Delft-Java Dynamic Translation

John Glossner[1,2]
[1]IBM Research
Yorktown Heights, NY
glossner@cardit.et.tudelft.nl

Stamatis Vassiliadis[2]
[2]Delft University of Technology
Electrical Engineering Department
Mekelweg 4, 2628 CD Delft, The Netherlands
stamatis@cardit.et.tudelft.nl

Abstract

This paper describes the DELFT-JAVA *processor 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. 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 3.2x for out-of-order execution with register reanaming and 2.7x when hardware constraints were considered. In addition, for translated instruction streams, we realized a 50% performance improvement for out-of-order execution when compared with in-order execution.*

1 Introduction

We have designed the DELFT-JAVA processor[2]. An important feature of this architecture is that it has been designed to efficiently execute JAVA Virtual Machine (JVM) bytecodes. The architecture has two logical views: 1) a JVM Instruction Set Architecture(ISA) and 2) a RISC-based ISA. The JVM is a stack-based ISA with support for standard datatypes, synchronization, object-oriented method invocation, arrays, and object allocation[7]. An important property of JAVA bytecodes is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code[4]. We utilize this property to dynamically translate JAVA bytecodes 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 which 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 which are not directly accessible from JVM bytecode include pointer manipulation, Multimedia SIMD instructions, unsigned datatypes, and rounding/saturation modes for DSP algorithms.

In Section 2 we give an architectural perspective of the JAVA Virtual Machine. In Section 3 we describe how JVM instructions are dynamically translated into DELFT-JAVA instructions including hardware support for executing dependencies. In Section 4 we present the results of translated bytecodes. In Section 5 we describe other related work. Finally, in Section 6 we summarize our findings and present conclusions.

2 Java Virtual Machine Architecture

The JVM is a stack-based Instruction Set Architecture designed to quickly transport programs across the Internet and allow register poor processor architectures to efficiently execute JAVA bytecodes. Instructions are not confined to a fixed length however all of the opcodes in the JVM are 8-bits[7]. This allows for efficient decoding of instructions while not requiring all instructions to be 32-bits or longer.

2.1 Memory Spaces

There are a number of storage spaces defined in the JVM[7]. The *Heap* is an unbounded run-time allocated space where all dynamically allocated objects are placed. There is one heap and it is shared among all threads. The heap is required to be garbage collected. The JVM specification does not require the heap to be a fixed size. If the physical memory capacity of the heap is exceeded, an `OutOfMemoryError` is thrown.

The *Method Area* is the location where the bytecode text is loaded. It 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 JVM specification, it may be of fixed or variable size. The memory area is not required to be contiguous. It is restricted to 2^{16} bytes per method. If enough memory can not be allocated, an `OutOfMemoryError` is thrown.

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 and has a maximum size of 2^{16} entries per class. 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 `OutOfMemoryError` is thrown. Each index into the Constant Pool references a variable length structure.

2.2 Working Store

The primary JVM working store is a stack. It consists of 32-bit words placed as a variable-length, variable-location segment in memory[7]. The *Operand Stack* is logically part of a *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 64-bit datatype 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. Currently the size of the operand stack is restricted to 2^{16} locations due to restrictions on the class file.

The *Local Variables* space is 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 and is placed as a fixed-length, variable-location segment in memory. A 64-bit datatype is considered to take two local variable locations. The local variables hold the formal parameters for the method and partial results during a computation.

2.3 JVM Datatypes

Operations in the JVM are strongly typed. Since there are only 256 opcodes available, this results in the trade-off that nearly all arithmetic operations are performed as integers or IEEE-754 floating point. An interesting property of the JVM is that integer arithmetic operations do not indicate overflow or underflow [7]. There are also load and store instructions which move values from memory spaces to and from the operand stack. In addition to standard operations, there is direct support for method invocation, synchronization, exceptions, and arrays. There are also two variable length instructions - `tableswitch` and `lookupswitch`.

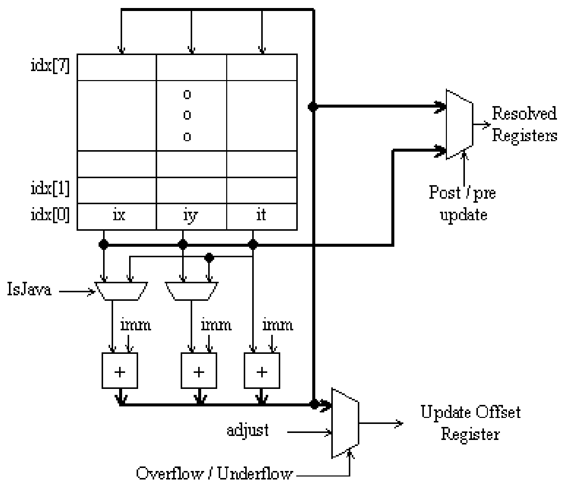

Figure 1. Indirect Register Access

3 Dynamic Translation

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

To illustrate how these operations are performed, consider the following examples: 1) the standard RISC instruction `add r2,r0,r1`. The add mnemonic specifies the operation, r2 is the destination (target) register. Registers r0 and r1 are the source operands. When no type is ex-

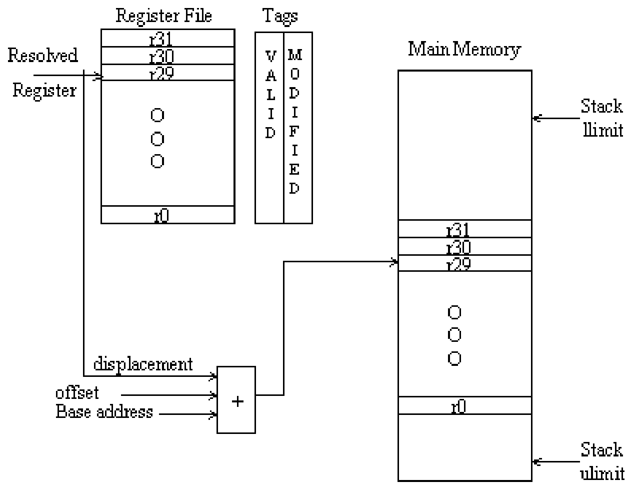

Figure 2. Indirect Register Mapping

plicitly specified, a $w32$ (signed integer 32-bits) type is implied. 2) the indirect instruction `addi [idx7] ++it, 2-ix, iy` 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-incremented with the contents of idx[7][ix] to determine the source operand register. 3) The instruction `storei [idx7] base0 + #3, it+1` specifies that an indirect memory store operation is performed. The target operand is a memory location addressed by an architected base register base0 with an immediate displacement of 3. To calculate the source operand, the value contained in idx[7][it] is used. Since it+1 contains the +1 on the right hand side of it, it implies that idx[7][it] is post-incremented by 1. For JVM bytecodes, the pre/post increment values can be implied from the JVM instruction. Additionally, as shown in Figure 1, the index *"it"* is used as the offset for all operands when the DELFT-JAVA processor is in JAVA translation mode.

Figure 2 shows the indirect mapping translation. The resolved register address from Figure 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.

3.1 Example Translation

We now describe the translation of a vector multiply program. A rudimentary JAVA program reads an element of a vector from an 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. When compiled with -O optimization using Sun's Java JDK 1.1, the inner loop bytecode (e.g. c[i]=a[i]*b[i}) produced 10 instructions. To be able to load a single element from an array, the address of the array is pushed onto the stack followed by the index to load. Prior to entering the inner loop each array was allocated on the heap. As a result of executing the instruction *"newarray int"*, the heap address where the integer array was allocated is returned on the stack. This address is immediately stored into a Local Variables location (e.g. LV[1], LV[2], and LV[3] for a[], b[], and c[] respectively).

Program 1 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

Program 1 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 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 1 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.

Program 2 Final DELFT-JAVAInstructions.

	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

Program 2 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.

3.2 Translated Instructions

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 DELFT-JAVA 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[5]. 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[7]. 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 the DELFT-JAVA architecture can take advantage of DELFT-JAVA 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 1. Instructions with Special Support.

3.3 Non-translated Instructions

Primarily, we dynamically translate arithmetic and data movement instructions. In addition to the translation process, the DELFT-JAVA 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 1 have special support in the DELFT-JAVA 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.

3.4 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 [11], branch prediction, a link translation buffer [3], and standard register files. We selectively describe some of these mechanisms.

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. 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[11]. After translation the instructions are placed in an instruction window. Superscalar techniques are used to extract instruction level parallelism from the instruction stream. When combined with register renaming, improved levels of parallelism may be achieved.

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 2. Model Characteristics

4 Results

We describe seven machine models and report on the relative performance of these models. A summary of the machine characteristics is shown in Table 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 in Section 3. 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 and maintained a four cycle latency for each memory space.

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 3. Machine Performance

Table 3 shows the relative performance of each of the models and a summary of instructions issued, peak issue rate, and overall speedup. 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.

5 Related Work

Hardware approaches to improving JAVA performance have been proposed. Sun's *picoJava* implementation directly executes the JVM Instruction Set Architecture but incorporates other facilities that improve the system level aspects of JAVA program execution[9]. The picoJava chip is a stack-based implementation with a register-based stack cache. Support for garbage collection, instruction optimization, method invocation, and synchronization is provided. Sun states that this provides up to 5x performance improvement over JIT compilers.

Others have also looked at similar methods of removing stack bottlenecks. Dynamic scheduling using the Tomasulo algorithm is described in [10]. We first proposed our mechanism in 1997[2]. Here we extend our previous work to quantify the performance effects of our approach. Both Sun[9] and Chang[1] describe a stack folding technique. Stack folding is the ability to detect some instructions with true data dependencies in the instruction stream and execute them as a single compound instruction. We described a similar technique using collapsing arithmetic units[2, 11].

More recently, a scheme called Virtual Registers was introduced[6]. This scheme is similar to the way in which we handle register references. Their method allows arithmetic instructions to get source operands from a virtual register which may reference stack operands below the top of the stack. As with the DELFT-JAVA processor, this allows them to issue multiple instructions in parallel. They have shown an effective IPC of 2.89 to 4.01. Our technique first published in 1997 gives results consistent with their model.

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. The hardware 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. 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. We note that the translation of JAVA bytecodes produces a large amount of memory operations. Fortunately, many of these are to independent memory spaces or local data which may be efficiently stored in local registers or buffers. Register renaming along with out-of-order execution is particularly important for balancing the scheduling of Loads and Stores with arithmetic operations.

Our current research is looking at combining the stack offset update logic with the out-of-order reorder buffer. In addition, compiler techniques such as software pipelining which operate on machine independent JVM bytecodes may allow more efficient use of higher-order interlock collapsing ALUs (e.g. 4-1 or 5-1 ALUs). It is also possible using register renaming to allow more physical registers than architected DELFT-JAVA registers. Some studies have shown that up to 98% of typical programs use less than 32 stack entries[8]. Therefore the addition of extra physical registers may not be warranted. Other more recent techniques such as value prediction may also be important in JVM execution. This is particularly true of Local Variable values which often return the address of an object allocated on the heap.

References

[1] 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.

[2] C. J. Glossner and S. 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.

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

[4] J. 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).

[5] J. Gosling, B. Joy, and G. Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[6] Y. Li, S. Li, X. Wang, and W. 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.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[8] J. Matthews, editor. *FORTH Applications in Engineering and Industry*. Ellis Horwood Series in Computers and Their Applications. John Wiley & Sons, New York, Chichester, Brisbane, Toronto, Singapore, 1989.

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

[10] W. Munsil and C.-J. Wang. Reducing Stack Usage in Java Bytecode Execution. *Computer Architecture News*, 1(7):7–11, March 1998.

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