late-binding is that the mix of objects in a system is not required to be fixed at compile-time. The cost of this additional flexibility is the run-time efficiency of deducing which methods to invoke. C++ is a hybrid language and only uses late binding when the virtual keyword is utilized but still requires the set of all potential objects which may be invoked to be known at compile time. JAVA , because of its dynamic linking facility, is inherently a late-binding language that allows an arbitrary set of objects, which may not all be known at compile time, to be invoked at run-time.

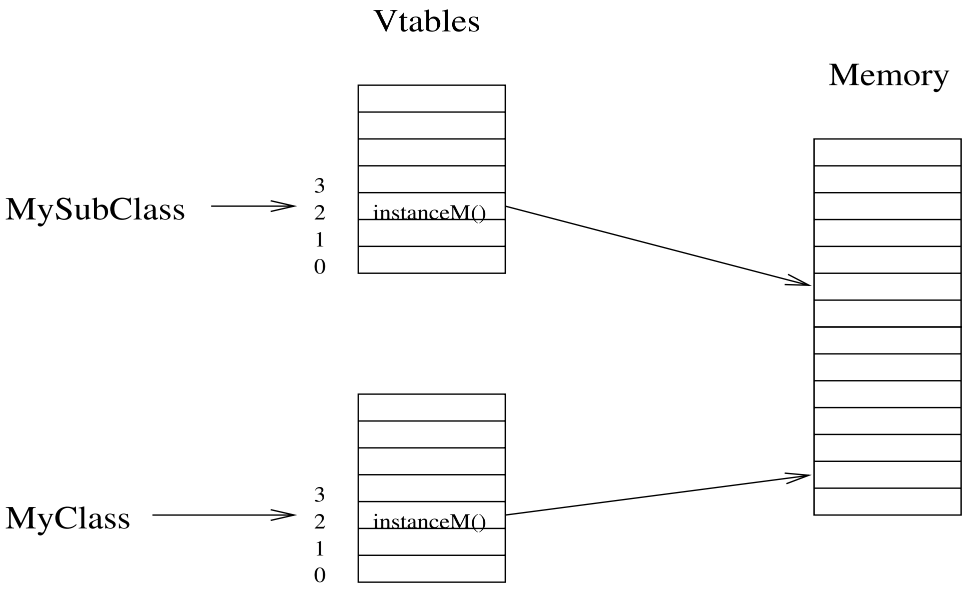


Figure 4.3: C++ Virtual Table Implementation.

In Figure 4.4 we show a possible implementation of C++ virtual tables. During compilation, two tables can be created - one for MyClass and one for MySubClass. These are stored in memory. When an object invokes the function instanceMethod(), they both have the same offset in their virtual tables. The contents of that memory offset is used to reference the actual location of the code in memory.

```
                   Method void main(java.lang.String [])
Line   InstrAddr   Instr
   1           0   new #3 <Class MyClass>
   2           3   dup
   3           4   invokenonvirtual #7 <Method MyClass.<init>()V>
   4           7   store_1
   5           8   new #4 <Class MySubClass>
   6          11   dup
   7          12   invokenonvirtual #6 <Method MySubClass.<init>()V>
   8          15   astore_2
   9          16   aload_1
  10          17   invokevirtual #5 <Method MyClass.instanceMethod()V>
  11          20   aload_2
  12          21   invokevirtual #5 <Method MyClass.instanceMethod()V>
  13          24   return
```

Program 4.8: JAVA Method Invocation Bytecode

To understand how dynamic linking and late-binding are performed in JAVA, it is instructive to see the compiled bytecodes. As shown in Program 4.8, the first instruction (new #3) creates a MyClass object on the heap. Line 3 invokes its constructor. Line 5 does similarly for MySubClass. The interesting cases are found at lines 10 and 12. The invokevirtual call for both the MyClass::instanceMethod() and MySubClass::instanceMethod() are called with the same Constant Pool index (e.g. #5). The only way to distinguish them is through the object reference that is loaded in Lines 9 and 11. In line 9, the method dispatch table to use is the one for a MyClass object. In line 11, it is for the MySubClass object. As in C++, the offsets into the Constant Pool are the same. The actual method to call is disambiguated by the object reference which is loaded onto the stack at runtime. This is exactly analogous to Figure 4.4, except that the method dispatch tables are built at run-time by the runtime system.

4.5 LTB Acceleration

The JAVA Virtual Machine [3] contains support for run-time bound method invocation. Because the DELFT-JAVA processor incorporates invocation instructions directly into it's ISA, architecturally transparent techniques can be used to accelerate dynamic linking and method invocation. In this section we briefly describe method resolution and invocation. We then introduce an architecturally transparent technique, the Link Translation Buffer, and explain its operation.

■ **Method Invocation:** In a JAVA program, the Constant Pool contains the names of the methods to be invoked. These are stored as strings and function much like a symbol table. When a method is invoked, the JAVA Virtual Machine searches the Constant Pool for the name of the method to invoke. Then, based on the run-time type of the object invoking the method, it determines if the method has already been resolved [1]. If the method has not been resolved, the runtime system searches for the method. If the method is found, the address of where the method is loaded is returned and execution continues from the new address. If the method has been resolved, the name contained within the constant pool can be associated with a physical location in memory for each object.

■ **LTB Operation:** A Link Translation Buffer is a buffer which accelerates late-binding of names with locations in memory. It is properly characterized as an organizational technique although some architectural support can be provided. In particular, a kernel program may need to enable, disable, lock, or judiciously flush the buffer.

The Link Translation Buffer provides an architecturally transparent means to accelerate the JVM's late-bound method invocations. When a DELFT-JAVA processor executes a method invocation instruction, it first dynamically translates the instruction into an equivalent DELFT-JAVA invocation instruction. When the instruction is executed, it checks the Link Translation Buffer to determine if the current object reference and its associated constant pool address are in the Link Translation Buffer. If the information is resident in the LTB, a new frame is created and the method is directly invoked. If the information is not in the LTB, the instruction is forwarded to the Control Unit. The Control Unit may be implemented as a state-machine, microcode, or be a separate processor executing a thin interpretive JVM layer. The Control Unit is responsible for resolving the method name and placing the physical method invocation address into the Link Translation Buffer.

■ **LTB Design:** In designing the Link Translation Buffer, there are two cases to consider: 1) dynamic method invocation and 2) static method invocation. In dynamic method invocation, a JAVA Virtual Machine invoke instruction takes a 16-bit Constant Pool value from the instruction format and searches the current object's Constant Pool for the name of the method to invoke. The current object's this pointer and the Constant Pool index from the instruction field provide a unique identifier to the name of the method to invoke. The name of the method includes the class name, the method name, and the method signature (e.g. the argument and return types). The DELFT-JAVA processor must then determine if the class is already loaded. If it is not, the Control Unit searches for the class and loads it. The instance reference is then retrieved from the stack and the list of methods defined by that class (and possibly it's superclasses) is searched. If a method is found that matches the name and descriptor, it is invoked. Once this relationship is resolved to a physical address, additional invocation instructions may use the previously resolved address directly. Thus, the caller's object id (this pointer and Constant Pool location) and the callee's object reference provide sufficient information to directly invoke the method. This relationship is stored in the LTB. In static method invocation, the method to be invoked is a class method (e.g. not an instance method) and does not need an explicit object reference. In this case the caller's object id is sufficient to invoke the method.

The Link Translation Buffer can support a variety of entries and associativities depending upon the desired implementation cost. We note that since the runtime can define the object ids, depending upon the actual associativity of the LTB, we may optimize the runtime to produce object ids which minimize cache conflicts. This is not true, however, for the 16-bit Constant Pool offset location.

■ **LTB Enhancements:** In the design of the Link Translation Buffer some further enhancements can be made that reduce the impact of creating new frames. Our initial design places a small amount of additional data into the LTB. In the DELFT-JAVA processor, a frame is created which holds the 32-bit base values of the Constant Pool and Local Variables. Because each invoke instruction causes the creation of a new frame, we would like the frame creation to have as minimal overhead as possible. To assist this, we designed all execution frames within a thread to be contiguous. The Heap, Local Variables, and Constant Pools do not have this requirement. Making the frames contiguous allows us to use the operand stack as the frame stack. Because there can be 2^{16} Local Variables in a JAVA frame, we do not place this data in the execution frame. By not placing them in the frame stack, we avoid excessive flushing of the register file stack cache. In JAVA , only the actual operands are considered to be placed on the stack. In our method, the calling parameters also traverse through the stack so that the values are cached by the register file. Because the register file operates as a type of cache, if we were to change the operand stack's offset register, the modified register file locations would be required to be flushed on every method invocation. This requires no more additional cycles than a caller-save calling convention however we can gain the advantages of using a register file with no register saving overhead and variable length parameter passing. By using contiguous frames, the only information that needs to be stored when a new frame is created is the base addresses of the Local Variables and Constant Pool and the instruction address to return to the previous frame. These additional pushes and pops are performed transparently to the JAVA programmer and are logically considered to be part of the Operating System.

Caller's Reference	CPool Entry	Callee's Object Ref	LV[0]	CP[0]	Other
32-bit	16-bit	32-bit	32-bit	32-bit	

Figure 4.4: Link Translation Buffer.

As shown in Figure 4.4, some other performance enhancements can also be made by associating additional data with the LTB. Examples include synchronization locks[1], garbage collection reference counts, actual data, and other information to accelerate dynamic invocation.

The JAVA Virtual Machine also defines other operations which require late-binding. The getstatic, putstatic, getfield, and putfield instructions may work in an analogous manner except that the value stored in the LTB is the data field itself rather than the address of the method to invoke. In addition, these instructions do not cause the creation of a new frame.

■ **Conclusion:** In this section we have described the Link Translation Buffer which is an organizational technique to accelerate JAVA'sdynamic method invocation. We have shown why it is effective, how it is designed, and how it may be enhanced. In the next chapter we will discuss organization details which allow us to accelerate JAVA execution. Dynamic instruction translation and the LTB are key aspects of accelerating JAVA execution. Other techniques from superscalar processors may also be employed. These are discussed next.

[1]In fact, this synchronization field is required since a synchronized method does not need to be defined through monitorenter and monitorexit instructions. Because Java allows the class file to contain data stating the entire method is synchronized, some mechanism must be present in the invocation scheme to cache this information.

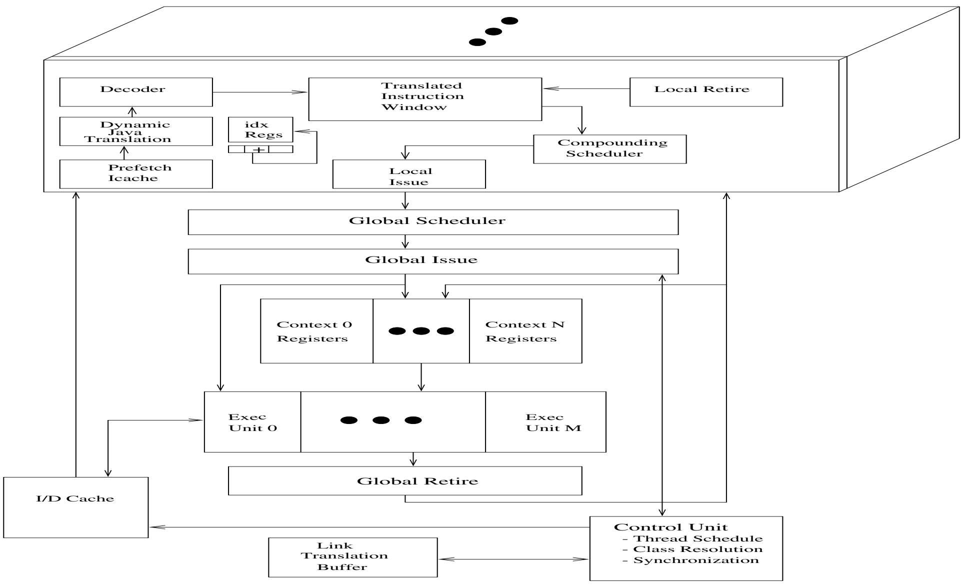


Figure 4.5: Concurrent Multithreaded Organization.

4.6 Concurrent Multithreaded Organization

In this section, we present a concurrent multithreaded organization of the DELFT-JAVA architecture. This organization provides hardware support for multiple context instruction issue and global instruction scheduling. The organization supports multiple concurrent execution of threads which share global execution units. We define a *context* as a hardware supported thread unit. Each context assumes that the processor's organization incorporates (logically) an instruction cache, a decode unit, a local instruction scheduler, a local instruction issue unit, and an instruction retire unit. A context does not include any shared resources such as a first level (L1) cache, execution units, a register file, global instruction schedulers, nor global issue units. The term *thread* is generally used to refer to the programmer's view of a thread - a possibly concurrent stream of independent executing instructions[66, 67]. In this thesis, the term context denotes the hardware on which a thread may run. The system software may map any number of threads to a particular context.

■ **Operation:** All instructions are fetched from global shared memory and placed into a global L1 on-chip instruction cache. Each context also assumes a (logical[2]) zero level (L0) instruction cache to provide concurrent per context *instruction fetch* capacity. During normal user-level operation, all instructions are fetched as JAVA instructions. After being fetched, most JAVA instructions are *dynamically translated* into the DELFT-JAVA instruction set. Because the instructions are stored in cache memory as JAVA instructions, branching and method invocation code produced by JAVA compilers will execute properly on the DELFT-JAVA architecture. After translation, the instructions are decoded and placed in a *local instruction window*. The instruction window keeps track of issued and pending instructions. The *local instruction scheduler* is responsible for determining how instructions within the window should be scheduled. This unit takes the instructions in a RISC form and performs instruction combining and compounding. Often, in stack based architectures, a number of optimizations pertaining to stack manipulation can be efficiently combined[48, 68]. The DELFT-JAVA processor may also dynamically build internal compound instructions[69]. Instructions are then sent to the *local issue unit* after they have been scheduled. The local issue unit determines if the instructions that have been locally scheduled can be issued to the global instruction scheduler. To resolve interlock dependencies, an interlock collapsing unit could be used[70].

All instructions which require access to shared resources must be forwarded to the *global instruction scheduler*. This unit schedules the aggregated instructions destined for execution units. Any number of implementation dependent scheduling policies can be utilized including priority-based, round-robin, earliest deadline, etc. The JAVA language specifies that in the absence of explicit synchronization, a JAVA implementation is free to update the main memory in any order[1]. This relaxed memory consistency model allows the scheduler to reorder the instructions from individual contexts to optimize the utilization of the shared execution units. After all instructions which request global shared resources have been scheduled, they are sent to the *global issue unit*. This unit ensures that global resources are available to begin issuing instructions. Instructions may be issued in one of two forms: single independent instructions and compound parcels. A parcel is a dynamically built compound instruction. Parcels are particularly effective in reducing the logic complexity of implementations and execute in less cycles when used in conjunction with interlock collapsing units. In a traditional processor implementation, all execution units would require bypass circuitry between each other. As the number of global execution units becomes large, it is no longer feasible to provide general bypassing between all sets of execution units. In the DELFT-JAVA processor, this

[2]logical meaning not necessarily physical.

requirement is removed by provided compound instructions which collapse interlocks and then scheduling the interlocked instructions within a parcel. The global issue unit has the capability of reordering the execution of individual instructions and parcels. If the global issue unit can find available resources, it can splice an independent instruction from an alternative context into a parcel. Since contexts are independent, this ensures that an instruction spliced into a parcel does not cause invalid results. Additionally, because each instruction contains a unique context identification, the results are forwarded to the proper context.

After global execution, all results are forwarded to the *global retire unit*. This unit removes the requirement for a general interconnection unit between all contexts and execution units. If instructions were not executed speculatively, the global retire unit writes the results to the register file after forwarding the instruction to the *local retire unit*. Otherwise, the result is maintained in the retire unit until the conditional outcome is known.

All instructions eventually return to the local retire unit in the context from which they originated. This unit is responsible for committing state to the context. Each context may retire multiple instructions per cycle.

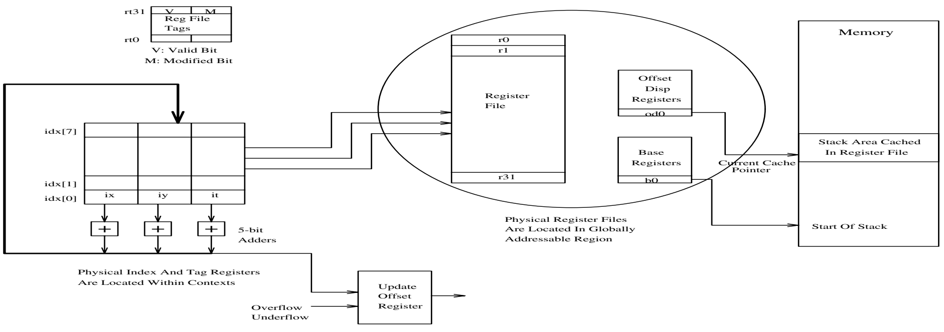


Figure 4.6: Concurrent Multithreaded Registers.

■ **Registers:** From the perspective of a context, the register file consists of a standard 32 entry by 32-bit register array. From the perspective of the machine, this resource is managed as a global register file that is addressed by a context identifier that is appended to the instruction's register reference. An alternative organization would be to place the register files logically within a context. This organization however creates a proliferation of register file ports. Managing the register file as a global resource reduces the number of ports to the peak retire rate of the machine versus the peak retire rate of a context.

Instructions have two methods of accessing the register file: 1) direct RISC-style references and 2) indirect index access. Even though there is an indirect reference, all instructions physically execute using direct RISC-style register references. The indirect index registers are only used to translate instructions. This implies that they are not part of the register file and do not affect the execution path.

The JAVA Virtual Machine instruction set architecture is inherently stack based [3]. When executing JAVA instructions, the register file index registers create a circular buffer that is mapped to the operand stack in memory. A set of valid and modified bits are associated with each register. These bits are maintained logically within the local context. These registers automatically prefetch and spill as the stack size changes.

■ **Translation:** As described in the previous chapter, the indirect access to the register file plays the largest role in the translation of JAVA bytecodes. As shown in Figure 4.6, when executing JAVA instructions, the register file index registers create a circular buffer that is mapped to the stack in memory. A set of valid and modified bits are associated with each register. These bits are maintained logically within the local context. A JAVA instruction such as iadd goes through two intermediate phases. The first phase translates the instruction into a valid DELFT-JAVA instruction. In this case, a add.ind.w32 idx[0] it, it-1, it-1 is generated by the translation logic. If we assume that the top of the cached stack in idx[0] is currently in r5, this instruction proceeds through the decoder and is placed in the decoded instruction window as add.w32 r5, r6, r6. Functionally, this performs r5 + r6 $\rightarrow$ r6. In the DELFT-JAVA processor, the stack grows upward in both memory and in register file references. These registers automatically prefetch and spill as the stack size changes.

■ **Execution and Context Switching:** When a thread begins execution within a context, the offset registers are written with the location of the frame, operand stack, and local variables memory locations. Additionally, the register file tags within the context are reset. When the operand stack address is written to the offset register, the context begins to generate speculative load instructions. This allows the register file to pre-fill only if there is adequate bandwidth available to the L1 cache. It also reduces cache thrashing because the L1 cache is not obligated to evict data upon a speculative load.

As instructions begin to execute, if the speculative pre-loads were successful, context execution proceeds without delay. If the pre-loads were not successful and the data is required for execution, the local context re-issues the load non-speculatively. This effectively raises the priority of the load instruction. When the data arrives at the context, a valid bit associated with that register file location is set. If the register is modified at any point during program execution, the modified bit is set. If the processor has spare resources, a speculative cache store instruction is generated. If there is spare bandwidth available, the processor stores the updated memory location and resets the modified bit. Otherwise, execution continues with a delayed write-back.

In some cases, the global thread management unit may determine that a particular software thread has resulted in a unacceptable degradation of a hardware context. In this case, the unit may make a request to the context to perform a context switch so that a new thread may be mapped to the context. Since results are only committed by the retire unit, it is possible to interrupt a context at any time. When a context becomes invalid, it signals the global instruction scheduler and issue unit to flush any remaining instructions in the queue. It then checks the modified bits of the register file to determine if any values must be written back into memory. After all state has been saved in memory, the context may signal the global thread management unit that a new thread may be mapped to the context. Even though the context is now freed to map a new thread onto it, it may still be the case that an instruction was executing at the time of the context switch request. It is the responsibility of the global retire unit to ensure that any instructions received from execution units destined for the switched context are not forwarded to the local retire unit. This is not difficult to implement when the longest instruction execution time is less than the context switch time.

■ **Control Unit:** The *control unit* is responsible for managing system resources, ensuring synchronization, cache locking, dynamically linking classes, performing I/O operations, running operating systems, loading instructions, and generally performing system functions. Since the JAVA Virtual Machine does not provide all the functionality generally required by a full operating system, many of these functions have been grouped into a special control unit. A control unit is analogous to a context except that it contains additional resources that are not necessarily required within a context. These resources could be implemented within a context but with a large number of contexts it would lead to unacceptable duplication of typically idle hardware. There are no architectural limits on the number of control units permitted in a system. The control unit is a logical independent entity so that the complexity of bussing between global system resources such as caches is significantly reduced.

Some of the differences that distinguish the control unit from a context are:

First, a control unit has direct access to the *Link Translation Buffer*. The LTB acts as a global repository for dynamically resolved names. During dynamic linking, the name of the class or field to be resolved is contained in the constant pool. After a process called resolution, the name contained within the constant pool can be associated with a physical location in memory. This association is placed in the Link Translation Buffer. If the control unit finds the constant pool address in the LTB and the requesting class has access permissions to the data, then the control unit very quickly returns the resolved address. There is still a potential problem that the LTB may hold data that is stale. To diminish the impact of this, the control processor regularly re-resolves addresses when it is not busy performing other tasks. A program may also completely disable the LTB or more judiciously issue `flushLTB` instructions.

Second, the global instruction scheduler has direct access to the control unit and may schedule instructions on execution units that are inherently owned by the control unit. This is to ensure that all addresses are resolved through the control unit and that all synchronization is performed by the control unit. When execution has completed, instructions are returned to the global retire unit which then returns the results to the context requesting the operation. Care is taken by the Global Retire Unit to ensure that any locks acquired for a context that have undergone a context switch are released.

Third, any unimplemented instructions trap first through the global instruction scheduler and global issue unit to the control unit. The control unit then either halts execution if it is an illegal instruction or can emulate the instruction sequence and return the instruction to the global retire unit.

Fourth, the control unit is responsible for synchronization. This is because generally it may be possible for an object to have acquired a lock but the locked object may not be fully resident in the L1 instruction cache. The easiest way to deal with this issue is to lock down all cache lines associated with object synchronization. Another alternative is to have the control unit check each address as it is brought into the cache to ensure that the address is not contained within an already locked object. If it is the context that currently owns the lock that requested the instruction, the new instructions are brought into the cache. If it is any other context requesting the instruction, the context is placed in a blocked state. This reduces thrashing within the cache and allows the thread scheduler to make better decisions about the mapping of threads to contexts.

Fifth, a thread scheduler in the control unit is responsible for mapping all of the software threads in the system to particular hardware contexts. It may update the state of threads (i.e. from active to blocked), it may preempt threads, and it may create and destroy threads. There are no restrictions on the mappings of threads to contexts. Multiple threads may be mapped to a single context or to multiple contexts.

Finally, the control unit performs all the necessary functions required in physical processors that are not required in virtual machines. These include I/O access, initialization, and system administration functions.

4.6.1 Enhancing Performance

Accelerating the JAVA Virtual Machine interpreter is only one aspect of JAVA performance improvement implemented in the DELFT-JAVA processor. We utilize a number of techniques including pipelining, load/store architecture, register renaming, dynamic instruction scheduling with out-of-order issue, compound instruction aggregation, collapsing units [70], branch prediction, a link translation buffer [65], and standard register files. We selectively describe some of these mechanisms.

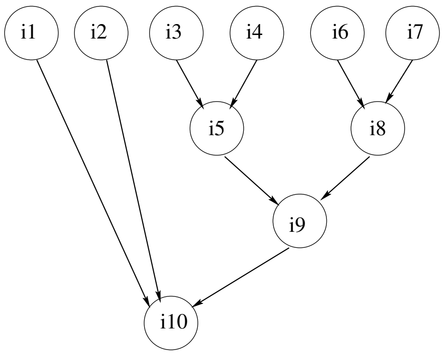


Figure 4.7: Vector Multiply Dependency Graph.

■ **Removing Hazards:** A common problem with stack architectures is that the stack may become a bottleneck for exploiting instruction level parallelism. The dependency graph for Program 4.5 is shown in Figure 4.7. Note that this dependency graph is shown prior to any register allocation scheme. Since the results of operations typically pass through the top of the stack, many interlocks are generated in the translated instruction stream [71]. Register renaming allows us to remove false dependencies in the instruction stream. In addition, an interlock collapsing unit can be used to directly execute interlock dependencies[69, 70, 72]. After translation the instructions are placed in an instruction window.

■ **Multiple Instruction Issue:** After translation the instructions are placed in an instruction window. Once instructions are translated into a RISC-based form, superscalar techniques are used to extract instruction level parallelism from the instruction stream. Reservation stations are an effective means of determining which instructions can execute concurrently[54] . Since all thread-units operate independently, multiple instructions can be issued from each thread unit as well as multiple thread units.

■ **Bounds Checking:** The JAVA language specifies that arrays must be bounds checked[1]. Special register sets can be provided for this purpose. The microarchitecture is not required to implement them but the architecture supports the use of bounds checking.

4.7 Garbage Collection

Microarchitectural support for the DELFT-JAVA processor has been considered in [73]. We are indebted to Alexandru Berlea for this investigation of garbage collection for the DELFT-JAVA processor.

An important feature of JAVA which distinguishes it from C++ is its support for garbage collection, i.e., the automatic reclamation of unused dynamically allocated memory. The garbage collection available in JAVA relieves the programmer from the burden of having to explicitly deallocate dynamic memory when it is no longer needed. All JAVA objects are created on a dynamically allocated memory zone (heap).

Generally speaking, the best possible garbage collector for a given application can be implemented when the allocation behavior of the application is known. This is an idealistic case since in general garbage collectors are not written for a specific application but for a group of applications. In this case the specific allocation behavior is generally not known before the applications are executed. A garbage collector tailored for each application is therefore not possible. However, best overall performance can be achieved by a garbage collector if a general pattern of dynamic memory allocation for the applications it will garbage collect exists and can be deduced.

One could chose to write the JAVA Virtual Machine in a language which provides garbage collection and not bother with garbage collection for the objects created by the JVM. By doing so there is no further need of an explicit garbage collection within the JVM. Objects that are not reachable for the JAVA Virtual Machine are also not reachable for the garbage collector in the source language. The initial simulator for the DELFT-JAVA processor took this approach. The concern with this approach is that the garbage collector for the JVM is at best as good as the collector of the implementing language. Another disadvantage is that the collector of the implementing language will have to deal not only with the objects created by the JVM (e.g. JAVA objects), but also with objects needed for the JVM itself in order to execute. This makes optimizations based on the observations which apply only to the JAVA Objects potentially problematic.

Another possible approach is to have the garbage collector written only for the JAVA objects. The other dynamically allocated objects needed for the JVM itself require explicit memory reclaimation. In the case that the source language of the JVM provides garbage collection, they may not require specific deallocation. Otherwise, they must be explicitly deallocated if the JVM is a program written in a non garbage-collected language. If the later solution is chosen then one could get again the same problem the garbage collection tries to solve - relieving the programmer from the burden of having to know when the space occupied by a dynamically allocated object is to be freed. Furthermore, a JVM is a complex program and is specifically the type of program for which garbage collection was intended. If memory losses accumulate as a consequence of the death objects allocated for the JVM itself and the JVM runs for a long period of time, the application could run out of memory even if the garbage collector in the JVM is very accurate. An example of this may be if the JVM is integrated into some other long running program. In fact, if the JAVA application and the JVM share the same dynamic memory zone, significant memory inefficiencies may occur.

A third approach is the effort-saving technique utilized in Kaffe. The idea is that since a garbage collector had to be written for the JAVA objects, why not use the same garbage collector for the dynamically allocated objects in the C program. The advantage of this approach is ease of the implementation. The C-language can be used as a garbage collected language since objects created for the internal use of the JVM could remain not explicitly deallocated. The disadvantage is that when both objects are combined, the garbage collector for the JAVA objects and the garbage collector for the objects allocated for the internal use of the JVM can only be as good as the worst case of the two collectors. C objects when combined with those used internally by the JVM can not be accurate collectors because the C-language does not support one of the conditions in order to implement accurate garbage collection. Specifically, one can not tell for sure if a variable contains a pointer or a non-pointer value. The only type of garbage collection possible in C is conservative garbage collection. That is why the Kaffe collector is a conservative collector even if accurate collection is possible for the JAVA objects. A key point of our approach is that we separated the collector for the C objects from JAVA Objects. The C objects were managed by the original Kaffe garbage collector whle we implemented an accurate garbage collector for the JAVA objects.

For the DELFT-JAVA engine we used a very simple garbage collector interface. We then tested it by simulating the objects in the heap and their garbage collection with a C++ program. The examples below are taken from the simulation program which used a very simple mark and sweep collector.

The garbage collector interface provides two basic operations:

1. dynamic memory allocation (the allocate() function) and
2. explicit garbage collection invocation (the collect() function).

```
class gcHeap{
  /* list of pointers to allocated heap objects */
  pointerList allocatedObjects;
  /* list of user pointers to live objects in the heap */
  pointerList liveObjects;
  void mark(POINTER p_object);
  void sweep();
  void myCollect(pointerList * p_rootSet);
 public:
  POINTER allocate(objectDescriptor * p_od, rootsObject * p_ro);
  void collect(rootsObject * p_ro);
};
```

Program 4.9: Garbage Collection Interface.

In fact, the garbage collection system has two parts: the dynamic memory allocator and the garbage collector. All dynamic allocation requests are directed to the memory allocator and must contain the description of the type of the object whose allocation is requested. An object descriptor contains the map (layout) of pointers within the object and is needed when scanning the object during garbage collection. Therefore, the object descriptor is made available by the memory allocator by storing in the header of each allocated object a pointer to its object descriptor. When garbage collector occurs, the graph of live objects is traversed starting with the roots and using the pointer maps located in the objects' headers. The allocation process is depicted in Figure 4.9 where the differences and the similarities to the Kaffe dynamic allocator depicted in Figure 4.8 are outlined.

An adapter from the DELFT-JAVA processor allocator interface to the Kaffe allocator interface would need to call from the DELFT-JAVA processor the allocate() function and the NewObject() function in the Kaffe interpreter in order to use the Kaffe allocator.

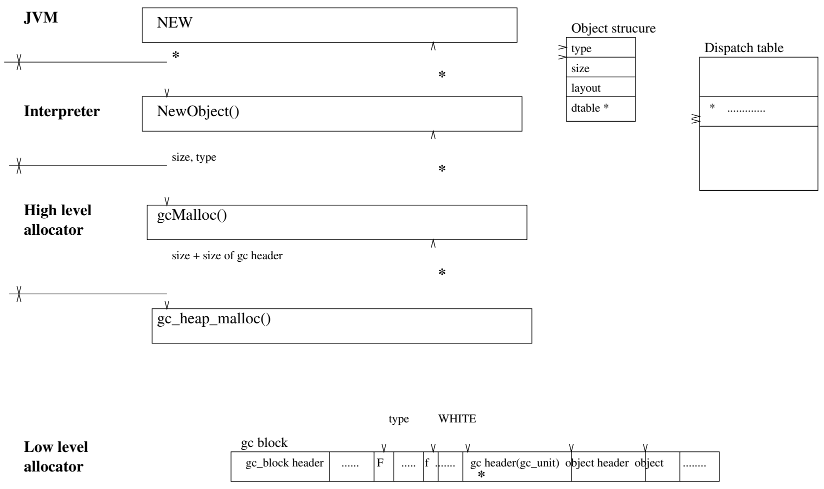


Figure 4.8: Object allocation in the Kaffe JVM

As a garbage collection can occur as the result of a memory allocation request, if enough free memory is not available to satisfy the request, the garbage collector roots must also be available at the time the request is made. We designed the garbage collector roots as a garbage collection friendly object that has a method to return the roots as a pointer list. This is shown in Program 4.10.

The garbage collector can be invoked directly or indirectly when a memory allocation request fails. In both cases the garbage collector roots are made

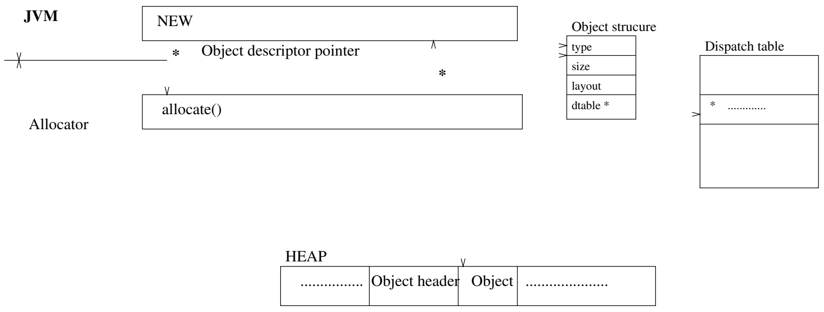


Figure 4.9: Heap occupancy in the *jess* benchmark, when interpreted by the CC Kaffe JVM with a heap of 5 Megabytes

```
class rootsObject {
 public:
   virtual pointerList * getPointers();
};
```

Program 4.10: Garbage Collection Roots.

available through the pointer to the garbage collector roots object. The roots object could be also made available to the garbage collector via a global variable. In this case the collector could be invoked without parameters just like the Kaffe collector. If a specific DELFT-JAVA processor implemented the modified garbage collection system of Kaffe, a minor modification would be needed in the ccWalkRootSet() function. The rootset in the modified Kaffe JVM would require modification so that the roots from the list made available by the rootsObject object pointed only to Kaffe objects and not JAVA objects.

Using the above analysis, a hardware accelerator for garbage collection was proposed. If an accurate garbage collector is to be implemented, information which makes it accurate must be provided. This implies that the collector must always be able to precisely distinguish references from non-references values. A tracing garbage collector determines reachability of objects, i.e., object liveness, from some set of roots. In JAVA , the JAVA stack forms one component of the rootset. Therefore, a stack map must be available. A *stack map* describes which locations on the JAVA stack contain references. Maintaining this stack map may introduce both performance and complexity overhead [74]. Each time a reference appears/disappears on/from the stack, the stack map must be updated. This type of bookkeeping is also necessary when implementing a reference counting collector, since this must update the corresponding reference counting each time a reference is created or destroyed. The overhead of maintaining the stack map could be significantly reduced if it could be supported in hardware.

Our study of the allocation behavior showed that JAVA could take advantage of a generational garbage collection approach. The generational garbage collector we implemented indicated that in order to take full advantage of the generational approach write barriers are required. The write barriers trap all the pointer writes in order to take special action in case an old-to-young intergenerational pointer is created. Moreover, write barriers are also needed in the case of an incremental approach. The overhead introduced by soft write barriers could be significantly diminished if they were implemented in hardware.

The JAVA dynamic allocation behavior study suggested that a reference counting collector only fails to reclaim a relatively small data percent. One of the advantages of reference counting garbage collection is that memory management overhead is distributed throughout the computation [75]. A DELFT-JAVA processor implementation may use a reference counting collection scheme in order to provide smooth response time. A second tracing collector could run from time to time in order to reclaim the garbage that the reference counting collector fails to reclaim. If the amount of garbage to be reclaimed by the tracing collector is not important, this scheme provides smooth overall response-time. A reference counting collector, needs to maintain the same stack map necessary in the case of an accurate tracing collector implementation. Besides marking and unmarking the corresponding bit in the stack map when a reference appears or disappears in the stack, the hardware support should also increment or decrement the corresponding reference counting field and take the needed reclaiming action in case the reference counting drops to zero.

4.8 Related Work

There are a number of projects which are investigating the usefulness of multithreading. Most of these projects are concerned with scientific workloads rather than JAVA-specific acceleration. In the multiscalar approach[76], ILP is exploited by keeping a single logical copy of the register file with physical copies distributed among parallel processing units. In the DELFT-JAVA architecture, there is no concept of independent processing units. Additionally, the register files are logically associated with each independent context although to conserve register file ports, they may be implemented within the global execution view. In the Tera MTA[77], each thread can issue an LIW from as many as 128 program counters. However, at any one time, only one stream can be active. The MIT M machine[78] uses V-threads which are also interleaved on a cycle-by-cycle basis. The DELFT-JAVA processor allows multiple instructions to be issued simultaneously from concurrent threads. We also intend this machine as a JAVA processor rather than a massively parallel machine. Both Multistreaming [79] and Simultaneous Multithreading[80] integrate fine-grained threads within a general purpose superscalar processor. Each of these processors is capable of achieving significant speedup over superscalar processors by issuing multiple instructions from several independent threads. These models are philosophically aligned with the DELFT-JAVA architecture and are shown to be powerful techniques. We extend this concept by providing architectural support for dynamic instruction translation, synchronization, and other JAVA constructs which are not currently supported in typical superscalar processors. In addition, we improve upon the organization by choosing to execute dependencies directly rather than using a superscalar organization. They based their architectures upon a DEC Alpha or an RS/6000 while we have chosen to exploit threaded parallelism inherent in the JAVA language.

4.9 Conclusions

In this chapter we have considered microarchitectural aspects of accelerating the JAVA Virtual Machine. We first considered how to dynamically translate JAVA Virtual Machine instructions into a RISC-based form. This was accomplished through the use of indirect register file access. We then considered how to accelerate dynamic method invocation. We incorporated a Link Translation Buffer to hold resolved methods. Finally we considered superscalar techniques for accelerating JAVA execution. We described a microarchitecture with multiple thread units (contexts), multiple instruction issue per context, and dependency collapsing arithmetic units. Since JAVA translation produces a large number of instruction dependencies, superscalar techniques effectively accelerate JAVA execution. Using a form of hardware register allocation, we transform stack bottlenecks into pipeline dependencies which are later removed using register renaming and interlock collapsing arithmetic units. We also described how microarchitectural support for write barriers could be used to accelerate garbage collection.

In the next chapter we will describe some example programs. In addition we will present translated programs and show how they execute on our processor.

There is no point in providing operations the compiler will not generate – Fred Brooks.

Chapter 5

Experimental Validation

In the previous chapters we gave an introduction to the JAVA Virtual Machine, discussed previous research on JAVA acceleration, provided an architectural introduction to the DELFT-JAVA processor, and described our hardware acceleration for efficient JAVA execution. We introduced the concept of dynamic translation where stack-based operations are translated on-the-fly into register-based instructions. This was accomplished through indirect access to the hardware register file. We also introduced the concept of a Link Translation buffer which stores methods and other dynamically linked JAVA constructs in a special cache that accelerates invocation. We also described a multithreaded organization that allows for JAVA acceleration.

In this chapter we describe the results of a number of experiments. We first describe a simple example based on a DSP kernel for vector multiplication. We characterize the speedup from dynamic translation hardware that can translate JAVA Virtual Machine instructions into DELFT-JAVA instructions. We describe this for several possible machine organizations and characterize the performance improvement. Next we describe the results of hardware that accelerates dynamic method invocation through the use of a Link Translation Buffer (LTB). We present results of a more complicated program - a tensor-based FFT suite. We show results for a number of Java Virtual Machines and compare those results with expected acceleration using our hardware techniques. Finally, we describe the results of our garbage collection experiments.

5.1 Test Methodology

Our general methodology for describing experimental results is to report on kernel performance. This illustrates the effectiveness of the techniques but does not require the tremendous time required to implement a full JAVA Virtual Machine and all the associated libraries written in native methods. Generally, our results are validated against both an analytical model and where possible a C++ model of the DELFT-JAVA processor.

5.2 Dynamic Translation Results

Model	Renaming	Issue	L/S units	Latency
IS	No	inorder	∞	1
IX	No	inorder	∞	1
IR	Yes	ooo	∞	1
PS	No	inorder	∞	4
PX	No	inorder	∞	4
PR	Yes	ooo	∞	4
BR	Yes	ooo	2LV/2H	4

Table 5.1: Model Characteristics

In this section we describe the results for a DSP Vector Multiply. We describe seven machine models and report on the relative performance of these models. A summary of the machine characteristics is shown in Table 5.1. The Ideal Stack (IS) model does not attempt to remove stack bottlenecks nor does it include pipelined execution. It assumes all instructions including memory operations complete in a single cycle. The Ideal Translated (IX) model uses the translation scheme described in Section 4.1. It also includes multiple in-order issue capability but no register renaming. The Ideal Translated with Register Renaming (IR) model includes out-of-order execution but with unbounded hardware resources. In addition to the ideal machines, we also calculated the performance on a more practical machine. The Pipelined Stack (PS) model assumes a pipeline latency of 4 cycles for all memory accesses to the Local Variables or Heap memory. The Pipelined Translated (PX) model and the Pipelined with Register Renaming (PR) include the same assumptions for memory latency but are equivalent to the IX and IR models in other respects.

The final experiment looked at the additional constraint of bounded resource utilization. We allowed two concurrent accesses to the Local Variable and Heap memories. We maintained a four cycle latency for each memory space.

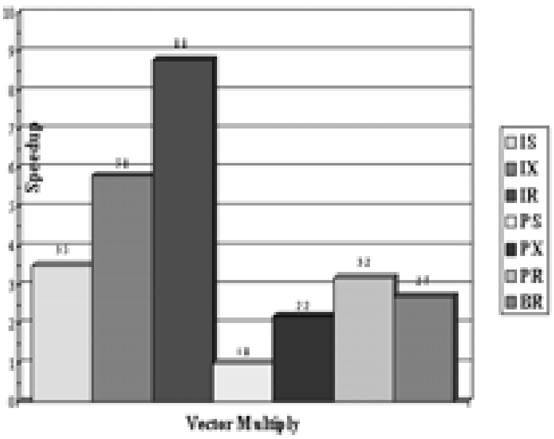


Figure 5.1: Performance Results.

Figure 5.1 shows the relative performance of each of the models. We chose the Pipelined Stack as the basis for comparison since it is a potentially realizable implementation. We note that compared with a reasonable implementation, the ideal stack (IS) model is 3.5 times faster than the PS model. When we compare the IX model with the IS model, we were able to reduce the stack bottlenecks by 40%. When register renaming was also applied in the IR model, the stack bottlenecks were reduced by 60%. When bounded resources constrained the issue capacity of the BR model, the performance still was 3.2x better than the PS model. In addition, register renaming with out-of-order execution successfully enhanced performance by about 50% in comparison with the same model characteristics but with in-order execution.

Table 5.2 shows the summary of instructions issued, peak issue rate, and overall speedup. In the unbounded resource case, a peak issue of 6 instructions per cycle was achieved with the ideal, register-renamed, out-of-order execution

Model	Peak Issue	IPC	Speedup
IS	1	1.0	3.5
IX	4	1.7	5.8
IR	6	2.5	8.8
PS	1	0.3	1.0
PX	4	0.6	2.2
PR	6	0.9	3.2
BR	2	0.8	2.7

Table 5.2: Machine Performance

model. The in-order issue peak rate was 4 instructions. When resource constraints were applied, the peak issue rate dropped to 2 and the average IPC was 0.8 even with out-of-order execution. However, the speedup achieved from the reduced stack bottlenecks was still 2.7x.

5.3 LTB Results

This section contains preliminary performance results for the DELFT-JAVA Link Translation Buffer. We investigate a range of performance the LTB can provide and then characterize the projected actual performance on an DELFT-JAVA processor. For the results presented, we assume: 1) all instructions have unit latency except for invocation instructions, 2) perfect branch prediction, 3) perfect L1 and L2 caches, 4) a single-context DELFT-JAVA processor, and 5) one instruction issued in-order per cycle.

We have built a C++ model which is capable of simulating JAVA programs that do not make system calls. The model supports dynamic translation and the Link Translation Buffer. It currently does not perform garbage collection or simulate I/O. We have used a program that produces synthetic workloads that can execute within our model. The workload generator allows the number of objects that are created and invoked to be varied. It also supports adjusting the dynamic instruction mix between invocation instructions and other instructions. Our results are based on the synthetic workloads generated by the program.

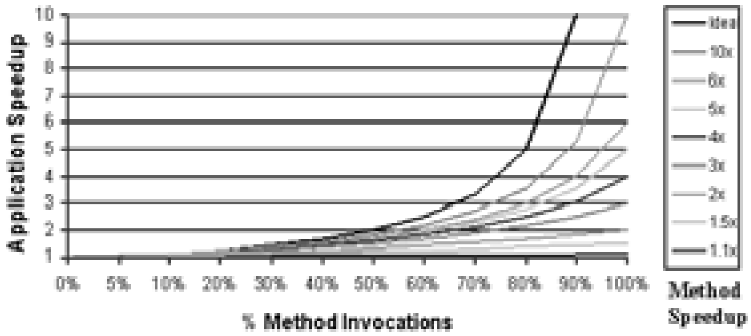


Figure 5.2: Application Speedup Versus Method Speedup

5.3.1 Available Performance

We have isolated all performance results to be relevant to a DELFT-JAVA processor with and without a Link Translation Buffer. This allows us to compare application speedup due to the LTB mechanism. Figure 5.2 shows application speedup versus the percent of invocation instructions. This relationship is based on Amdahl's law [81]. Ideal speedup refers to code that has all invocation instructions removed from the instruction stream (e.g. they execute in zero time). As an example, if 50% of the dynamic instructions are invocation instructions, then an ideal DELFT-JAVA processor would accelerate the application by 2x. In the above example, if we accelerate method invocation by 10x, we anticipate an application speedup of about 1.8x.

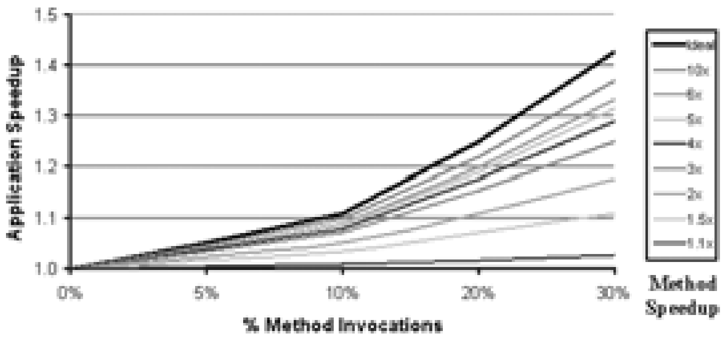


Figure 5.3: Application Speedup Versus Method Speedup

Not all applications have large opportunities for acceleration. Sun has shown an instruction distribution with less than 10% of all instructions being invocations [47, 52]. Figure 5.3 highlights a portion of Figure 5.2 and is intended to represent a more typical distribution of invocations.

Work	**Objects**	**Percent**	**Ideal**
Load	**Created**	**Dynamic Instr**	**Speedup**
WL1	2048	40%	1.67
WL2	32	10%	1.11
WL3	512	20%	1.25
WL4	1024	30%	1.43

Table 5.3: Workload Characteristics

5.3.2 Workload Characterization

Since our simulator does not yet execute API or System calls, we have generated four synthetic workloads that are intended to represent a reasonable range of JAVA programs. Table 5.3 shows the characteristics of the workloads. The first workload, WL1, creates many objects and has a high percent of dynamic instructions which access the Link Translation Buffer. An example of this type of program is object-oriented Tensor Fast Fourier Transforms (FFTs) [45]. The Tensor program is distinguished by a complex type implemented in JAVA. Many objects are created in this program. Additionally, many instance methods are recursively invoked to perform the FFT.

The second workload, WL2, creates very few objects and contains very few method invocations. This is intended to simulate code which may have been ported from C. An example of this type of program is the Press FFTs [45]. A key feature of this type of code is the lack of method invocation and the use of static (e.g. class) method invocations where possible.

The final two workloads, WL3 and WL4, represent intermediate points between the extremes of workload WL1 and WL2. They may be typical of object-oriented code that uses most of the features of the JAVA language.

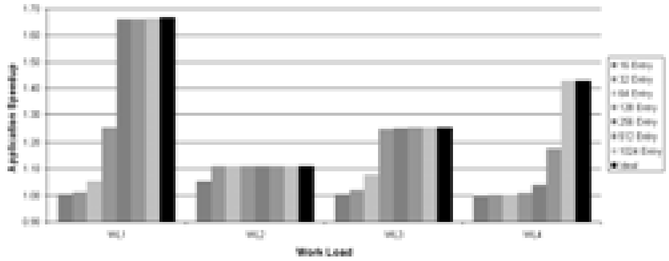


Figure 5.4: Application Speedup Versus LTB Cache Size for Workloads

5.3.3 LTB Characterization

Figure 5.4 shows overall application speedup for the four workloads versus the number of LTB entries. We assume a fully associative Link Translation Buffer with greater than 100x method invocation speedup. In addition, we use a random replacement policy. This allows us to minimize the effect of degenerate loops which invoke more objects than can fit within a particular Link Translation Buffer working set. Furthermore, these should be considered optimistic since the effect of I/O and garbage collection is ignored. We also note that these numbers include only the instructions of the simulated workload. Sun's JVM, for example, implements some of the JAVA Virtual Machine functions in JAVA . This creates additional objects in their JVM. Our simulator is implemented in C++ but is not fully compliant and does not yet execute system calls. Therefore, our model does not create any additional objects.

For workload WL1, the largest performance gain is achieved. Since this workload had the most opportunity for method acceleration, the results improve to nearly the ideal maximum if the LTB contains sufficient entries to hold the most frequently used objects. Workload W2 achieves near optimal application speedup as soon as the LTB is of sufficient size to hold the most frequently used objects.

Figure 5.5 shows the Link Translation Buffer miss rates for the four workloads. By miss rate, we mean the traditional definition - the probability of a reference made to the buffer is not in the buffer [82].

Notably, workload WL4 is the only workload that requires more than 512 entries. As Table 5.3 shows, 1024 objects are created for this workload. However,

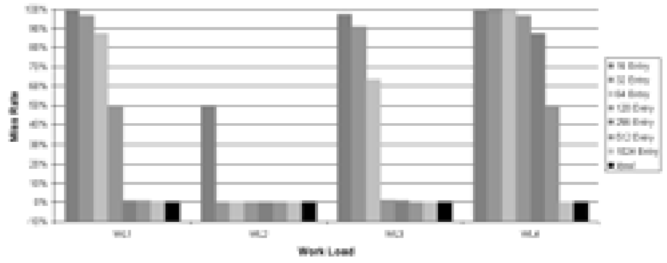


Figure 5.5: LTB Miss Rate Versus Entries for Workloads

a large number of the objects are required for the working set. Workload WL1 has 2048 objects created but its working set of objects is much smaller and therefore achieves lower miss rates at 256 LTB entries.

5.3.4 LTB Conclusions

In most Instruction Set Architectures, it is not possible to directly encode high-level operations in a single instruction. In particular, late binding method invocation is generally not supported. In this case, a C++ compiler would be required to emit a sequence of instructions that would point to a dispatch table based on the object calling the function. The dispatch tables are set up so that the offset of the function is the same regardless of the actual class. However, at the instruction level, because a sequence of memory and arithmetic operations is required, the information concerning the high-level operation is lost. Therefore, it is difficult for a traditional processor to optimize this operation. Because the DELFT-JAVA processor architecture retains the high-level information, it is possible to optimize for dynamic operations.

In conclusion, we have presented the operation of the DELFT-JAVA Link Translation Buffer. The purpose of this buffer is to accelerate JAVA's dynamic linking capability. This buffer is architecturally transparent but requires Instruction Set support for dynamic method invocation. If the target architecture contains these types of instructions, this technique may accelerate the performance of a JAVA application from 1.1x to 1.5x depending upon the number of objects utilized. The DELFT-JAVA processor architecture contains instruction set support for dynamic method invocation and may utilize a Link Translation Buffer to accelerate JAVA applications.

5.4 Java FFT Results

In this section we present preliminary results for a 32-bit full complex 4-point FFT kernel and then extend this analysis to a number of JVMs. The results are based on a C++ model of the DELFT-JAVA processor and represent figures for preresolved classes with single-cycle execution units. The FFT is compiled using Sun's javac -O. The FFT algorithm is based on Pease's tensor product decomposition[83].

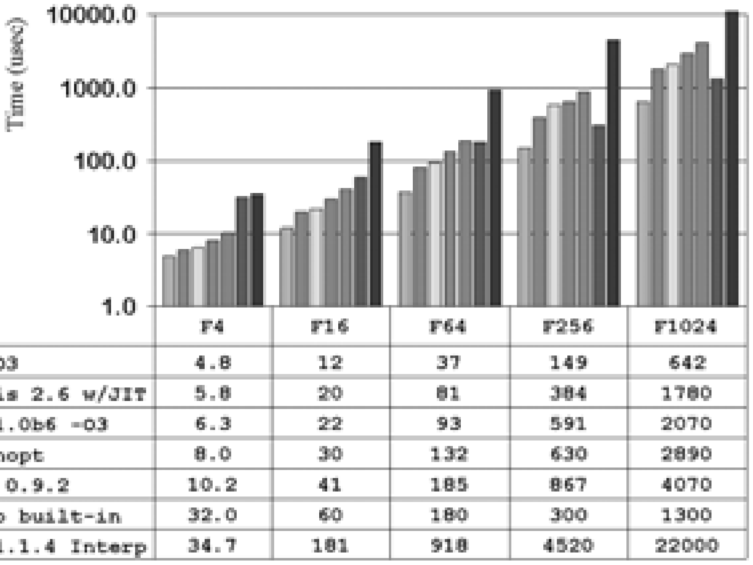
For a single-issue, single context, inorder processor, 226 cycles are required. For a single-issue, four context, inorder processor, 84 cycles are required when amortized over 4 concurrent FFT's with adequate execution units. Because the javac compiler is conservative in optimizing loads and stores, a number of instructions are generated that could be further optimized. Because we are accelerating Java programs produced directly from a Java compiler, we do not use any of the multimedia datatypes which would enhance the FFT performance. We anticipate with better optimizations and multiple issue per thread, the FFT performance of the DELFT-JAVA processor will improve by 10x based on a similar algorithm used in [84].

We now analyze the impact of C versus Java programming of FFT kernels. In the above analysis, the DELFT-JAVA processor's performance on an F_4 direct implementation was described. We now extend this to generalized FFTs and compare the perfromance of various JVMs and corresponding direct execution C performance.

$$\begin{aligned}
F_4 &= \text{direct implementation} && (5.1)\\
F_{16} &= P_4^{16}(I_4 \otimes F_4)P_4^{16}T_4^{16}(I_4 \otimes F_4)P_4^{16} && (5.2)\\
F_{64} &= P_{16}^{64}(I_4 \otimes F_{16})P_4^{64}T_4^{64}(I_{16} \otimes F_4)P_{16}^{64} && (5.3)\\
F_{256} &= P_{64}^{256}(I_4 \otimes F_{64})P_4^{256}T_4^{256}(I_{64} \otimes F_4)P_{64}^{256} && (5.4)\\
F_{1024} &= P_{256}^{1024}(I_4 \otimes F_{256})P_4^{1024}T_4^{1024}(I_{256} \otimes F_4)P_{256}^{1024} && (5.5)
\end{aligned}$$

All experiments were conducted using a 300MHz Sun Ultra-Sparc II with 128MB of memory running Solaris 2.6. For all experiments, multiple iterations were performed to minimize start-up effects. The average time is reported based on a time-of-day function in a root window at maximum priority in an otherwise nearly idle system. We measure the execution time of the algorithms and not the time to load the interpreters or the time to compile the C code.

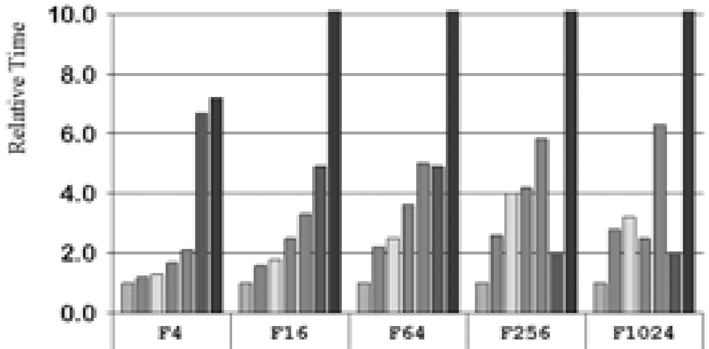
The first experiment investigates a traditional FFT algorithm from Press[85]. The *C code* is compiled using gcc 2.7.2. Results are presented for unoptimized and -O3 optimized execution. The C program was then hand-translated into an equivalent Java program. The resulting code does not take advantage of most of the Java programming constructs. It is essentially the same C code with a Java class wrapper. Results are presented for -O optimized Java code. There were no practical differences between unoptimized and -O optimized Java results. The *Matlab* numbers are the times for the built-in FFT routines. The *Sun Solaris 2.6* results are for the built-in Java interpreter with JIT that comes standard as part of the Solaris 2.6 installation. The *Sun JDK 1.1.4* results are reported for comparison. The *Kaffe* 0.9.2 just-in-time compiler results are also presented for comparison . The *Toba*[42] results are for the 1.0b6 beta release. Toba *off-line translates* Java bytecodes to C code. The C code was then compiled with gcc 2.7.2 with -O3 optimization.



	F4	F16	F64	F256	F1024
▨ gcc -O3	4.8	12	37	149	642
▨ Solaris 2.6 w/JIT	5.8	20	81	384	1780
□ Toba 1.0b6 -O3	6.3	22	93	591	2070
▨ gcc unopt	8.0	30	132	630	2890
▨ Kaffe 0.9.2	10.2	41	185	867	4070
■ Matlab built-in	32.0	60	180	300	1300
■ Java 1.1.4 Interp	34.7	181	918	4520	22000

Figure 5.6: Traditional FFT Speedup

Figure 5.6 and Figure 5.7 shows that the best performance in all cases was for optimized C code. A notable point is that the Sun Solaris 2.6 platform came within 20% of the optimized C code for the smallest FFT but diverged to



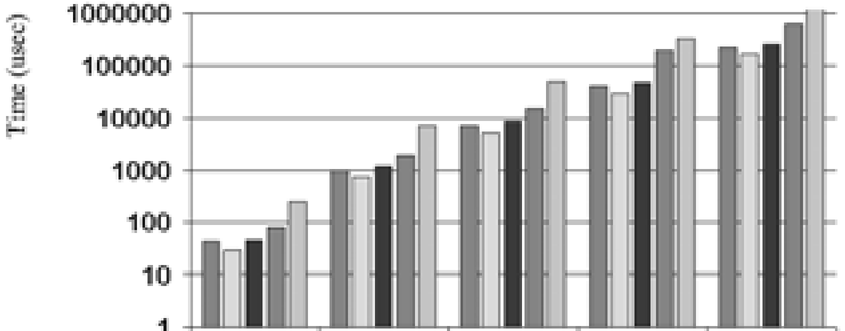
	F4	F16	F64	F256	F1024
▫ gcc -O3	1.0	1.0	1.0	1.0	1.0
▪ Solaris 2.6 w/JIT	1.2	1.6	2.2	2.6	2.8
▫ Toba 1.0b6 -O3	1.3	1.8	2.5	4.0	3.2
▪ gcc unopt	1.7	2.5	3.6	4.2	2.5
▪ Kaffe 0.9.2	2.1	3.3	5.0	5.8	6.3
■ Matlab built-in	6.7	4.9	4.9	2.0	2.0
■ Java 1.1.4 Interp	7.2	15.0	25.0	30.0	34.0

Figure 5.7: Relative Traditional FFT Speedup

about 3 times slower for larger FFTs. This is a significant improvement over the JDK 1.1.4 platform. The new results improve Java's performance up to 12 times versus the JDK 1.1.4. In addition, the Solaris 2.6 Java platform is comparable in performance to the Toba off-line compiler which produced very impressive results. Matlab's built-in FFT libraries are also very competitive with optimized C - particularly as the FFT size increases. The Kaffe JIT is competitive when compared to the JDK 1.1.4 but does not perform as well as the Sun Solaris 2.6 platform.

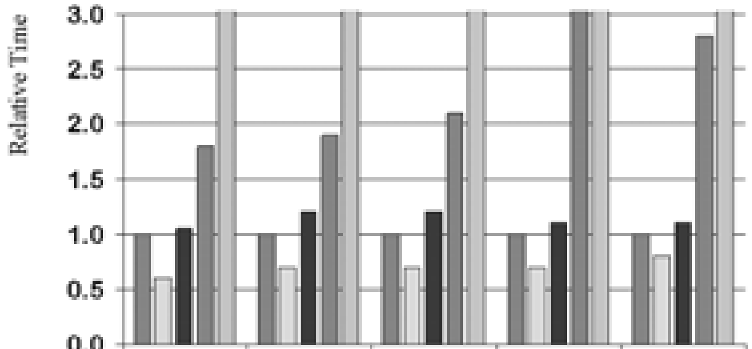
The second experiment investigates the effects on performance when many of the features of the Java language are involved. Unlike the code of the first experiment, the Java Tensor library makes use of many of the Java programming constructs. In particular, inherited class objects are used and the code is reentrant to allow for multithreaded subroutines. Because of this, many objects are created which require garbage collection.

Figure 5.8 and Figure 5.9 shows the results of the Tensor algebra library for Matlab and Java. In all cases, the Toba compiler performs better than any other alternative. Most interestingly, the performance difference between the



	F4	F16	F64	F256	F1024
▨ Solaris 2.6 w/JIT	44	1000	7200	41000	221000
□ Toba 1.0b6 -O3	28	730	5100	29000	166000
■ Java 1.1.4 Interp	46	1200	8400	47000	254000
▨ Kaffe 0.9.2	81	1900	15000	196000	629000
▨ Matlab Interp	250	7100	50000	318000	3470000

Figure 5.8: Tensor FFT Speedup



	F4	F16	F64	F256	F1024
▨ Solaris 2.6 w/JIT	1.0	1.0	1.0	1.0	1.0
□ Toba 1.0b6 -O3	0.6	0.7	0.7	0.7	0.8
■ Java 1.1.4 Interp	1.1	1.2	1.2	1.1	1.1
▨ Kaffe 0.9.2	1.8	1.9	2.1	4.8	2.8
▨ Matlab built-in	5.7	7.1	6.9	7.8	15.7

Figure 5.9: Tensor FFT Relative Speedup

Solaris 2.6 platform and the JDK 1.1.4 are within 20% for all cases. This is significantly different than the 12x improvement noted in the first experiment. We postulate that the JIT compiler may not have been able to perform as many optimizations due to the large number of objects created. A profile of each execution stream showed computations to be the predominant time factor in the first experiment while object creation and method invocation were much larger percentages of the execution time for the second experiment. Toba, on the other hand, may take as long as necessary to compile (although our experience is that it is not much more than a few minutes) and therefore produces more optimized code. Also, it may not be proper to compare these FFT results with the results presented in Figure 5.6 and Figure 5.7. The tensor libraries were written to assist partitioning of parallel code onto multiprocessors and multithreaded processors and not particularly for absolute performance on a uni-processor.

We have compared FFT performance in Java, Matlab, and C. We have found that Java may offer sufficient performance for FFTs. However, when compared against native C code, the best Java off-line translation and JIT's may still execute 2 to 3 times slower than an equivalent algorithm written in C. However, Sun's latest JVMs have significantly closed the gap between native C performance and Java. The JVM shipped as part of the Solaris 2.6 operating system is nearly 10 times more efficient on C-like Java code than the JDK 1.1.4 and produces results within 20% to 60% of optimized C for small FFTs. The Kaffe JIT runs about 2 to 5 times slower than the interpreted code for the tensor model. We attribute this to the large number of objects created and the requirement for an efficient garbage collector. We also note that the state of Java compilation is relatively immature compared to C compilers. As Java compilers become more sophisticated, the gap may narrow further. Furthermore, direct compilation of Java to native machine code may provide performance closely rivaling C. In addition, raw performance is not the only metric influencing the success of a product. For example, Matlab has an exceptionally rich library of efficient signal processing routines. The ease with which these are integrated with other Matlab programs makes Matlab an excellent DSP development environment. In addition, the number of lines of code written for the Matlab program was less than for Java due to the built-in Complex type. Finally, we conclude that for applications that make predominant use of Java, application specific processors may accelerate Java execution to be at least on par with and potentially better than C code execution on a traditional processor. Depending upon parameters chosen (e.g. issue rates, clock speed, and functional unit latencies), we have estimated that a DELFT-JAVA processor can also execute in comparable time relative to native C code. However, at this point our models do not take into account I/O or garbage collection.

5.5 Garbage Collection Results

After constructing a reference counting collector for the Kaffe JVM to instrument the Java language and establishing the most usual pattern of dynamic memory allocation, we used the results of our experiments in implementing different garbage collectors for the Kaffe JVM. In particular we developed a copy collector and the framework for a generational collector. We then compared their performance with the performance of the original mark and sweep collector.

5.5.1 Copy Collector Experiment

In one of the experiments we implemented a copy collector. The original collector of the Kaffe JVM is a mark and sweep conservative collector. It is used to collect both the Java objects when they become garbage (e.g. objects created as result of interpreting the Java user program) and all other dynamically allocated objects used by the internal structures of the Kaffe virtual machine.

Theoretically, there are a few advantages a copy collector has over a mark and sweep collector:

- lower allocation costs: Dynamic memory is allocated much simpler and only requires incrementing the value of a pointer (the free pointer [73]). The out of space check is also a pointer comparison that simply checks if by incrementing the free pointer the bounds of the available free space are exceeded.
- less fragmentation: By successively copying the surviving objects, a copy collector also implicitly performs compaction. A mark and sweep collector only picks up the garbage among the live objects.
- reduced complexity: The theoretical time complexity of a copy collection algorithm is proportional to the memory occupied by the live objects in the heap at the moment the garbage collection occurs. This is better than that of a mark and sweep collection algorithm which is proportional to the heap memory dimension. The mark and sweep collector must touch all the memory in its sweep phase. The prior complexity is

lower because only a part of the heap memory is occupied by live objects. We studied this and found that in fact this advantage does not hold in practice.

A copy collector is a collector that moves objects in the heap memory. Thus, a copy collector needs to know exactly whether a memory location contains a pointer or a non-pointer value. If this distinction can not be made accurately, a copy collector will treat an object that is not a pointer as a pointer and wrongly update the pointer-holding location to reflect the hypothetical move of the object. The location will then no longer hold valid data. Conversely, if a location contains a pointer-value but it is not recognized as such by the collector, when the object moves, the location will not be updated and will continue to point to the old copy that is now an invalid object. A copy collector can only be an accurate collector (in contrast to a conservative one).

The original garbage collector of the Kaffe JVM takes care of both types of dynamically allocated objects, i.e., Java objects and objects internally used by the Kaffe JVM. The latter case is handled through normal C-pointers and there is no way of using a non-conservative collector since it is not possible to accurately make the distinction between allocations containing a valid reference to an object and a location containing an integer which happens to hold an integer value which resembles to a valid reference. That is why the garbage collector for the Kaffe JVM is a conservative one.

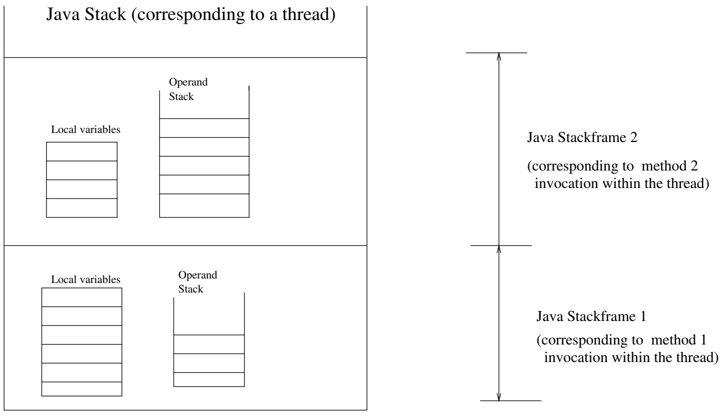

Java Stack (corresponding to a thread)
Operand Stack
Local variables
Java Stackframe 2
(corresponding to method 2 invocation within the thread)
Local variables
Operand Stack
Java Stackframe 1
(corresponding to method 1 invocation within the thread)

Figure 5.10: A thread's stack in Java

We wrote an accurate garbage collector for the Kaffe JVM by separating garbage collection for Java objects, for which accurate collection is possible, from the garbage collection for other objects used internally by the Kaffe JVM. We accomplished this by separating the Java stack (containing the Java frames which hold the locals and the operand stack for a method invocation) from the stack used by the Kaffe JVM. Practically, we chained together the Java stack frames contained in the frames on the system stack. A Java stack is depicted in Figure 5.10.

As the content of the locations in the Java stack dynamically changes with the interpretation of the Java program, we also needed a parallel structure to the Java stack frames to keep track of what locations contained pointers at any moment garbage collection could occur. This is similar to the one used in the reference counting garbage collector for the Kaffe JVM. We also let the original mark and sweep collector run so that garbage collection for the non-Java objects was completed. This was so our collector did not have to account for them. However, we made a clear distinction between requests of allocation for user Java-objects and the other objects. We treated each of them separately so its corresponding collector was responsible for the appropriate objects.

■ **The Allocator:** We redirected all the dynamic memory request coming as result of interpreting the user bytecodes from the original allocator to our allocator. This is done from each of the four JVM instructions used to create Java objects: NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY. We used a simple allocation algorithm from [75] and implemented it by the cc_heap_malloc() function.

The memory allocator first checks to see if enough space is available to satisfy the memory allocation request. If enough space is available it allocates the requested space by simply incrementing the freePointer and returning a user pointer to the allocated memory (the object reference). Otherwise it initiates a garbage collection cycle. After this garbage collection, if enough space is available the request is satisfied. Else, the allocator fails, returns 0, and the JVM will raise an Out Of Memory Exception. Additional space is reserved for the object header.

The allocator manages from the whole available memory one half of it at a time. It knows that it can always allocate memory contained between a freePointer and the topOfSpace. These pointers are initially set along with the other pointers used by the garbage collector and are updated after every garbage collection cycle to point to new locations in the new memory space containing the live objects.

■ **The Collector:** The copying garbage collector is invoked directly by invoking System.gc() from a Java program. It is also invoked when an allocation request fails because there is not enough free space. It is similar to the interface of the mark and sweep collector but contains two new functions: copyObject() which will be used for copying live objects from the current semispace (fromSpace) to the new semispace (toSpace) and the function used to walk memory CCwalkGC Memory().

Each object type is registered with the copying garbage collector. For each object type there is a correspondent walking function that is registered when the Kaffe JVM starts in the gcFuncs.c file in the ccInitCollector() function. The walking functions are used in the copy phase of the garbage collection. The correct function is chosen by the collector according to the type of object being walked so that pointers contained in the object are properly updated and the objects pointed from within this object are also registered as being alive. The walk function for String objects just copies the array of chars which represent the String.

With the previously mentioned functional support we developed a four phase copying collector. In phase 1, the CCstartGC() function tells the collector to start the copy (save) of live objects from the current semispace (fromSpace) to the other semispace (toSpace). At the end of the garbage collection, all the live objects will be moved in the toSpace and all the accessible references to these objects will be updated to point to the new locations of the objects. The collector first stops the other working threads for the interval of time in which it will perform garbage collection. Then objects that were found dead already at a previous collection but still need to be finalized are copied. Otherwise these objects would get lost. If it is the case (which is characteristic of the generational copying collector) that other objects registered explicitly are also treated as roots, these objects are then copied. In our implementation, roots are considered all the object to which references exists in the Java stack that we originally separated from the system stack. The information as to whether a location in the Java stack frame is a pointer is available in the structure we maintain parallel to the Java stack frames.

After phase 1 is completed all the copied objects are scanned (walked) in phase 2. This determines whether other live objects exist. We then save them into the new semispace and update references within the copied live objects. This second phase ends when all the live objects have been scanned (when the scanPointer catches up with the newFreePointer). A forwarding pointer is set in the old object copy (in the tospace) at the time the object is copied into fromspace.

The object references encountered during scanning are tested in order to see whether the object was already forwarded (i.e. copied) or not. If the object is already forwarded, then the forwarding pointer is written instead of the reference to the old object. If the object is not forwarded then it needs to be copied and the forwarding pointer must be set to point to the new copy. Since the number of live objects is limited, it is ensured that at the end of this phase all the accessible references to live objects will be updated.

All objects remaining in the old semispace and not being copied to the new one are dead. Before they can be reclaimed, a scanning of the whole old semispace is needed in order to find dead objects that need finalization. This is performed in phase 3. These are Java objects which request special actions to be taken after they become garbage and before they are effectively destroyed. The Java Language Specification guarantees that these actions will be executed before these objects needing finalizing are effectively reclaimed[1]. This third phase has complexity O(size of the useful heap). This means that the entire garbage collection process will have at least this complexity. This is nearly the same as the complexity of the sweep phase of a mark and sweep collector. Thus, the theoretical advantage of copy collection over mark and sweep collection is minimized due to the objects' finalizing requirement. This applies generally to all languages in which object finalization is supported. All objects needing finalization are also saved in the new semispace. Even if they are dead, their finalization methods are executed just prior their effective destruction. This requires that they be kept alive untill the finalizer executes the finalize method. The resuscitation of the objects needing finalizing can also bring to life other objects. This requires a new scan of the resuscitated objects untill the scanPointer catches up again with the newFreePointer. After the garbage collection completes, if there are objects needing finalizing, the finalizer is executed. The user program can then continue.

■ **Results:** We first compare the original Kaffe Java Virtual Machine and the modified Virtual Machine measuring the performance of the garbage collection. Our copy collector can allocate memory fast and efficiently since it only needs to make a pointer comparison and increment when it serves an allocation request. It also avoids fragmentation since compaction of the heap is intrinsic to a copy collector. However, some overhead is introduced by maintaining a structure that provides the garbage collector the layout of the Java stack (e.g. a stack map containing which locations are pointers). This is likely to be significant as many instructions in the JVM (e.g. ALOAD, ASTORE, NEW) are used extensively and imply the updating of this stack map structure. Reduction of the copy collector overhead can be achieved if the stack map update can be completely or partially implemented in hardware as is the case for the DELFT-JAVA processor.

We used some of the benchmarks from the JVMspec98 suite to evaluate the performance of our collector. The time performance of the modified Copy Collected Kaffe JVM (CC Kaffe JVM) is very close to the performance of the original Mark and Sweep Collected Kaffe JVM (MS Kaffe JVM). For example, in the case of the jess benchmark, the original finished in 3576 seconds while our modified virtual machine completed in 3712. The relative difference is 3%.

For the mpegaudio benchmark which allocates relatively little dynamic memory, our modified CC Kaffe JVM performed more efficiently. It took it 3269 seconds to complete compared to 5186 seconds how long the original MS Kaffe JVM. Our modified machine performed 36% better in terms of time performance. Only one collection was needed for the modified JVM. No collections were needed for the original JVM. The better time performance of the CC Kaffe JVM in this case is due to the faster memory allocator.

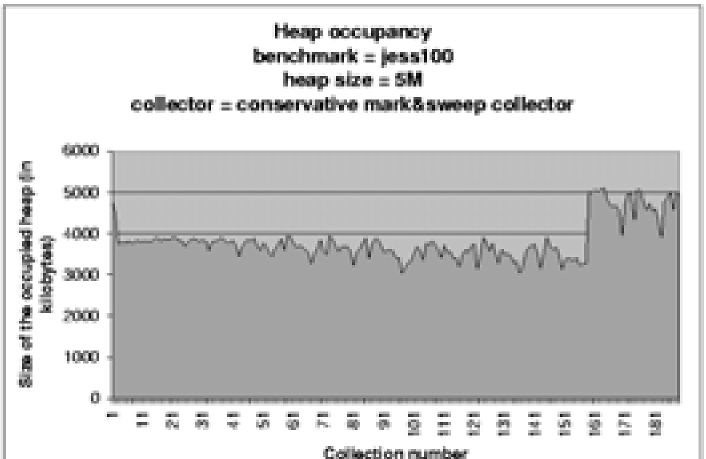


Figure 5.11: Heap occupancy in the jess benchmark, when interpreted by the MS Kaffe JVM with a heap of 5 Megabytes

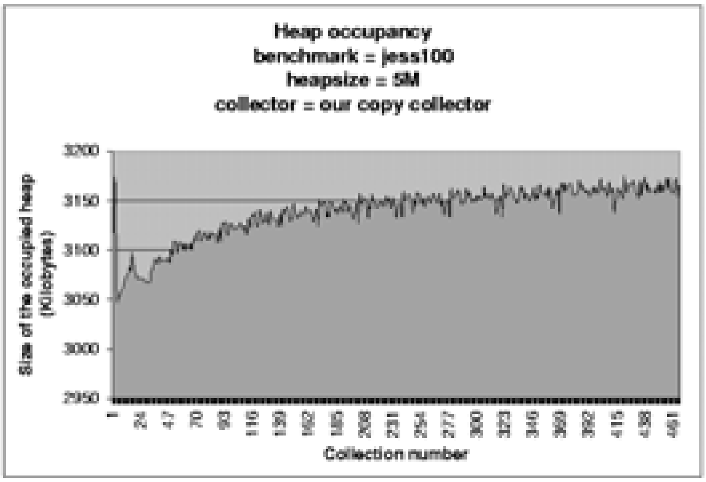


Figure 5.12: Heap occupancy in the *jess* benchmark, when interpreted by the CC Kaffe JVM with a heap of 5 Megabytes

The CC Kaffe JVM in some cases also performed better in terms of memory. In the jess benchmark program the heap used by the CC Kaffe JVM was 5120 K (both semispaces) and the maximum heap occupancy (the total amount of memory effective occupied) was 1420K. This means our modified machine could run in a heap memory space of at least 1420x2=2840K (since there are two semispaces). The memory occupied by the other non-Java objects that was not collected by our copy collector was at most 1400 K. The original machine used a total heap of 6144K and had the maximum heap occupancy of 5039K. This means that the original required a memory space of at least 5039K. If we compare this with the memory needed in our case, even adding the memory used by the non-Java objects (2840+1400 = 4240K) our collector required 15% less memory for the jess benchmark. The dynamic evolution of the effective jess heap dimension is presented in Figures 5.11 and Figure 5.12 for the CC Kaffe JVM.

Similar results were obtained for a heap size of 10 Megabytes as shown in Fig-

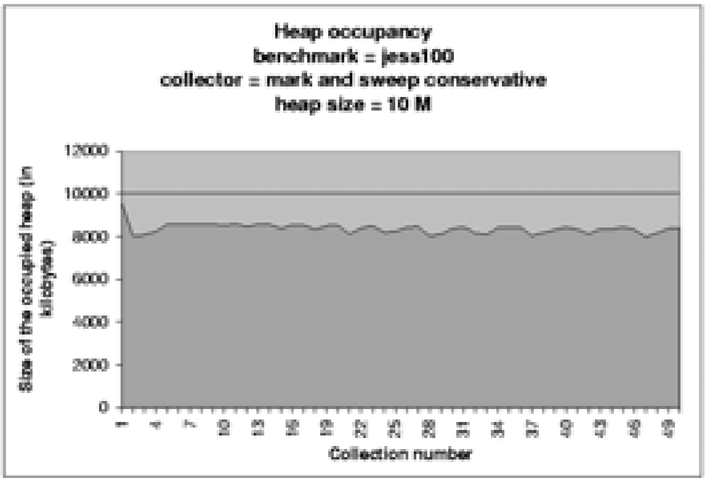


Figure 5.13: Heap occupancy in the *jess* benchmark, when interpreted by the MS Kaffe JVM with a heap of 10 Megabytes

ure 5.13 and Figure 5.14. As in the case of the previous copy collector results, the memory occupied by other dynamically allocated objects (needed for the internal use of the JVM) is being taken into account. The maximum occupied memory by the Java objects in the copy collected heap was 2817 K. This implies that the copy collected heap could have been at most 2817x2=5634K. Furthermore, adding the total dynamic memory in the mark and sweep collected heap for the internal JVM objects(981K), we obtain a total memory requirement of 5634+981=6615K. The maximum heap occupancy in the case of the original mark and sweep collector was of 9621K. Thus, more then 30% more space was required by the JVM using only the accurate copy collector.

5.5.2 Generational Collector Experiment

The results presented in the previous section have showed that generational garbage collection for Java is meaningful because most Java objects live a very short time. Only a small percentage of Java objects survive multiple collections. Thus, we modified the previous garbage collector to implement a

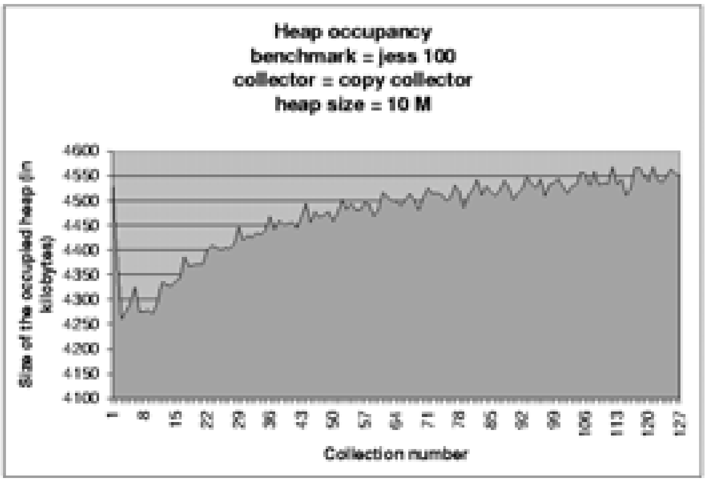


Figure 5.14: Heap occupancy in the jess benchmark, when interpreted by the CC Kaffe JVM with a heap of 10 Megabytes

generational collector. In our approach we have only two generations: the younger generation (YG) and the older generation (OG). This partitioning was clearly indicated by the lifetime analysis of the Java objects.

To fully benefit from generational garbage collection it must be possible to collect the younger generation(s) without collecting the older one(s). Since liveness of data is a global property[86], old-memory must be taken into account. For example, if there is a pointer from old memory to new memory, that pointer must be found at collection time and used as one of the roots of the traversal[1].

Ensuring that the collector can find pointers into young generation(s) requires the support similar to the write barrier of an incremental collector. Each potential pointer store must be accompanied by some extra-bookkeeping in case an intergenerational (from old to young) pointer is being created. Using these inter-generational pointers as roots ensures that all reachable objects in the

[1]Otherwise, an object that is live may not be properly preserved by the garbage collector. Additionally, the pointer may not be updated appropriately when the object is moved. Either event destroys the integrity and consistency of data structures in the heap.

younger generation are accessible by the collector. In the case of a copying collector, it also ensures that all pointers to moved objects are appropriately updated.

When implemented in software, write-barriers are expensive in terms of time-penalty. The interpreter needs to trap any pointer store and execute extra actions each time an old-to-young inter-generational pointer is created. As hardware support from a DELFT-JAVA processor is to be expected for write-barrier implementation, we have not implemented soft write-barriers for this study. However, in order to take into consideration the old-to-young intergenerational pointers, we had to consider all the pointers in the older generation as roots for the garbage collection of the young generation.

The age of the objects is measured as the number of garbage collections they have survived. Young objects are promoted to the OG when they reach the run-time configurable parameter PROMOTE AGE. An age field was added in the header of the object for the generational collection. The age field was incremented every time the object survived a copy collection in the ccCopyObject() function.

When an object reach the PROMOTE AGE (the age at which it is considered old enough to not die soon) the object is moved to a so-called stable set. This stable set is kept apart from the heap where "younger" objects reside. The objects contained in it are considered alive untill a major collection takes place. They will then be treated again as normal objects and will be reclaimed if they become garbage. Since they are considered a priori live, in our implementation the old objects are registered with the collector as roots using cc_add_ref(newObjp) for the minor collections. Conceptually, classes belong also to the stable set when no class garbage collection is explicitly requested. Objects are considered live and are not subject to garbage collection even if they may become dead. The user that has explicitly requested no class garbage collection knows that they are very likely not to be dead or it is possibly more convenient to treat them as being always alive. In much the same manner we first treat old enough objects as alive knowing that it is very likely that they will not be dead at the next minor collection.

These objects will still be scanned when a minor collection takes place but they will not be repeatedly copied from one semispace to the other. Copying objects can be very time consuming especially if objects are large. If objects prove to live long there is no need to copy them. The above approach should perform better than the non-generational one.

In our implementation when an object is promoted, space for it is reserved from the original mark and sweep collector gc_malloc(size, CC_GET_FUNCS(header). This collector is still running for all the non-Java objects. As this is a non-moving collector our goal is achieved.

■ **Results:** While providing the framework for fully generational collection, our approach does not fully benefit from the generational approach. When collecting the young generation, our collector must also scan all the pointers in the older generation in order to find and take into consideration the old-to-young intergenerational pointer updates. However, the repeated copying of old objects is avoided by keeping the old generation in a memory space managed by the non-moving mark and sweep collector. The younger generation is copy collected and the heap is continually compacted. However, our analysis showed that the time saved by avoiding irrelevant moves of older object was not important. Therefore, the simple copy-collector and the copy-collector using two generations had virtually the same time-performance.

Furthermore, when using a generational garbage collector a number of trade-offs are possible between memory and time performance. We illustrated these possible trade-offs by adjusting the PROMOTE AGE of the objects. We further presented the results of our experiments obtained on the jess benchmark from the jvmSPEC98 benchmarks suite. The interpreting of the benchmarks by the non-generational CC Kaffe JVM requested 467 garbage collections. Thus, when run with a PROMOTE AGE of greater than or equal to 467, the generational collector turns into a non-generational one. We provided results of varying the PROMOTE AGE between 0 and 467 and we illustrated the possible trade-offs.

As a generational garbage collector is based on the assumption that the promoted objects are unlikely to die soon, it is conservative[2]. In our experiments we found that promoted objects which become garbage must be kept alive for finalization. This in turn keeps other objects alive that would have been otherwise reclaimed. In our case the grade of conservativeness was given by the objects' promote age; the greater the promote age, the more conservative the collector. The more conservative the collector, the more memory the system that uses the garbage collector needs. This can be observed in Figure 5.15 which depicts the maximum amount of memory needed for the YG (young

[2]The term conservate as applied to garbage collection refers to the property of a garbage collector to make "conservative" approximations about object liveness.

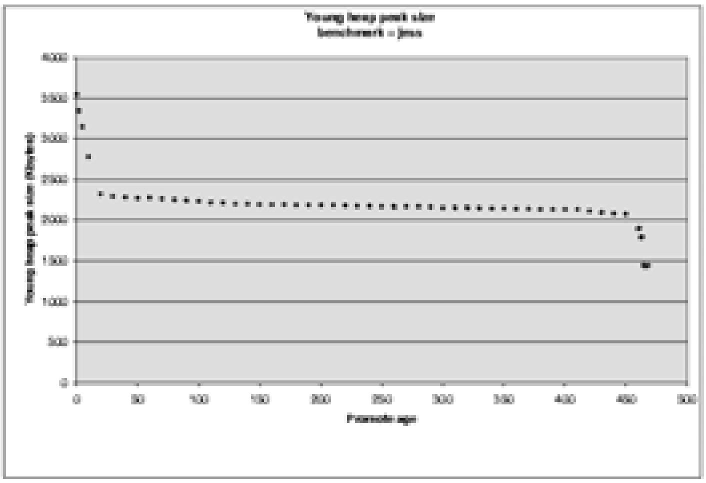


Figure 5.15: Young heap peak size in the *jess* benchmark

heap peak size). The young heap peak size is diminishing with growing promote age.

As is evident in Figure 5.15, the biggest young heap is needed when objects are promoted as soon as possible (i.e. after they have survived one garbage collection). The least heap memory required for young object is when no objects are promoted. This correspond to a non-generational collector. However, as the Figure 5.15 suggests, an amount of memory not far from the minimum is achievable if a reasonably small promote age is used. This provides a convenient trade-off between execution time and memory consumption. The advantage of a relatively small promote age is that the number of objects the garbage collector has to deal with is smaller since more objects are promoted to the OG.

Figure 5.16 depicts the sum of the sizes of the promoted objects. The less the promote age is, the more objects are promoted. After a pattern similar to that of Figure 5.15, at a reasonably small age, the total amount of promoted memory

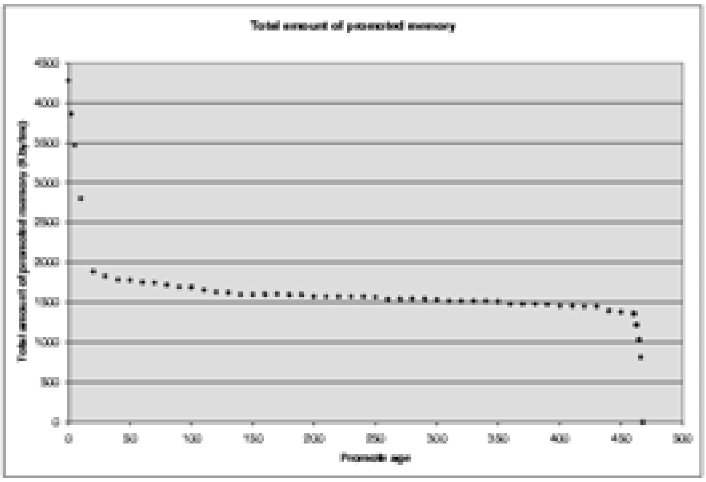


Figure 5.16: Total amount of promoted memory jess benchmark

is close to the smallest promoted memory. Thus, if a reasonably small age is used, a relatively small heap for the older objects is needed.

The number of garbage collections performed by the modified JVM is represented in Figure 5.17 as a function of the promote age. When the promote age is small, the objects are promoted faster and the young objects' heap fills more slowly. Thus, a smaller number of collections is required. If the promote age is large, old objects remain longer in the heap and the heap becomes full faster. Thus, more collections are required. The dependency of promote age and number of minor collections required is almost linear as Figure 5.17 suggests. However, the heap size required diminishes drastically with the promote age as Figure 5.15 shows. This suggests that even if a greater promote age requires a greater number of garbage collections, the total time required for garbage collection is likely to be less even for a greater promote age. In our implementation, time-performance characteristics remain nearly the same for different promote ages. This is because even if the young objects' heap dimension decreases with the promote age, memory needs to be scanned for pointers

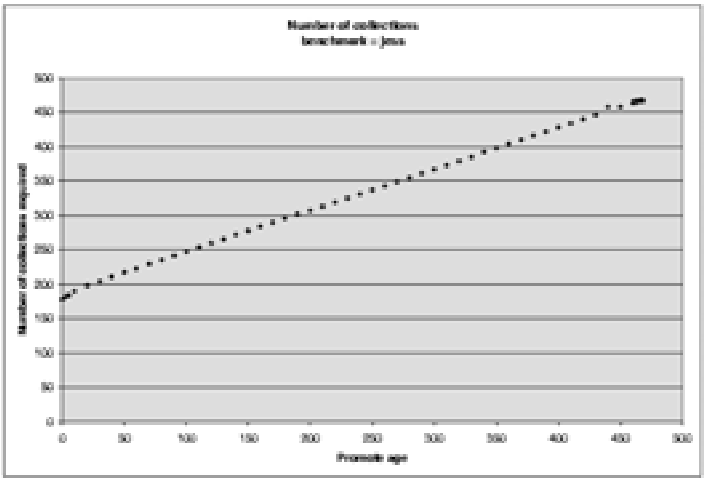


Figure 5.17: Number of collections required for different promote ages in the jess benchmark

by the collector in order to find old-to-young intergenerational pointers.

5.6 Conclusions

In this chapter we described the results of a number of experiments. We first described a DSP vector multiply. We compared this for a number of machines and presented relative performance improvement. We showed that when compared with a single-issue pipelined stack machine, a significant speedup can be realized. We also showed that out-of-order execution machines are particularly attractive for stack-based Java code. This is due in part to the large number of dependencies generated by stack references. A reasonable out-of-order implementation may yield a performance improvement of 2.7x in comparison with a single-issue stack model.

We also showed that a Link Translation Buffer can be effective in accelerating Java dynamic method invocation. We assumed a 100x speedup for resolved references that are loaded into the LTB cache. This is reasonable assumption because Java classes must be resolved in a complicated manner which involves multiple traversals through the Constant Pool. Using this assumption, a program with 20% dynamic references can experience a 25% speedup under fully associative cache parameters.

We compared the performance of a number of Java Virtual Machines using a tensor-based FFT library that we developed. We did this to compare the claim that Java-based programs are significantly slower than C-based programs. The Java-optimized tensor library competed attractively with C-based counterparts. It was able to achieve performance within 30% for some kernels.

We have shown that accurate garbage collection for Java is possible if Java objects are separated from other dynamically allocated objects used internally by the interpreting JVM. The copy collector implementation showed that a JVM using a copying garbage collector can allocate dynamic memory much faster than a JVM that uses a mark and sweep collector. Fast dynamic memory allocation is especially important in interpreting programs which allocate many small objects. An accurate collector implementation for Java requires that at the moments a garbage collection cycle can occur a stack map is available in order to tell the collector which locations on the Java stack are pointers. The maintaining of a stack map structure incurs much overhead over the execution of the interpreting JVM. Therefore a significant time-performance improvement is to be expected if this could be partially or totally implemented in hardware. A generational garbage collector for Java can take advantage of the Java objects tendency to die relatively very young. If a promoting policy is used in which the age of an object is measured as the number of collections to which it has survived, different memory-execution time trade-offs could be possible by simply adjusting the promote age. However, in order to take full advantage of the generational hypothesis, write-barriers needed for generational collections need to be efficiently implemented. Significant performance improvement is therefore expected if they are implemented in hardware.

In the next chapter we summarize all of our work and comment on open problems.

Science per se has little to do with politics, its results and how to use them definitely a lot. – Stamatis Vassiliadis.

Chapter 6

Conclusions

IN this chapter we summarize the results of our study. We first considered the JAVA language and its characteristics. As indicated in Chapter 1, JAVA is a C++-like programming language designed for general-purpose object-oriented programming[1]. The language includes a number of useful programming features including programmer defined parallelism support in the form of threads with synchronization, strong typing, garbage collection, classes, inheritance, and dynamic linking. The JAVA Virtual Machine (JVM) is a stack based instruction set designed to efficiently transport programs across the Internet and allow register poor processors to efficiently execute JAVA bytecode[2]. Chapter 1 described various methods of implementing a JAVA Virtual Machine. Both hardware and software strategies have been employed. The earliest JVMs were implemented as interpreters. In this method a software program emulates the JAVA Virtual Machine. Just-in-time compilation is a technique where the JAVA program is compiled just prior to execution and the resulting native code is executed. Because of the overhead and latency of compiling every method, flash compilation techniques evolved. We described this hybrid approach where a highly optimizing just-in-time compiler is integrated into the runtime environment. The compiler only optimizes loops where a performance gain is likely. Offline compilers were also described. In this technique the entire JAVA program is translated into native code prior to execution. This technique requires all methods to be available at compilation time. Chapter 1 also introduced the hardware technique of direct execution. At the time we began our investigation (circa 1996), the most prominent project was Sun's picoJava [47–49]. Since this time, JAVA direct execution has exploded. Chapter 1 also reviewed some of the more recent direct execution projects. Finally, we concluded chapter 1 with a list of open issues that we desired to investigate.

Chapter 2 was dedicated to providing a formal architectural description of the JAVA Virtual Machine. JVM instructions are not confined to a fixed length however all of the opcodes in the JAVA Virtual Machine are 8-bits[3]. This allows for efficient decoding of instructions while not requiring all instructions to be 32-bits or longer. Generally speaking, the JAVA language supports many of the basic data representation types[1]. In contrast to C, their values are not implementation dependent[59]. In addition, the JAVA language also supports `char` which is a 16-bit unsigned Unicode character and a true `boolean`[2] for relational and logical operators. While JAVA does not allow operations on C-style pointers, it does have the concept of a reference type. There are three kinds: `class, interface,` and `array` types. These objects are created on a dynamically allocated heap. Multiple references may exist to the same object. The reference values are handles (e.g. pointers) to the object. The distinction is that a reference can not be operated on arithmetically as is often done in C-style pointers. Operations in the JAVA Virtual Machine are strongly typed. The 8-bit opcode imposes the availability of only 256 opcodes, this results in the tradeoff that nearly all operations are performed as integers or IEEE-754 floating point. An interesting JAVA definition is that the JAVA Virtual Machine does not indicate overflow or underflow during operations on integer data types[3]. There are also load and store instructions which move values from memory locations to the operand stack in a very RISC-like manner. In addition to standard operations, there is direct support for method invocation, synchronization, exceptions, and arrays. Of the more unusual instructions, the `iinc` is a memory-to-memory instruction that increments the contents of a local variable location by a signed constant. There are two variable length instructions[3] - `tableswitch and lookupswitch.` In Chapter 2, we gave a brief overview of the JAVA Virtual Machine architecture. We described how stack machines operate in general and also how the stack-based JAVA Virtual Machine operates. We showed how a simple JAVA program is translated to JAVA Virtual Machine bytecode and how it executes. We gave an architectural overview of the JAVA Virtual Machine. We described the storage organization and each of the spaces the JAVA Virtual Machine can

[1] e.g. `byte, short, int, long, float,` and `double.`

[2] Note that while the JAVA language supports a `boolean` datatype, the JAVA Virtual Machine does not support it.

[3] This is in contrast with instructions of variable length where the length of the instruction is fixed but variable.

access. We described how this memory is accessed and how index arithmetic is computed. We also described all JAVA Virtual Machine operations and the data upon which they operate. Finally, we described how instructions execute and their control structures. This chapter gave the background for the work which we described in following chapters where we showed how our RISC-based architecture can efficiently translate JAVA Virtual Machine instructions into our instruction set. Chapter 2 also laid the foundation to understand why certain techniques which we have developed accelerate JAVA execution. The following chapters explained in detail the techniques we have developed as a result of our research.

Chapter 3 was dedicated to the description of the DELFT-JAVA architecture - a 32-bit RISC-based architecture. More specifically we described how to accelerate JAVA execution and provided details of the DELFT-JAVA architecture for executing JAVA Virtual Machine bytecode. Before we began our discussion we briefly described the design philosophy underlying our approach. The basic architecture implemented a Media Processor with Signal Processing capabilities. The architecture took the perspective that to maximally accelerate a compiled application, the machine language should accurately reflect the type of operations the compiler specified. Except where JAVA Virtual Machine operations were unusually complex, we preferred to allow the compiler to optimize directly to the implementation. This was independent of any particular organization. The architecture was then a superset of the JAVA Virtual Machine and provided operations that were necessary for system execution (e.g. I/O, supervision, etc.). Rather than just supporting the JAVA Virtual Machine, the architecture took a more general purpose approach. While it continued to support JAVA Virtual Machine specific constructs, it also was intended to be programmed from a number of additional high-level languages including C and C++. Chapter 2 also introduced *dynamic instruction translation*, a new approach to JAVA hardware acceleration, which was used in the DELFT-JAVA processor. In hardware assisted dynamic translation, JAVA Virtual Machine instructions were translated on-the-fly into the DELFT-JAVA instruction set. The hardware requirements to perform this translation was not excessive. In general, we dynamically translated about 80 percent of JAVA Virtual Machine instructions. Consequently, support for some JAVA language constructs were also incorporated into the processor's ISA. Without the special instruction support, many cycles may be required to emulate the operations. A key point of our architecture was introduced - where it is easy to dynamically translate JAVA Virtual Machine bytecode into DELFT-JAVA instructions, no instruction set support is provided. For those JAVA Virtual Machine bytecode which are highly complex, instruction set support is provided to allow acceleration of the function through microarchitectural support. This technique allowed application level parallelism inherent in the JAVA language to be efficiently utilized as instruction level parallelism while providing support for other common programming languages such as C/C++. In addition to the basic RISC design philosophy, there were some key organization structures that we deemed appropriate to provide architectural support for. In particular, we supported the following important categories: 1) Synchronization for multithreaded organizations, 2) garbage collection, 3) array bounds checking, 4) real-time caches, 5) multiple machines which can time-share the same datapath (e.g. the JAVA Virtual Machine and Media Processing functions) and 6) vector operations.

Chapter 4 was dedicated to describing the organization of our processor. We introduced microarchitectural support for dynamic translation, dynamic linking, and provided mechanisms for multiple thread units, multiple instruction issue, dependency collapsing, and other features common to modern superscalar processors. These techniques took advantage of key JAVA language properties to transparently extract parallelism without programmer intervention. First we described our hardware support for JAVA Virtual Machine dynamic translation. We described how indirect access to the register file provides the basic mechanism required to dynamically translate JAVA Virtual Machine instructions. Then we provided an example of the translation process. Next we described special hardware features we incorporated to assist in translation. Finally, we listed instructions which are not translated and have special architectural support. Since JAVA translation produced a large number of instruction dependencies, superscalar techniques effectively accelerate JAVA execution. Using a form of hardware register allocation, we transformed stack bottlenecks into pipeline dependencies which are later removed using register renaming and interlock collapsing arithmetic units. Second, we described how we support dynamic method invocation. We provided background on dynamic method invocation. Next we described the Link Translation Buffer (LTB) and its operation including enhancements which can be made to the LTB. Finally, we described our concurrent multithreaded organization and how multiple thread units and multiple instruction issue efficiently accelerate JAVA program execution. We also briefly described how the indirect registers and Link Translation Buffer operated within the description of the microarchitecture. We then described some features of the architecture that are not primarily for JAVA execution but allow speedup of native methods. Finally we described some related work on multithreaded architectures and presented some conclusions.

In Chapter 5 we described the results of a number of experiments. We first described a DSP vector multiply. We compared this for a number of machine organizations and presented relative performance improvement. We showed that when compared with a single-issue pipelined stack machine, a speedup of more than 2x can be realized. We also showed that out-of-order execution machines are particularly attractive for stack-based JAVA code. This is due in part to the large number of dependencies generated by stack references. A reasonable out-of-order implementation may yield a performance improvement of 2.7x in comparison with a single-issue stack model. We also showed that a Link Translation Buffer can be effective in accelerating JAVA dynamic method invocation. We assumed a 100x speedup for resolved references that are loaded into the LTB cache. This is reasonable assumption because JAVA classes must be resolved in a complicated manner which involves multiple traversals through the Constant Pool. Using this assumption, a program with 20% dynamic references can experience a 25% speedup under fully associative cache parameters. Finally, we compared the performance of a number of Java Virtual Machines using a tensor-based FFT library that we developed. We did this to compare the claim that Java-based programs are significantly slower than C-based programs. The Java-optimized tensor library competed attractively with C-based counterparts. It was able to achieve performance within 30% for some kernels.

At the beginning of our investigation we desired to answer the following questions:

- Is it possible to improve JAVA execution time with hardware?
- Is it possible to dynamically translate JAVA Virtual Machine instructions in hardware?
- Is it possible to accelerate JAVA dynamic linking?
- Is it possible to accelerate garbage collection in hardware?
- Is it possible to apply modern computer organizations to accelerate JAVA execution?
- Is it possible to overcome ILP limiting stack bottlenecks in JAVA bytecode?

- Is it possible to take advantage of the inherent parallelism expressed in the JAVA language?

When we began our study in 1996, we theorized that it was possible to significantly enhance JAVA Virtual Machine execution when hardware support was provided. We found this to be the case. Using a technique called dynamic hardware translation we were able to translate JAVA bytecode into a RISC-based architecture on-the-fly thereby allowing modern architectural and organizational techniques to be utilized in accelerating JAVA execution. A minority of instructions specified by the JAVA Virtual Machine were found to be difficult to dynamically translate. In these rare cases special architectural support was provided so that a particular processor instantiation could implement them in an efficient manner. We found architectural performance improvements of more than 2x were realizable versus other stack-based hardware methods. Given the frequency advancements of modern RISC processors, dynamic instruction translation is a significant technique for accelerating

Since JAVA specifies dynamically bound method invocation, we also investigated the feasibility of accelerating this using hardware techniques. This led to the Link Translation Buffer. This method cache was shown to be able to accelerate a JAVA program by up to 25%. It was also proposed that this cache could be extended to accelerate other JAVA specific constructs.

We investigated the lifetime of objects in JAVA programs and used the results to understand the requirements for garbage collection. We studied two particular collectors and investigated what hardware could help accelerate garbage collection. The copy collector had lower allocation costs and less fragmentation. The generational collector was very efficient because our analysis showed that the lifetime of most JAVA objects is very short.

Using modern superscalar organizational techniques, we designed a simultaneous multithreaded architecture capable of efficiently execution JAVA programs. This organization provided hardware support for multiple context instruction issue and global instruction scheduling. The organization supported multiple concurrent execution of threads which shared global execution units. We defined a *context* as a hardware supported thread unit. Each context assumed that the processor's organization incorporated (logically) an instruction cache, a decode unit, a local instruction scheduler, a local instruction issue unit, and an instruction retire unit. A context did not include any shared resources such as a first level (L1) cache, execution units, a register file, global instruction schedulers, nor global issue units. The important point of this architecture was that it transformed stack-based dependence hazards into pipeline hazards. The superscalar hardware was then able to issue multiple instructions using register renaming and other out-of-order execution techniques. Multiple contexts which could issue instructions simultaneously also provided acceleration and transparently extracted the parallelism expressed in the JAVA language.

6.1 Major Contributions

This section discusses the major contributions of our study.

■ **Dynamic Translation:** Using the statically determinable typestate property of JAVA bytecode, we proposed that it would be possible to dynamically translate JAVA bytecode into another instruction set architecture with minimal hardware. The DELFT-JAVA processor was the first to propose this technique. We found that 80% of JAVA bytecode was easily translated dynamically into the DELFT-JAVA instruction set architecture. For the remaining more complex instructions we found it to be beneficial to provide instruction set support. This was particularly true of dynamic linking instructions and complicated change of control instructions.

■ **Indirect Register Access:** To facilitate the translation process we showed that indirect access to the register file was an efficient solution. The register file could in effect function as a stack-cache allowing transparent JAVA bytecode execution. Special hardware was proposed that would automatically spill and fill the register file based on heuristic algorithms inspired from other stack-based languages and processors. The DELFT-JAVA processor, however, always executed direct register addresses. This was accomplished through a two phase translation process whereby the JAVA instruction to be executed was first translated into an indirect register address. Subsequently, the indirect address was converted to a direct physical address. This dual-translation technique was trivial to implement in machines that already support register renaming and was found to be efficient for JAVA bytecode execution.

■ **Link Translation Buffer:** We created a transparent organizational technique for accelerating dynamic linking. When a method is called in a JAVA class, the Constant Pool, which is string based, must be resolved. The resolution process is very complex and may take many thousands of cycles on a typical processor. The Link Translation Buffer caches the relevant information that was obtained through the constant pool and places it in hardware for future use. Because JAVA programmers are encouraged to use many small subroutines, this technique is important in accelerating the translation process. Our technique is effective in storing the information.

■ **Dependence conversion:** A very subtle feature of our approach is that we convert stack-based dependencies into pipeline dependencies. These pipeline dependencies are then later removed through superscalar techniques such as register renaming. Because stack-based instruction sets created bottlenecks on the top of the stack, it is important to try to remove these dependencies. Our approach removes the bottlenecks by translating the problem into another problem that is already solved by our approach. This yields to highly parallel execution without additional hardware.

■ **Multithreaded Organization used for Java acceleration:** The JAVA language is multithreaded. Parallelism is already expressed by the programmer. Furthermore, the JAVA language specifies that JAVA programmers must write thread-safe programs. This is in contrast to other languages such as C and C++ that are inherently serial. JAVA programs must protect critical sections with the synchronized keyword. Otherwise, the semantics of the language allow parallel execution of code. This is an ideal situation for processor architects. It turns out that a simultaneous multithreaded processor organization is efficient at exploiting the parallelism inherent in JAVA programs. The DELFT-JAVA processor takes advantage of this situation to exploit multiple threads concurrently executing.

■ **Thread level parallelism in Java converted to ILP:** Another subtle feature in the DELFT-JAVA organization is that we convert thread-level parallelism in to Instruction Level Parallelism (ILP). Once the instructions are translated into a RISC-based form, they are aggregated from multiple hardware contexts and issued as a single global packet. If this was not done then the number of execution units required would be equal to the number of thread units multiplied by the peak issue rate. This technique allows a single set of execution units to be shared allowing higher utilization and less cost.

■ **Multiple machines which can time-share a datapath:** One of the results of our design is that a processor that can execute both JAVA bytecode and programs compiled from other high-level languages is possible. For languages such as C and C++, a RISC-based model would be presented to the compiler. JAVA compilers do not require any modifications to execute on our processor. The operating system is responsible for controlling which machine view executes through the use of a branchJVM instruction.

■ **JVM performance characterization of DSP/FFT execution:** As part of our studies we showed that the JAVA Virtual Machine is capable of approaching C performance for some DSP kernels. During our optimization studies, we created a very efficient FFT implementation. We compared this against other interpreted languages and compiled languages. The FFT JAVA code has been requested by a number of universities and research labs.

■ **Tensor algebra as a basis for optimizing Java programs:** As part of the DSP performance studies, we optimized the JAVA programs using tensor algebra. We derived equations for the number of thread units available and showed how to distribute the FFT computation among multiple thread units. Other researchers have shown how this could automated in a compiler.

■ **Garbage collection study:** We investigated the effects of garbage collection and how hardware may accelerate garbage collection. An important result is that the lifetime of JAVA objects is very short. This suggests that a copy collector may be more efficient for JAVA programs because of the need to allocate objects quickly. We successfully separated JAVA objects from Kaffe-specific objects. This allowed us to implement an accurate generational copy collector.

6.2 Open Issues

■ **Complete JVM simulator and simulation:** In our models we did not require full a JAVA Virtual Machine to verify the architectural soundness of our ideas. We desired to show that it is possible to accelerate JAVA bytecode execution through hardware means. A simulation model was sufficient to validate the proof of concept. Future work could extend the models to allow complete simulation of a JAVA Virtual Machine. This would provide complete application performance speedup and serve to refine our estimates.

■ **Multithreaded execution with full OS results:** Our simulation results are based on uni-processor models that assume certain scheduling policies for multithreaded execution. To have a complete and accurate picture of multithreaded operation, a full operating system should be implemented with a choice of thread scheduling policies. The effect of the scheduling heuristics could then be studied.

■ **Characterizing cache effects on the LTB:** In our analysis we did not quantify performance degradation under various parameters including LTB associativity, multithreading, and non-unit latency memory access. As a further extention it would be interesting to use the instruction address as a caller's object id. This may provide a benefit similar to Sun's quick instructions without requiring additional opcodes. Another advantage of using the instruction address is that the 16-bit constant pool location does not need to be stored in the Link Translation Buffer. However, potentially more entries may be registered for the same object since more than one instruction address may invoke the same method.

■ **Using an LTB to hold additional data:** In our study we hinted that the Link Translation Buffer could be used to hold additional information such as synchronization data. We did not carry this far enough to confirm its usefulness. We postulate that this would be an efficient organizational technique to implement monitors but further investigation is necessary.

■ **Garbage collection:** In our studies we used garbage collectors that have variable timeframes for execution. Embedded and DSP environments require deterministic system responses. The collector that most closely fits this requirement is the reference counting collector. Reference counting collectors impose their own set of design constraints. These should be investigated so that real-time JAVA performance can be guaranteed.

Samenvatting

De Delft-Java Engine
Clair Johnston Glossner

In dit proefschrift beschrijven wij de DELFT-JAVA Engine - een op RISC gebaseerde 32-bits architectuur die zorgt voor hoge prestaties bij de uitvoer van JAVA programma's. Nauwkeuriger uitgedrukt beschrijven wij een micro-architectuur die de uitvoer van JAVA versnelt en verschaffen wij gedetailleerde informatie over de DELFT-JAVA Engine voor de uitvoer van JAVA Virtual Machine bytecode. De basis architectuur implementeert een Media Processor met Signal Processing mogelijkheden. De invalshoek van de aanpak is dat voor het maximaal versnellen van een gecompileerde applicatie de machinetaal een nauwkeurige afspiegeling behoort te zijn van de operatie typen die de compiler specificeert. Behalve in het geval dat de JAVA Virtual Machine operaties ongewoon gecompliceerd zijn, geven wij er de voorkeur aan om de compiler toe te staan om direkt voor de applicatie te optimaliseren. Dit is onafhankelijk van een bepaalde machine organisatie. In dit geval is de architectuur een superset van de JAVA Virtual Machine en zorgt zij voor operaties die noodzakelijk zijn voor de systeem uitvoer (e.g., I/O, overzicht, etc.). In plaats van het uitsluitend ondersteunen van de JAVA Virtual Machine volgt de architectuur een meer algemene aanpak in zoverre dat zij ook bedoeld is om geprogrammeerd te worden voor een aantal extra high-level talen inclusief C en C++. Verder introduceren wij het concept van het dynamisch vertalen van JAVA instructies, een nieuwe aanpak bij het hardwarematig versnellen van JAVA. Bij het hardware ondersteund dynamisch vertalen worden JAVA instrukties meteen vertaald naar de DELFT-JAVA instructie set. De hardware vereisten om deze vertaling uit te voeren zijn niet buitensporig. Als gevolg hiervan is ondersteuning voor JAVA taalconstructies ook ingebouwd in de Intructie Set Architectuur van de processor. Deze techniek zorgt ervoor dat het parallellisme op applicatie niveau, dat inherent aanwezig is in de JAVA taal, op efficiente wijze gebruikt kan worden als parallellisme op instructie niveau. Behalve dynamisch vertalen kan ook een speciaal Link Translation Buffer (LTB) gebruikt worden om de prestaties bij het dynamisch linken te vergroten. Verder zijn er enige organisatorische strukturen waarvan wij het belangrijk vinden dat er ondersteuning op architectuur niveau plaatstvindt, zoals: a) synchronisatie voor multi-threaded organisaties, b) garbage collection, c) controleren van array grenzen, d) real-time caches, e) het gelijktijdig gebruik van hetzelfde data pad door meerdere machines (e.g., de JAVA Virtual Machine en Media Processing funkties), en f) vector/dsp operaties. Middels het creëren van verscheidene modellen van de DELFT-JAVA Engine waren wij in staat om de prestatie metrieken van kernels die op onze processor worden uitgevoerd, te karakteriseren. Wij ontdekten dat, vergeleken

met een realiseerbare stack-based implementatie, onze aanpak de uitvoer met een factor 2,7 versneld. Bovendien hebben wij laten zien dat out-of-order superscalar machines tot 60 % van de hazards kunnen verwijderen door stack-gebaseerde afhankelijkheden om te zetten in pipeline afhankelijkheden.

Bibliography for C. John Glossner

Journal Publications

1. J. Glossner, J. Thilo, and S. Vassiliadis, **"Java Signal Processing: FFT's with bytecodes"**, *Journal of Concurrency and Experience*, vol. 10, No. 11-13, pages 1173-1178, October, 1998.

2. W.F. Lawless, C.J. Glossner, and G.G. Pechanek, **"Line Drawing using Mfast DSP Array Processor."** *IBM Technical Disclosure Bulletin*, Vol. 38, No. 08, pages 395-399, August, 1995. A parallel line rendering algorithm for multiprocessors.

3. C.J. Glossner, **"CMOS Open Drain Common I/O Test Methodology"**. *IBM Technical Disclosure Bulletin*, Vol. 31, No. 5, pages 343-344, October, 1988.

Conference Publications

1. J. Glossner, D. Routenberg, E. Hokenek, M. Mougill, M. Schulte, P. Balzola, and S. Vassiliadis, **"Towards a Very High Bandwidth Wireless Battery Powered Device"**, *In Proceedings of the 2001 IEEE Computer Society Workshop on VLSI (WVLSI)*, Orlando, Florida, April 19-20, 2001.

2. J. Glossner, **"The challenges in Microelectronics associated with broadband access"**, *Semiconductor Industry Association of Japan (SIAJ)*, Tokyo, Japan, November, 2000.

3. J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware, **"Trends In Compilable DSP Architecture"**, *Proceedings of SIPS-2000*, Lafayette, LA, October, 2000.

4. A. Berlea, S. Cotofana, I. Athanasiu, J. Glossner, and S. Vassiliadis, **"Garbage Collection for the Delft-Java Processor"**, *"8th IASTED International Conference on Applied Informatics (AI-2000)"*, pp. 232-238, Innsbruck, Austria, February, 2000.

5. D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy, **"A New Approach to DSP Intrinsic Functions"**, *Proceedings of the Hawaii International Conference on System Sciences*, pp. 908-918, Hawaii, January, 2000.

6. D. Batten, S. Jinturkar, J. Glossner, M. Schulte, R. Peri, and P. D'Arcy, **"Interaction Between Optimizations and a New Type of DSP Intrinsic Function"**, *Proceeding of the International Conference on Signal Processing Applications and Technology (ICSPAT '99)*, Orlando, Florida, November, 1999.

7. J. Glossner and S. Vassiliadis, **"Delft-Java Dynamic Translation"**, *Proceedings of the 25th EUROMICRO conference (EUROMICRO '99)*, Vol. 1, pp., Milan, Italy, September 8-10, 1999.

8. N. Yadav, M. Schulte, and J. Glossner, **"Parallel Saturating Fractional Arithmetic Units"**, *Proceedings of the 9th Great Lakes Symposium on VLSI*, pp. 172-179, Ann Arbor, Michigan, March 4-6, 1999.

9. D. Parson, P. Beatty, J. Glossner, and B. Schlieder, **"A Framework for Simulating Heterogeneous Virtual Processors"**, *Proceedings of the 1999 Simulation Symposium.*

10. S. Jinturkar, J. Thilo, J. Glossner, P. D'Arcy, and S. Vassiliadis, **"Profile Directed Compilation in DSP Applications"**, *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT '98)*, September, 1998.

11. J. Glossner and S. Vassiliadis, **"Delft-Java Link Translation Buffer"**, *Proceedings of the 24th EUROMICRO conference (EUROMICRO 98)*, Vol. 1, pp. 221-228, Vasteras, Sweden, August 25-27, 1998.

12. J. Glossner, J. Thilo, and S. Vassiliadis, **"Java Signal Processing:FFT's with bytecodes"**, *Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing*, February 28 and March 1, 1998, Stanford University, Palo Alto, California.

13. C.J. Glossner and S. Vassiliadis, **"The Delft-Java Engine: An Introduction"**, *Lecture Notes In Computer Science. Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pp. 766-770, Universitaet Passau, Passau, Germany, August 26-29, 1997, Springer-Verlag.

14. C.J. Glossner, G.G. Pechanek, S. Vassiliadis, and J. Landon, **”High-Performance Parallel FFT Algorithms on M.f.a.s.t. Using Tensor Algebra.”** *Proceedings of the Signal Processing Applications Conference at DSPx’96*, March 11-14, 1996, pp. 529-536, San Jose Convention Center, San Jose, California..

15. G.G. Pechanek, C.J. Glossner, Z. Li, C.H.L. Moller, and S. Vassiliadis, **”Tensor Product FFT’s on M.f.a.s.t.: A Highly Parallel Single Chip DSP.”** In *Proceedings of DSP95 - Digital Signal Processing and Its Applications*, eighth paper, pages 1-10, October 1995, Paris, France.

16. G.G. Pechanek, C.W. Kurak, C.J. Glossner, C.H.L. Moller, and S.J. Walsh,**”Mfast: A Highly Parallel Single Chip DSP With a 2D IDCT Example.”** In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, pages 69-72, October 1995, Boston.

17. G.G. Pechanek, M. Stojancic, S. Vassiliadis, and C.J. Glossner, **”Mfast: A Single Chip Highly Parallel Architecture for Image Processing.”** In *Proceedings IEEE International Conference on Image Processing (ICIP)*, volume 1, pages 1375-1379, Arlington, Virginia, October 1995.

18. G.G. Pechanek, C.J. Glossner, W.F. Lawless, D.H. McCabe, C.H.L. Moller, and S.J.Walsh, **”A Machine Organization and Architecture for Highly Parallel, Scalable, Single Chip DSPs,”** In *Proceedings of the 1995 DSPx Technical Program Conference & Exhibition*, pp. 42-50, 5/15-18/95, San Jose, California.

19. J.C. Le Garrec, C. Marion, J.P. Mifsud, S. Nicot, B. Rousseau, R. Saura, T. Tatry, C.J. Glossner, and R.D. Kilmoyer, **”Design and Application Trade-offs Between High-Density and High-Speed ASICs.”** *IEEE International Conference on Computer Design: VLSI In Computers & Processors*, 4 pages, September, 1990, Boston, MA.

Invited Talks

1. J. Glossner, D. Routenberg, E. Hokenek, M. Mougill, M. Schulte, P. Balzola, and S. Vassiliadis, **”Towards a Very High Bandwidth Wireless Battery Powered Device”**, *In Proceedings of the 2001 IEEE Computer Society Workshop on VLSI (WVLSI)*, Orlando, Florida, April 19-20, 2001.

2. J. Glossner, **”The challenges in Microelectronics associated with broadband access”**, *Semiconductor Industry Association of Japan (SIAJ)*, Tokyo, Japan, November, 2000.

3. J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware, **”Trends In Compilable DSP Architecture”**, *Proceedings of SIPS-2000*, Lafayette, LA, October, 2000.

4. J. Glossner, **”Compilable DSP Architecture”**, *University of Pennsylvania Computer Science Department*, March 16, 1998, Philadelphia, Pa.

5. J. Glossner, **”DSP Architecture”**, *Lehigh University Electrical and Computer Engineering Department*, October 28, 1997, Bethlehem, Pa.

Technical Reports

1. D. Batten, C.J. Glossner, N. Yadav, P. D’Arcy, S. Jinturkar, and K. Wires, **Methods of Cluster Interconnection for Reducing Register Port Pressure**, *Bell Labs* ITD-98-34825D, July, 1998, pp. 1-16. A method for reducing register file ports on wide issue clustered architectures

2. C.J. Glossner, S. Jinturkar, P. D’Arcy, and K. Wires, **”Method and Apparatus for Multiple Processor Machine View Execution”**, *Bell Labs* ITD-98-34500A, May, 1998, pp. 1-10. A method that allows a processor to execute code written for multiple instruction set architectures on one datapath implementation.

3. D. Batten, C. J. Glossner, P. D’Arcy, S. Jinturkar, and J. Thilo, **”Impatient Execution: A Method for Virtual Single-Cycle Execution in Pipelined Processors”**, *Bell Labs* ITD-98-34497X, May, 1998, pp. 1-13. A register locking mechanism for pipelined processors that avoids pipeline hazards in variable latency execution units.

4. C.J. Glossner, K. Wires, D. Batten, S. Jinturkar, J. Thilo, and P. D’Arcy, **”Compiler-Controlled Dynamic Dispatch: A Method for Reducing Stalls in Pipelined Processors”**, *Bell Labs* ITD-98-34498Y, May, 1998, pp. 1-10. A method is proposed to dynamically regulate the data and predication dependencies in a pipelined processor with multiple execution units.

5. C.J. Glossner, S. Jinturkar, P. D'Arcy, and K. Wires, **"Compiler Controlled Dynamic Scheduling"**, *Bell Labs* ITD-98-34499Z, May, 1998, pp. 1-13. A method of dynamically storing multiple instruction dependencies which the compiler has pre-specified.

6. Z. Li, G.G. Pechanek, C.J. Glossner, and C.H.L. Moller, **"Design and Implementation of FFT Algorithms for M.F.A.S.T. Using Tensor Products"**. *IBM* TR 29.2126, July 1996, pp. 1-19.

7. C.J. Glossner, **"Improved Performance Product 25 (IPP25) Technical Report"**. *IBM* TR-19.0874. pages 1-35. This technical report details the process changes innovated to provide a 25% performance increase to an existing process with minimal work and no circuit redesign.

Patents

1. Dean Batten, Paul G. D'Arcy, C. John Glossner, Sanjay Jinturkar, and Kent E. Wires, **"File replication methods and apparatus for reducing port pressure in a clustered processor."** U.S. Patent 6230251. *Issued for Agere Systems.* The invention provides techniques for reducing the port pressure of a clustered processor. In an illustrative embodiment, the processor includes multiple clusters of execution units, with each of the clusters having a portion of a register file and a portion of a predicate file associated therewith, such that a given cluster is permitted to write to and read from its associated portions of the register and predicate files. A replication technique in accordance with the invention reduces port pressure by replicating, e.g., a register lock file and a predicate lock file of the processor for each of the clusters. The replicated files vary depending upon whether the technique is implemented with a write-only interconnection or a read-only interconnection.

2. G.G. Pechanek, L.D.Larsen, C.J. Glossner, S. Vassiliadis, **"Distributed Processing Array with Component Processors Performing Customized Interpretation of Instructions."** U.S. Patent 6128720. *Issued for IBM on October 3, 2000.* A multi-processor array organization is dynamically configured by the inclusion of a configuration topology field in instructions broadcast to the processors in the array. Each of the processors in the array is capable of performing a customized data selection and storage, instruction execution, and result destination selection, by uniquely interpreting a broadcast instruction by using the identity of the processor executing the instruction. In this manner, processing elements

in a large multi-processing array can be dynamically reconfigured and have their operations customized for each processor using broadcast instructions.

3. C. John Glossner, Paul Gerard D'Arcy, Sanjay Jinturkar, and Stamatis Vassiliadis. **"Multiple Machine View Execution in a Computer System"** U.S. Patent 6079010. *Issued for Lucent Technologies on June 20th 2000.* A computer system supporting N different machine views, where N.gtoreq.2, includes a memory for storing instructions, a number of execution units for processing data based on execution controls, and N different decoders for generating the execution controls using instructions retrieved from the memory. Each of the N decoders is operative to decode retrieved instructions in accordance with one of the N machine views. A particular one of the N decoders to be used to decode a given retrieved instruction may be selected by a program running on the system. In one embodiment, the decoders for the N machine views are implemented as N separate decoders, and a multiplexer is used to select the output of one of the N decoders for connection to one or more of the execution units. In another embodiment, a set of reconfigurable hardware is dynamically reprogrammed to implement one or more of the N decoders as directed by the program running on the system.

4. G.G.Pechanek, L.D. Larsen, C.J. Glossner, S. Vassiliadis, D.H. McCabe, **"Selective Processing and Routing of Results among Processors Controlled by Decoding Instructions using Mask Value Derived from Instruction Tag and Processor Identifier"**. U.S. Patent 5682491. *Issued for IBM on Oct. 28, 1997.* describing an array of VLIW processors and dynamic reconfiguration of interconnections. An array processor topology reconfiguration system and method enables processor elements in an array to dynamically reconfigure their mutual interconnection for the exchange of arithmetic results between the processors. Each processor element includes an interconnection switch which is controlled by an instruction decoder in the processor. Instructions are broadcast to all of the processors in the array. The instructions are uniquely interpreted at each respective processor in the array, depending upon the processor identity.

5. G.G. Pechanek, S. Vassiliadis, L.D. Larsen, and C.J. Glossner, **"Array Processor Communication Architecture with Broadcast Processor Instructions."** U.S. Patent 5659785. I*ssued for IBM on Aug. 19, 1997.* Communications Architecture of the Mfast processor. A plurality of

processor elements (PEs) are connected in a duster by a common instruction bus to a sequencing control unit with its associated instruction memory. Each PE has data buses connected to at least its four nearest PE neighbors, referred to as its North, South, East and West PE neighbors. Each PE also has a general purpose register file containing several operand registers. A common instruction is fetched from the instruction memory by the sequencing control unit and broadcast over the instruction bus to each PE in the cluster. The instruction includes an upcode value that controls the arithmetic or logical operation performed by an execution unit in the PE on one or more operands in the register file. A switch is included in each PE to interconnect it with a first PE neighbor as the destination to which the result from the execution unit is sent. The broadcast instruction includes a destination field that controls the switch in the PE, to dynamically select the destination neighbor PE to which the result is sent. An 8x8 pixel-array 2-dimensional Discrete Cosine Transform (DCT) example is presented, using this invention in a folded array topology, which executes in 18-cycles.

6. G.G. Pechanek, C.J. Glossner, L. D. Larsen, S. Vassiliadis, **”Parallel Processing System and Method Using Surrogate Instructions.”** U.S. Patent 5649135. *Issued for IBM on July 15, 1997.* Parallel VLIW Control for an Array of VLIW processors. A parallel processing system and method is disclosed, which provides an improved instruction distribution mechanism for a parallel processing array. The invention broadcasts a basic instruction to each of a plurality of processor elements. Each processor element decodes the same instruction by combining it with a unique offset value stored in each respective processor element, to produce a derived instruction that is unique to the processor element. A first type of basic instruction results in the processor element performing a logical or control operation. A second type of basic instruction results in the generation of a pointer address. The pointer address has a unique address value because it results from combining the basic instruction with the unique offset value stored at the processor element. The pointer address is used to access an alternative instruction from an alternative instruction storage, for execution in the processor element. In this invention, it is possible to generate a one-to-one mapping of the surrogate provided address to a group of VLIWs that become individually selectable within each PE through the use of a PE address modifying mechanism.

Biography for C. John Glossner

John Glossner is currently CTO and Executive Vice President of Engineering at Sandbridge Technologies, Inc. In his position with Sandbridge, John directs the engineering group working on JAVA-based DSPs for cellular systems in broadband communications. Previously, John managed groups at IBM Research whose efforts were directed at DSP designs and high-performance transmission systems. During this time, he also served as Access Aggregation Business Development Manager for IBM Microelectronics. Prior to joining IBM Research, John worked for Lucent Microelectronics where he was chief architect for wireless DSPs. As a founding member of Starcore, John managed the software and compiler efforts. John started his career with IBM Microelectronics in Burlington, Vermont, working on ASIC designs after graduating with a B.S.E.E. degree from Penn State. While continuing to work full time, John earned an M.S.E.E. and an M.S. in Engineering Management from the National Technological University. After moving to IBM in RTP, North Carolina, John began working on DSPs in the IBM Mwave group. He began Ph.D. studies at the University of North Carolina at Chapel Hill which subsequently led to this thesis work completed at the Delft University of Technology.

Bibliography

[1] James Gosling, Bill Joy, and Guy Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[2] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, Mountain View, California, October 1995. Available from ftp.javasoft.com/docs.

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[4] M. Gagnaire. An Overview of Broadband Access Technologies. In *Proceedings of the IEEE*, volume 85, pages 1958–1972, December 1997.

[5] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, pages 51–59, August 1998.

[6] Junko Yoshida. Java chip vendors set for cellular skirmish. EE Times, January 30 2001.

[7] P. Lapley. *DSP Processor Fundamentals*. IEEE press, New York, 1997.

[8] M. Saghir, P. Chow, and C. G. Lee. Towards Better DSP Architecture and Compilers. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 658–664, October 1994.

[9] Texas Instruments. TMS320C54x DSP Reference Set. Volume 1: CPU and Perhipherals. Technical Report SPRU131E, Texas Instruments, June 1998.

[10] Jeff Bier. DSP16xxx Targets Communictaions Apps. *Microprocessor Report*, 11(12), 1997.

[11] G. Ungerboeck, D. Maiwald, H. P. Kaeser, P. R. Chevillat, and J. P. Beraud. Architecture of a Digital Signal Processor. *IBM Journal of Research and Development*, 29(2), 1985.

[12] N. L. Bernbaum, B. Blaner, D. E. Carmon, J. K. D'Addio, F. E. Grieco, A. M. Jacoutot, M. A. Locker, B. Marshall, D. W. Milton, C. R. Ogilvie, P. M. Schanely, P. C. Stabler, and M. Turcotte. The IBM Mwave 3780i DSP. In *Proceedings of the 1996 International Conference on Signal Processing Applications and Technology (ICSPAT '96)*, pages 1287–1291, Boston, MA, October 1996.

[13] Tom R. Halfhill. TI Cores Accelerate DSP Arms Race. *Microprocessor Report*, March 6 2000.

[14] J. Eyre and J. Bier. Carmel Enables Customizable DSP. *Microprocessor Report*, 12(17), December 1998.

[15] LSI Corporation. *LSI402Z Digital Signal Processor*. LSI Corporation, r20012 edition, 1999.

[16] Gerald G. Pechanek, C. John Glossner, William F. Lawless, Daniel H. McCabe, Chris H. L. Moller, and Steven J. Walsh. A Machine Organization and Architecture for Highly Parallel, Scalable, Single Chip DSPs. In *Proceedings of the 1995 DSPx Technical Program Conference and Exhibition*, pages 42–50, San Jose, California, May 1995.

[17] Gerald G. Pechanek, Mihilo Stojancic, Stamatis Vassiliadis, and C. John Glossner. M.F.A.S.T.: A Single Chip, Highly Parallel Image Processing Architecture. In *Proceedings IEEE International Conference on Image Processing*, volume I, pages 1375–1379, Arlington, Virginia, October 1995. IEEE Press.

[18] Gerald G. Pechanek, Charles W. Kurak, C. John Glossner, Chris H. L. Moller, and Steven J. Walsh. M.F.A.S.T.: A Highly Parallel Single Chip DSP with a 2D IDCT Example. In *Proceeding of the International Conference on Signal Processing Applications and Technology*, pages 69–72, Boston, Mass., October 1995.

[19] Gerald G. Pechanek, C. John Glossner, Zhiyong Li, Chris H. L. Moller, and Stamatis Vassiliadis. Tensor Product FFT's on M.*F.A.S.T.: A Highly Parallel Single Chip DSP*. In *Proceedings of DSP 95 - Digital Signal Processing and Its Applications*, Paris, France, October 1995.

[20] B. Case. Philips hopes to displace DSPs with VLIW. *Microprocessor Report*, pages 12–15, December 1997.

[21] C. P. Feigel. TI Introduces Four-Processor DSP Chip. *Microprocessor Report*, 8(4), March 28 1994.

[22] Dave Epstein. Chromatic Raises the Multimedia Bar. *Microprocessor Report*, 9(14), October 28 1995.

[23] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 42–50, August 1996.

[24] H. Nguyen and L. K. John. Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology. In *Proceedings of the International Conference on Supercomputing*, pages 11–20, 1999.

[25] J. C. Bier, A. Shoham, H. Hakkarainen, O. Wolf, G. Blalock, and Philip D. Lapsley. *DSP on General-Purpose Processors: Performance, Architecture, Pitfalls*. Berkeley Design Technology, Inc., 1997.

[26] O. Wolf and J. Bier. StarCore Launches First Architecture. *Microprocessor Report*, 12(14), October 1998.

[27] O. Wolf and J. Bier. TigerSHARC Sinks Teeth Into VLIW. *Microprocessor Report*, 12(16), December 1998.

[28] J. Fridman and Z. Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1):66–76, January 2000.

[29] J. Turley and H. Hakkarainen. TI's New C6x Screams at 1,600 MIPS. *Microprocessor Report*, 11(2), 1997.

[30] David Strube. High perfromance DSP technology brings new features to digital systems. In *Electronic Product Design*, pages 23–26, October 1999.

[31] Gerald G. Pechanek, Stamatis Vassiliadis, and Nikos Pitsianis. ManArray processor interconnection network: an introduction. In *Euro-Par '99 Parallel Processing Proceedings. (Lecture notes in computer science)*, pages 761–765, Toulouse, France, August/September 1999. Springer, Berlin.

[32] Gerald G. Pechanek and Stamatis Vassiliadis. The ManArray Embedded Processor Architecture. In *Proceedings of the 26-th Euromicro Conference: Informatics: inventing the future*, volume I, pages 348–355, Maastrict, The Netherlands, September 5-7 2000.

[33] Bryan Ackland and Paul D'Arcy. A New Generation of DSP Architectures. In *Proceedings of the 1999 Custom Integrated Circuits Conference*, pages 531–536, 1999.

[34] B. Ackland and et. al. A Single-Chip 1.6 Billion 16-b MAC/s Multiprocessor DSP. In *Proceedings of the Custom Integrated Circuits Conference*, pages 537–540, 1999.

[35] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen mei W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceeding of the 29th Annual Internation Symposium on Microarchitecture (MICRO-29)*, pages 90–97, Los Alamitos, CA, USA, December 2-4 1996. IEEE Computer Society Press.

[36] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guie-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceeding of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation (PLDI'98)*, volume 33, pages 280–290. Association for Computing Machinery, May 1998.

[37] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. JIT vs. Offline Compilers: Limits and Benefits of Bytecode Compilation. Technical Report 1063, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, December 1996. http://www.irisa.fr.

[38] Sun Microsystems. The Java Hotspot Performance Engine Architecture. Sun Microsystems, 1999. http://java.sun.com/ products/ hotspot/ whitepaper.html.

[39] Kemal Ebcioglu, Eric R. Altman, and Erdem Hokenek. A Java ILP Machine Based on Fast Dynamic Compilation. In *IEEE MASCOTS International Workshop on Security and Efficiency Aspects of Java*, Eilat, Israel, January 9-10 1997. IEEE Computer Society Press.

[40] Richard Agnews. Sun's Java HotSpot Performance Engine a Speed Demon. Sun Microsystems, April 1999. http://java.sun.com/ features/ 1999/ 04/ jess_reports/ hotspot.announce.html.

[41] Michal Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 9(4):1063–1073, November 1997.

[42] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java For Applications - A Way Ahead of Time (WAT) Compiler. In *Proceedings Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, 1997.

[43] Hewlett Packard. HP Turbo Chai Release 2.0. Hewlett-Packard, May 1999. http://www.hp.com/emso/products/turbochai/TchaiPDF.pdf.

[44] John Glossner, Jesse Thilo, and Stamatis Vassiliadis. Java Signal Processing: FFT's with bytecodes. In *Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February 28 and March 1 1998.

[45] John Glossner, Jesse Thilo, and Stamatis Vassiliadis. Java Signal Processing: FFT's with bytecodes. *Journal of Concurrency and Experience*, 10(11-13):1173–1178, 1998.

[46] Cygnus. Gcj compiler, 1999.

[47] Sun Microelectronics. picoJava I Microprocessor Core Architecture. Technical Report WPR-0014-01, Sun Microsystems, Mountain View, California, November 1996. Available from http://www.sun.com/ sparc/ whitepapers/ wpr-0014-01.

[48] Marc Tremblay and Micahel O'Connor. picoJava: A Hardware Implementation of the Java Virtual Machine. In *Hotchips Presentation*, 1996.

[49] Harlan McGhan and Mike O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[50] L. C. Chang, L. R. Ton, M. F. Kao, and C. P. Chung. Stack operations folding in Java processors. *IEE Proceedings - Computers and Digital Techniques*, 145(5):333–343, September 1998.

[51] Lee-Ren Ton, Lung-Chung Chang, Min-Fu Kao, Han-Min Tseng, Shi-Sheng Shang, Ruey-Liang Ma, Dze-Chaung Wang, and Chung-Ping Chung. Instruction Folding in Java Processor. In *1997 International Conference on Parallel and Distributed Systems*, pages 138–143, Seoul, Korea, December 12-13 1997. IEEE Computer Society Press.

[52] Sun Microelectronics. Sun Microelectronic's picoJava I Posts Outstanding Performance. Technical Report WPR-0015-01, Sun Microsystems, Mountain View, California, November 1996. Available from http://www.sun.com/ sparc/ whitepapers/ wpr-0015-01.

[53] C. John Glossner and Stamatis Vassiliadis. The Delft-Java Engine: An Introduction. In *Lecture Notes In Computer Science. Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pages 766–770, Passau, Germany, Aug. 26 - 29 1997. Springer-Verlag.

[54] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, II:25–33, 1967.

[55] Wes Munsil and Chia-Jiu Wang. Reducing Stack Usage in Java Bytecode Execution. *Computer Architecture News*, 1(7):7–11, March 1998.

[56] Yamin Li, Sanli Li, Xianzhu Wang, and Wanming Chu. JAViR - Exploiting Instruction Level Parallelism for JAVA Machine by Using Virtual Register. In *The Second European IASTED International Conference on Parallel and Distributed Systems*, Vienna, Austria, July 1-3 1998.

[57] Advancel Logic Corporation. TinyJ Processor Core Product Datasheet. Datasheet, July 1999.

[58] Patriot Scientific Corporation. Psc1000/a microprocessor datasheet. Patriot Scientific, 1997. http://www.ptsc.com/downloads/psc1000/specs/datasheet.pdf.

[59] James Gosling. Java Intermediate Bytecodes. In *ACM SIGPLAN Notices*, pages 111–118, New York, NY, January 1995. Association for Computing Machinery. ACM SIGPLAN Workshop on Intermediate Representations (IR95).

[60] William Stallings. *Computer Organization and Architecture*. Macmillan Publishing Company, New York, 1987.

[61] Tim Lindholm and Frank Yellin. Inside the java virtual machine. *Unix Review*, 15(1):31–39, January 1997. Adapted From [3].

[62] Sun Microsystems. *The Java Virtual Machine Specification*, volume Release 1.0 Beta. Sun Microsystems, Mountain View, California, August 1995.

[63] Gerrit A. Blaauw and Frederick P. Brooks Jr. *Computer Architecture: Concepts and Evolution*. Addison-Wesley, Reading, MA, USA, 1997.

[64] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, 1971.

[65] John Glossner and Stamatis Vassiliadis. Delft-Java Link Translation Buffer. In *Proceedings of the 24th EUROMICRO conference*, volume 1, pages 221–228, Vasteras, Sweden, August 25-27 1998.

[66] Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press - A Prentice Hall Title, Mountain View, California, 1996.

[67] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. Sunsoft Press - A Prentice Hall Title, Mountain View, California, 1996.

[68] Peter Wayner. Sun gambles on java chips. *Byte*, 21(11):79–85, November 1996.

[69] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. SCISM: A Scalable Compound Instruction Set Machine. *IBM Journal of Research and Development*, 38(1):59–78, January 1994.

[70] James Philips and Stamatis Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3):257–268, March 1994.

[71] Brian Case. Implementing the java virtual machine. *Microprocessor Report*, 10(4):12–17, March 25 1996.

[72] Stamatis Vassiliadis, James Phillips, and Bart Blanar. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[73] A. Berlea, S. Cotofana, I. Athanasiu, J. Glossner, and S. Vassiliadis. Garbage Collection for the Delft-Java Processor. In *8th IASTED International Conference on Applied Informatics (AI-2000)*, pages 232–238, Innsbruck, Austria, February 2000.

[74] Ole Agesen and David Detlefs. Finding references in java stacks. In *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, pages 766–770, Atlanta, GA, USA, October 1997.

[75] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. JOHN WILEY & SONS, NY 10158-0012, USA, 1996.

[76] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processor. In *International Symposium on Computer Architecture*, volume 23, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[77] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6. Association for Computing Machinery, 1990.

[78] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The m-machine multicomputer. Technical report, MIT Artificial Intelligence Laboratory, 1995.

[79] Wayne Yamamoto and Mario Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques. PACT'95*, pages 49–58, 1995.

[80] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, volume 23, pages 392–403, Santa Margherita Ligure, Italy, June 1995.

[81] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings AFIPS National Computer Conference*, pages 483–485, 1967.

[82] Michael J. Flynn. *Computer Architecture: Piplelined and Parallel Processor Design.* Jones and Bartlett, Boston, Mass., 1995.

[83] Marshall C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM*, 15(2):252–264, April 1968.

[84] C. John Glossner, Gerald G. Pechanek, Stamatis Vassiliadis, and Joe Landon. High-Performance Parallel FFT Algorithms on M.f.a.s.t. Using Tensor Algebra. In *Proceedings of the Signal Processing Applications Conference at DSPx'96*, pages 529–536, San Jose Convention Center, San Jose, Ca., March 11-14 1996.

[85] William H. Press, Saul A. Teukolskey, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, 2 edition, 1992.

[86] Paul Wilson. Uniprocessor garbage collection techniques, 1996.