The DELFT-JAVA Engine: An Introduction

C. John Glossner[1,2] and Stamatis Vassiliadis[2]

[1] Lucent / Bell Labs, Allentown, Pa.
[2] Delft University of Technology, Department of Electrical Engineering
Delft, The Netherlands
(glossner,stamatis)@einstein.et.tudelft.nl

Abstract. In this paper we introduce the DELFT-JAVA multithreaded processor architecture and organization. The proposed architecture provides direct translation capability from the JAVA Virtual Machine instruction set into the DELFT-JAVA instruction set. The instruction set is a 32-bit RISC instruction set architecture with support for multiple concurrent threads and JAVA specific constructs. The parallelism is extracted transparently to the programmer. Except for kernel programs, programmers need only be concerned with the semantics of the JAVA programming language. In addition, programmers who desire to take greater advantage of parallelism can execute privileged instructions which provide additional capabilities for Multimedia and DSP processing.

1 Introduction

The JAVA language provides processor architects with opportunities for exploiting Instruction Level Parallelism (ILP). Rather than requiring the processor to extract all ILP from a single executing thread, the JAVA language intrinsically supports programmer specification of parallelism through threads. Our goal in the DELFT-JAVA architecture is to extract maximal parallelism as defined by the JAVA language without burdening the programmer to specify any additional parallelism that is not inherent in the language constructs. At the highest level, a programmer of a DELFT-JAVA processor views it as a JAVA Virtual Machine (JVM).

Using RISC architectural concepts, we present a concurrent multithreaded architecture and organization that fully utilizes the JAVA programming paradigm. Furthermore, the architecture allows mechanisms for increasing single thread performance by allowing a single thread to issue multiple instructions per cycle. The architecture is scalable in the number of concurrent threads that can be supported and in the number of execution units that can be implemented. In addition to JVM execution, the DELFT-JAVA architecture provides general support for C compilers and other operations that are required in general purpose processors. Architectural support for Multimedia SIMD and DSP instructions is also incorporated into the architecture.

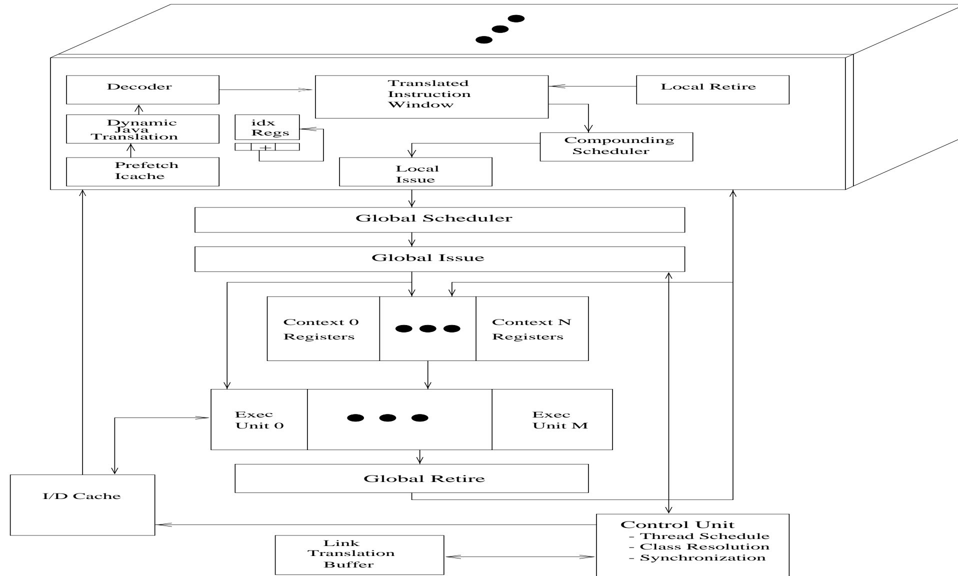

Fig. 1. Concurrent Multithreaded Organization.

2 Delft-Java Architecture

The Delft-Java architecture has two logical views: 1) a JVM Instruction Set Architecture (ISA) and 2) a RISC-based ISA. The JVM view is stack-based with support for standard datatypes, synchronization, object-oriented method invocation, arrays, and object allocation[1]. An important property of Java bytecodes is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code[2]. We utilize this property to dynamically translate Java bytecodes into Delft-Java instructions. 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 Java code include pointer manipulation, Multimedia SIMD instructions, unsigned datatypes, and rounding/saturation modes for DSP algorithms. The remaining sections of this paper refer to the Delft-Java architecture and not to the Java architectural view.

Instructions: In Delft-Java, nearly all instructions are executed as 32-bit fixed width instructions with an 8-bit opcode. A typical encoding useful for Java translation is `add.ind.w32 idx[0], ix, iy-1, it-1`. This instruction specifies a 32-bit, 2's complement integer addition. Normally, the register file is accessed with a direct reference (e.g. `add rx, ry, rt`). The Delft-Java processor facilitates JVM translation by allowing indirect access through 3 indices into the register file which create a circular address. Using this mechanism, it is possible to provide both LIFO stack and FIFO vector operations.

3 Concurrent Multithreaded Organization

An organization of the DELFT-JAVA architecture which supports multiple concurrent execution of threads and shared global execution units is shown in Figure 1. We define a *context* as a hardware supported thread unit. A context does not include 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[3]. In this paper, the term context denotes the hardware on which a thread may run.

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) 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. The *control unit* is responsible for synchronization, cache locking, dynamic linking, I/O, loading instructions, and general 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 JAVA context. 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* which keeps track of issued and pending instructions. The *local instruction scheduler* takes translated instructions in a RISC form and schedules them for execution. The *local issue unit* determines if the instructions that have been locally scheduled can be issued to the global instruction scheduler.

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. The JAVA language specifies that in the absence of explicit synchronization, a JAVA implementation is free to update the main memory in any order[4]. This relaxed memory consistency model allows the scheduler to reorder the instructions from individual contexts to optimize the utilization of the shared execution units. The *global issue unit* ensures that global resources are available prior to issuing instructions. Instructions may be issued in one of two forms: single independent instructions and compound parcels[5].

After 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. Otherwise, the result is maintained in the retire unit until the conditional outcome is known. The *local retire unit* removes the instruction from the window and commits state in the context. Each context may retire multiple instructions per cycle.

Translation: Most JVM bytecodes are translated. However, some more com-

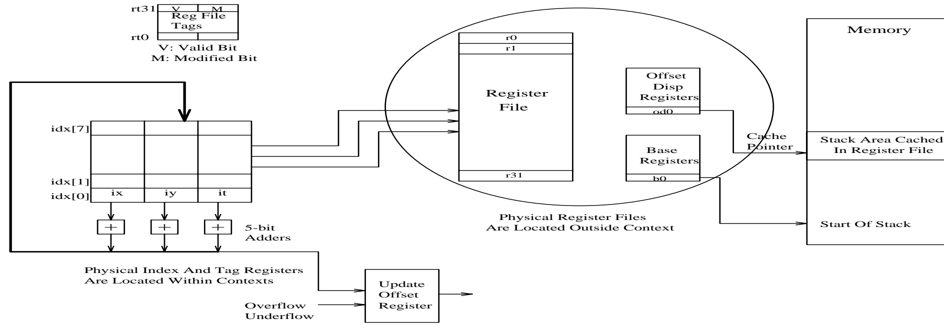

Fig. 2. Concurrent Multithreaded Registers.

plex instructions are directly incorporated into the Delft-Java ISA. In particular, the following categories of instructions are not translated: a) synchronization, b) array management c) object management, d) method invocation, e) exception handling, and f) complex branching. Indirect access to the register file plays the largest role in the translation of Java bytecodes to Delft-Java instructions. As shown in Figure 2, 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 translation phases. The first phase translates the instruction into a valid Delft-Java instruction. In this case, an `add.ind.w32 idx[0] it, it-1, it-1` is generated by the translation logic. As an example, assume the top of stack referenced by `idx[0].rt` points to `r5`. The second phase decodes the indirect register reference and places the decoded instruction into the instruction window as `add.w32 r5, r4, r4`. When the instruction is placed in the window, the `idx[0].rt` is modified by the destination decrement. Functionally, this performs `r5 + r4` $\rightarrow$ `r4`. 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.

Link Translation Buffer: An important consideration in accelerating Java's dynamic linking and polymorphic method invocation is a *Link Translation Buffer* (LTB). The LTB acts as a global repository for dynamically resolved names. During dynamic invocation, the name to be resolved is contained in the constant pool. After a process called resolution[4], the name contained within the constant pool can be associated with a physical location in memory for each object. This association is placed in the Link Translation Buffer. If the control unit finds the constant pool address of the requesting object in the LTB and the requesting class has access permissions to the data, then the LTB efficiently returns the resolved address. A programmer may also completely disable the LTB or more judiciously issue `flushLTB` instructions.

4 Results and Conclusions

We present preliminary results for a 32-bit full complex 4-point FFT kernel. 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. 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 [6].

In this paper we have introduced the DELFT-JAVA processor architecture and organization. The goal of the processor is to exploit the parallelism inherent in JAVA multithreaded programs without requiring the programmer to specify any additional information. We have accomplished this by designing a concurrent multithreaded organization using modern RISC techniques with multiple instruction issue capability per context. Compared to current techniques, our processor efficiently exploits the JAVA programming language to provide a very high performance JAVA processor architecture.

References

1. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.
2. 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).
3. Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press - A Prentice Hall Title, Mountain View, California, 1996.
4. James Gosling, Bill Joy, and Guy Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
5. 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.
6. 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. 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.

This article was processed using the LaTeX macro package with LLNCS style