Chapter 6

THE DELFT-JAVA ENGINE

Microarchitecture and Java Acceleration

John Glossner
Sandbridge Technologies, Inc.
White Plains, NY, USA
glossner@sandbridgetech.com

Stamatis Vassiliadis
Delft University of Technology
Delft, The Netherlands
stamatis@cardit.et.tudelft.nl

Abstract This chapter describes the DELFT-JAVA architecture and the mechanisms required to dynamically translate JVM instructions into DELFT-JAVA instructions. 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. The hardware requirements to perform this translation are not excessive when support for Java language constructs are incorporated into the processor's ISA. When combined with superscalar techniques and multiple instruction issue, we remove up to 60% of translated dependencies. When compared with a realizable stack-based implementation, our approach accelerates a Vector Multiply execution by 2.7x when hardware constraints were considered. Because this approach requires minimal additional hardware for Java translation, it is an efficient technique for executing Java bytecode.

Keywords: Java, Computer Architecture, dynamic translation, processor design

1. Introduction

We have designed the DELFT-JAVA processor [GV97]. An important feature of this architecture is that it has been designed to efficiently execute Java

Virtual Machine (JVM) bytecode. The architecture has two logical views: 1) a Java Virtual Machine Instruction Set Architecture(ISA) and 2) a RISC-based ISA. The Java Virtual Machine is a stack-based ISA with support for standard datatypes, synchronization, object-oriented method invocation, arrays, and object allocation [LY99]. An important property of Java bytecode is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code [Gos95]. We utilize this property to dynamically translate Java bytecode into DELFT-JAVA instructions. Because the bytecodes are stored as pure Java instructions, Java programs generated by Java compilers execute on a DELFT-JAVA processor without modification. Programmers who wish to take advantage of other languages that exploit the full capabilities of the DELFT-JAVA processor may do so but require a specific compiler. Some additional architectural features in the DELFT-JAVA processor that are not directly accessible from JVM bytecode include pointer manipulation, multimedia SIMD instructions, unsigned datatypes, and rounding/saturation modes for DSP algorithms.

This chapter is dedicated to describing the organizational techniques the DELFT-JAVA processor uses to accelerate Java bytecode. 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 concurrent multithreaded organization and describe how multiple thread units and multiple instruction issue efficiently accelerate Java program execution. We introduce briefly a method for accelerating Java dynamic method invocation through the use of a Link Translation Buffer (LTB). We next describe how indirect access to the register file provides the basic mechanism required to dynamically translate JVM instructions. We then provide an example of the translation process and describe special hardware features we incorporated to assist translation. We also discuss instructions that are not translated. We then present results on a number of simulated machines and conclude by summarizing our investigation.

2. Concurrent Multithreaded Organization

In this section, we present a concurrent multithreaded organization of the DELFT-JAVA architecture. As shown in Figure 6.1, 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. As depicted by the large rect-

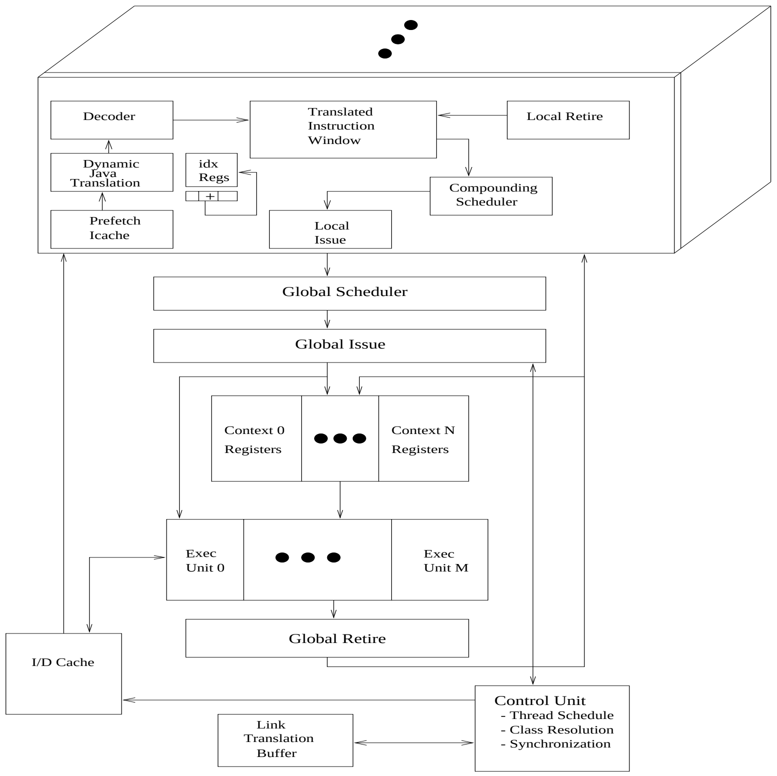

Figure 6.1. DELFT-JAVA concurrent multi-threaded processor organization showing multiple thread units, local and global processor units, thread register files, cache memory, control unit, and Link Translation Buffer (LTB)

angle with ellipsis in Figure 6.1, we define a *context* as a hardware supported thread unit. Each context assumes that the processor's organization incorporates (logically) an instruction cache (the Prefetch Icache), a Dynamic Java Translation unit, a decode unit, a translated instruction window 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 (the I/D block), 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 [LB96, KSS96]. In this chapter, 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 (e.g., common to all contexts) and placed into a global L1 on-chip instruction cache

(I/D Cache). Each context also assumes a (logical) zero level (L0) instruction cache to provide concurrent per context *instruction fetch* capacity (Prefetch Icache). 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 by the Dynamic Java Translation unit. 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* (e.g., local to each context). The instruction window keeps track of issued and pending instructions. The *local instruction scheduler* (Compounding Scheduler Unit) 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 folded [MO98]. The DELFT-JAVA processor may also dynamically build internal compound instructions [VBE94]. Whereas combining and compounding have to do with grouping both independent and dependent instructions for concurrent execution, folding, by contrast, has to do with combining multiple operations into a single operation (e.g., a local variable access and an add operation). 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 [PV94].

All instructions that require access to shared resources must be forwarded to the *global instruction scheduler*. This unit schedules the aggregated instructions from each context. 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 main memory in any order [GJS96]. 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 that 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, most execution units would require bypass circuitry. 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 requirement is removed by providing 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.

■ **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 [LY99]. 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.

■ **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 an unacceptable degradation of a hardware context (e.g., lock request, priority inversion, etc.). 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 free 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 JVM 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*[GV98, Glo01]. 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 invalid data (e.g., when a class is unloaded). The control unit is responsible to remove associations that are no longer valid by issuing a `flushLTB` instruction. A program may also completely disable the LTB.

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 must be 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.

3. Enhancing Performance

Accelerating the JVM 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 [PV94], branch prediction, a link translation buffer [GV98], and standard register files. We selectively describe some of these mechanisms.

■ **Removing Hazards:** A common problem with stack architectures is that the stack may become a bottleneck for exploiting instruction level parallelism. Since the results of operations typically pass through the top of the stack, many interlocks are generated in the translated instruction stream [Cas96]. 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 [VBE94, PV94, VPB93].

■ **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 [Tom67]. 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 [GJS96]. 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. Dynamic Translation

The DELFT-JAVA architecture supports the same basic datatypes as the Java Virtual Machine. We dynamically translate JVM instructions into DELFT-JAVA instructions by providing indirect access into the register file. Figure 6.2 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 spec-

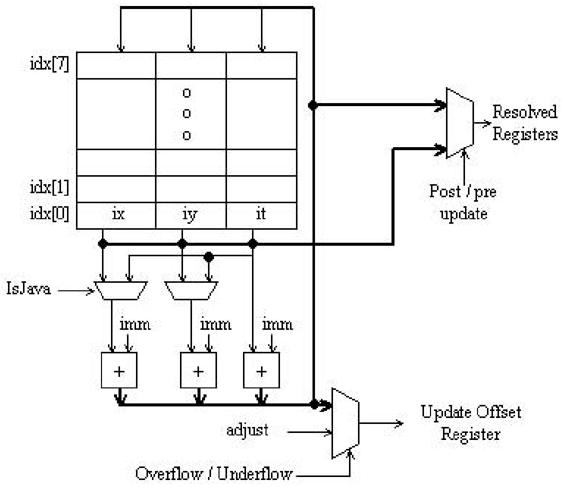

Figure 6.2. Indirect register access mechanism showing indirect memory locations (idx), update adders, underflow/overflow signal, and resolved register address multiplexor

ify 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 that maps the register file into main memory must be adjusted.

As shown in Figure 6.2, 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.

Program 4.1 Indirect instructions

```
add             r2, r0, r1
addi [idx7]     ++it, 2-ix, iy
storei [idx7]   base0 + #3, it+1
```

To illustrate how these operations are performed, consider the code shown 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,

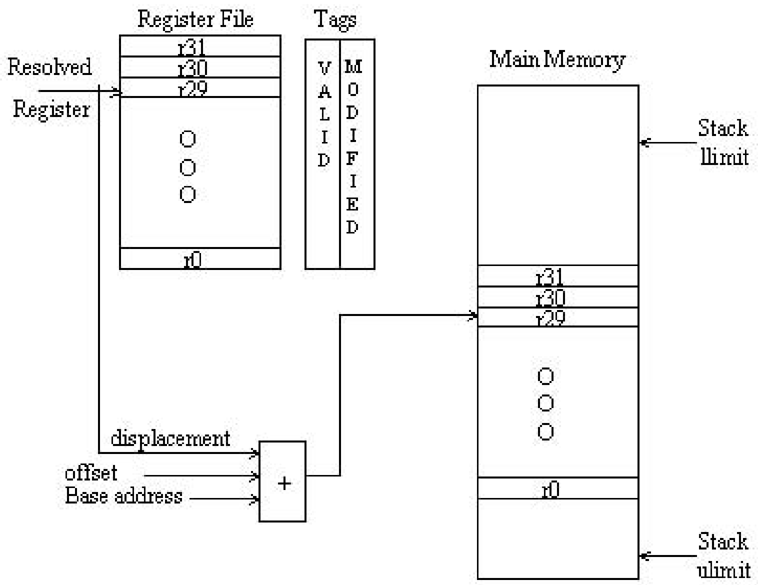

Figure 6.3. Indirect register mapping showing how a resolved register address is mapped to main memory. Tag bits allow the processor to read/write only valid data.

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 JVM byte-codes, the pre/post increment values can be implied from the JVM instruction.

Figure 6.3 shows the indirect mapping translation. The resolved register address from Figure 6.2 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.

Program 4.2 Vector multiply example

```
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};
} } }
```

4.1 Example Translation

In this section we present the translation of a Vector Multiply example. 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 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).

■ **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 of Java bytecodes, a *"branchJVM"* instruction is executed by a DELFT-JAVA processor. As shown in Figure 6.2, 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

Program 4.3 Compiled inner loop bytecode

```
aload_3    ; address of c[0] on heap
iload 5    ; index into c[]
aload_1    ; address of a[0]
iload 5    ; index into a[]
iaload     ; load element from a[index]
aload_2    ; address of b[0]
iload 5    ; index into b[]
iaload     ; load element from b[index]
imul       ; multiply a[i]*b[i]
iastore    ; store it into c[index]
```

Program 4.4 Translation bytecode

```
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
```

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.

Program 4.5 Final DELFT-JAVA instructions

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

■ **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 a 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.

4.2 Hardware Support

In order to perform Java translation, the DELFT-JAVA processor has a number of special registers that 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

Table 6.1. Java Virtual Machine instructions with special support in the DELFT-JAVA processor

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

Java implementation is free to update the main memory in any order [GJS96]. 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 JVM instruction is used. The JVM specification states that 3 opcodes will permanently be reserved for implementation dependent purposes [LY99]. The DELFT-JAVA processor utilizes one of these instructions to transition a context between JVM 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.

4.3 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 6.1 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 JVM 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.

Table 6.2. Processor organization characteristics for various processor models

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

5. Results

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 JVM 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.

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 6.2. 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 above. It also includes multiple inorder 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 measured 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 6.4 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

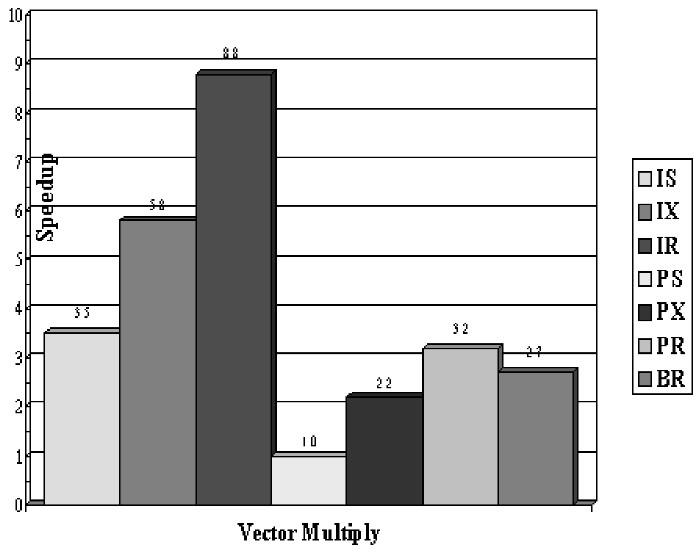

Figure 6.4. Performance results of a vector-multiply routine for various processor models showing speedup normalized to an implementable pipelined stack model

Table 6.3. Processor performance and speedup for various processor models normalized to an implementable pipelined stack model

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

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 6.3 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. 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.

6. Conclusions

We have presented our approach to Java hardware acceleration using *dynamic instruction translation*. In hardware assisted dynamic translation, JVM instructions are translated on-the-fly into the DELFT-JAVA instruction set. This is accomplished through the use of indirect register file access. The additional indirect access hardware and decoder logic requirements to perform this translation are not excessive when support for Java language constructs are 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 and C++. We have shown that our dynamic translation technique (which is a form of register allocation) is useful in removing up to 40% of stack bottlenecks [GV99]. When register renaming is combined with our translation technique, upwards of 60% of stack dependencies can be removed. Our technique effectively converts stack dependencies into pipeline hazards which are later removed from the instruction stream using superscalar techniques. When compared with a realizable stack-based implementation, our approach accelerates a Vector Multiply execution by 2.7x when hardware constraints were considered. Because this approach requires minimal additional hardware for Java translation when incorporated into an out-of-order superscaler machine, it is an efficient technique for executing Java bytecode.