A Java-enabled DSP

C. John Glossner[1,3], Michael Schulte[2] and Stamatis Vassiliadis[3]

[1] Sandbridge Technologies, Inc., / White Plains, NY, USA
[2] Lehigh University, / Bethlehem, PA, USA
[3] Delft University of Technology, / Delft, The Netherlands
glossner@sandbridgetech.com

Abstract. In this paper we explore design techniques and constraints for enabling high-speed Java-enabled wireless devices. Since Java execution may be required for 3G devices, efficient methods of executing Java bytecode are explored. We begin by setting a historical context for DSP architectures and describe salient characteristics of classical, transitional, and modern DSP architectures. We then discuss methods of executing Java bytecode - both software and hardware - and discuss the merits of each approach. We next describe the Delft-Java engine that we designed at Delft Technical University in the Netherlands. Finally, we compare this design to other techniques and comment on ways that Sandbridge Technologies is modifying organizational characteristics to achieve power-efficient Java execution.

1 Introduction

DSPs have become a ubiquitous enabler for integration of audio, video, and communications[1]. 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 introduced Java-based services for its cellular subscribers and hardware solutions for efficient JAVA execution are being proposed[2]. 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 [3]. 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 a 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[4]. 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.

This paper explores the design constraints in building very high performance engines that enable broadband communications. We look at existing DSPs and characterize their architectures by classical, transitional, and modern architectures. We describe the key characteristics of each generation of DSP architecture. We also look at methods of JAVA execution. We then give a brief introduction to the DELFT-JAVA engine. The DELFT-JAVA engine directly addresses efficient JAVA execution. We then present other related work. Finally, we conclude by describing how the DELFT-JAVA engine is being modified to fit ultra-low power design challenges required by 3G wireless.

2 DSP Architectures

Execution predictability in DSP systems often 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 [5], Lucent's 16xx[6], and IBM's Mwave DSP[7, 8].

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 [6]. Likewise, TI's C55x extends the

C54x to a dual-MAC machine [9]. 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[10] and LSI's ZSP[11].

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 [12, 13, 14, 15], Philips' Trimedia [16], TI C80 [17], and Chromatics MPACT [18].

Another special class of processors with DSP functionality is general-purpose processors which include SIMD extensions. Examples of this include Intel's MMX[19] and PowerPC's Altivec [20]. 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 [21].

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 [22], ADI's TigerSHARC[23, 24], TI's C64x[25], BOPS' ManArray[26, 27], and Lucent's Daytona[28].

3 Methods of Java Execution

JAVA is a C++ like programming language designed for general-purpose object-oriented programming[29]. An appeal for the usage of such a language is its "write once, run anywhere" philosophy [30]. This is accomplished by provid-

ing a JVM interpreter and runtime support for each platform[31]. 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.

JVM 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 JVM 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 [32].

■ **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[33]. 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 [32, 34]. 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[32]. Additionally, multiple instructions are required to implement JVM 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 JVM are integrated into a runtime environment[34, 35, 36]. 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[35], released in 1999, 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 pauseless 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[36] 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[37], 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[34]. 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[33].

The Toba system first translates bytecodes to C and then compiles the C program[38]. 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 JVM. Toba simulates the program counter by assigning values to a local pc variable. Hewlett Packard has a similar system[39]. Using the Toba compiler, performance improvements nearly twice a standard interpreter have been reported for FFT signal processing functions[40, 41]. The Toba group found performance improvements of 2 to 10 times versus a standard interpreter[38].

■ **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" [30], 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 JVM instructions. The gcj compiler is an example of a native compiler[42].

■ **Direct Execution:** The previously mentioned questions could be possibly answered using direct JAVA execution. This approach assumes hardware capabilities that execute the JVM 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 JVM execution time. At that time only Sun Microsystems proposed similar approaches to improving JAVA performance. Sun's *picoJava* implementation directly executes the JVM Instruction Set Architecture (ISA) but incorporates other facilities that improve the system level aspects of JAVA program execution[43, 44, 45]. 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 JVM does not implement an entire machine, Sun added 115 additional extended bytecodes to the picoJava core[45]. 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[46, 47]. Sun states that the picoJava core provides up to 5x performance improvement over JIT compilers [48].

4 The Delft-Java Engine

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[31]. An important property of JAVA bytecodes is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code[49]. 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

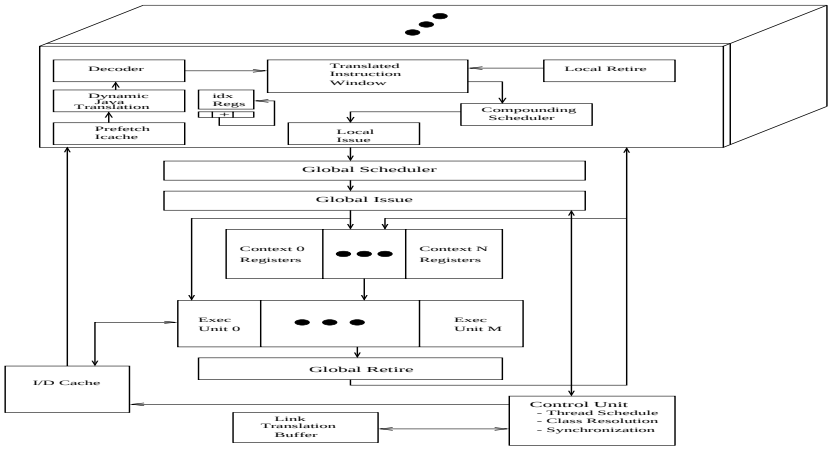

Fig. 1. Concurrent Multithreaded Organization.

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.

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 that create a circular address. Figure 2 shows how the indirect access can be implemented. Using this mechanism, it is possible to provide both LIFO stack and FIFO vector operations.

An organization of the DELFT-JAVA architecture that supports multiple concurrent execution of threads and shared global execution units was presented in[50, 51, 52] and is shown in Figure 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. 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[53]. 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[4]) 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[54, 44]. The DELFT-JAVA processor may also dynamically build internal compound instructions[55]. 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[56].

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[29]. 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 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 inde-

[4] logical meaning not necessarily physical.

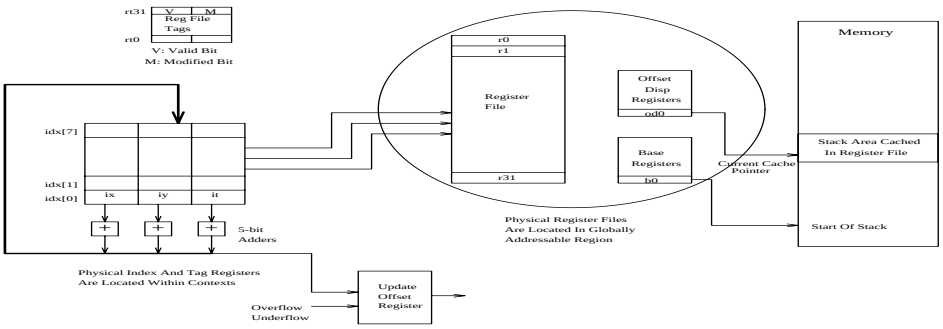

Fig. 2. Concurrent Multithreaded Registers.

pendent, 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 JVM instruction set architecture is inherently stack based [31]. 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:** Indirect access to the register file plays the largest role in the translation of JAVA bytecodes. 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 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. 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 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* [51]. 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.

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 [56], branch prediction, a link translation buffer [51], 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 [57]. 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[55, 58, 56]. 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[59] . 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[29]. 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.

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,

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.

and f) complex branching operations. The JAVA instructions shown in Table 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.

5 Dynamic Translation Results

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 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 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

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

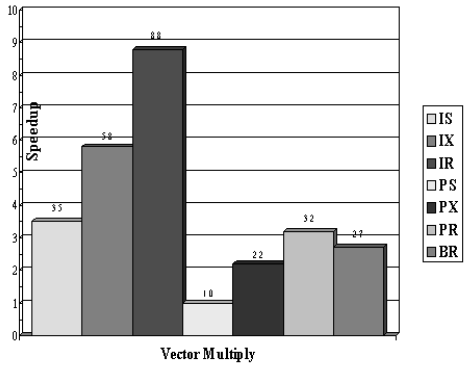

Fig. 3. Performance Results.

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 (BR) utilization. We allowed two concurrent accesses to the Local Variable and Heap memories. We maintained a four cycle latency for each memory space.

Figure 3 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 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 Related Work

Using the Tomasulo technique[59], Munsil and Wang show that an adapted algorithm on simple benchmarks could reduce stack usage[60]. Li et. al. from

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

Tsignhua University also used a Tomasulo algorithm combined with a technique called virtual registers[61]. 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[62]. Complicated bytecode are emulated with software. They have two views of the machine - a Java Virtual Machine view and a RISC-based execution engine. They transition between the two views using a DISP (JVM 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 JVM [63]. 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 JVM instruction set architecture and the ShBoom instruction set architecture.

7 Conclusions

In this paper we have given a historical background of DSP architecture and the design constraints that have influenced their design. We have discussed the issues and characteristics of the JAVA language and how hardware may be applied to accelerate JAVA execution. We have described the DELFT-JAVA engine that is an efficient JAVA bytecode acceleration engine without forcing undue burden on the programmer for specifying parallelism. In addition, we have compared our design to other related projects. The DELFT-JAVA engine provides for efficient JAVA execution targeted for high-performance infrastructure applications. At Sandbridge

Technologies, new multithreaded organizational techniques are being explored for power-efficient high-performance JAVA execution that will allow handsets to benefit from these techniques. Some of these techniques include software-controlled performance tuning, LIW organizations (in contrast with superscalar or direct execution organizations), and alternative thread-execution models (in contrast with simultaneous multithreading). It is anticipated that these techniques may bring to the handset environment a real-time, power-efficient platform with automatic extraction of parallelism for JAVA applications.

References

1. J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, pages 51–59, August 1998.
2. Junko Yoshida. Java chip vendors set for cellular skirmish. EE Times, January 30 2001.
3. P. Lapley. *DSP Processor Fundamentals.* IEEE press, New York, 1997.
4. 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.
5. Texas Instruments. TMS320C54x DSP Reference Set. Volume 1: CPU and Perhipherals. Technical Report SPRU131E, Texas Instruments, June 1998.
6. Jeff Bier. DSP16xxx Targets Communictaions Apps. *Microprocessor Report*, 11(12), 1997.
7. 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.
8. 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.
9. Tom R. Halfhill. TI Cores Accelerate DSP Arms Race. *Microprocessor Report*, March 6 2000.
10. J. Eyre and J. Bier. Carmel Enables Customizable DSP. *Microprocessor Report*, 12(17), December 1998.
11. LSI Corporation. *LSI402Z Digital Signal Processor.* LSI Corporation, r20012 edition, 1999.
12. 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.
13. 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.
14. 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.

15. 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.
16. B. Case. Philips hopes to displace DSPs with VLIW. *Microprocessor Report*, pages 12–15, December 1997.
17. C. P. Feigel. TI Introduces Four-Processor DSP Chip. *Microprocessor Report*, 8(4), March 28 1994.
18. Dave Epstein. Chromatic Raises the Multimedia Bar. *Microprocessor Report*, 9(14), October 28 1995.
19. A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 42–50, August 1996.
20. 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.
21. 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.
22. O. Wolf and J. Bier. StarCore Launches First Architecture. *Microprocessor Report*, 12(14), October 1998.
23. O. Wolf and J. Bier. TigerSHARC Sinks Teeth Into VLIW. *Microprocessor Report*, 12(16), December 1998.
24. J. Fridman and Z. Greenfield. The TigerSHARC DSP Architecture. *IEEE Micro*, 20(1):66–76, January 2000.
25. J. Turley and H. Hakkarainen. TI's New C6x Screams at 1,600 MIPS. *Microprocessor Report*, 11(2), 1997.
26. 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.
27. 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.
28. 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.
29. James Gosling, Bill Joy, and Guy Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
30. 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.
31. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.
32. 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.

33. 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.
34. 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.
35. Sun Microsystems. The Java Hotspot Performance Engine Architecture. Sun Microsystems, 1999. http://java.sun.com/ products/ hotspot/ whitepaper.html.
36. 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.
37. Michal Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 9(4):1063–1073, November 1997.
38. 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.
39. Hewlett Packard. HP Turbo Chai Release 2.0. Hewlett-Packard, May 1999. http://www.hp.com/emso/products/turbochai/TchaiPDF.pdf.
40. 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.
41. 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.
42. Cygnus. Gcj compiler, 1999.
43. 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.
44. Marc Tremblay and Micahel O'Connor. picoJava: A Hardware Implementation of the Java Virtual Machine. In *Hotchips Presentation*, 1996.
45. Harlan McGhan and Mike O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998.
46. 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.
47. 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.
48. 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.
49. 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).
50. J. Glossner and S. Vassiliadis. Delft-Java Dynamic Translation. In *Proceedings of the 25th EUROMICRO conference (EUROMICRO '99)*, volume 1, Milan, Italy, September 8-10 1999.
51. 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.
52. 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.
53. Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press - A Prentice Hall Title, Mountain View, California, 1996.
54. Peter Wayner. Sun gambles on java chips. *Byte*, 21(11):79–85, November 1996.
55. 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.
56. James Philips and Stamatis Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, 43(3):257–268, March 1994.
57. Brian Case. Implementing the java virtual machine. *Microprocessor Report*, 10(4):12–17, March 25 1996.
58. Stamatis Vassiliadis, James Phillips, and Bart Blanar. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.
59. R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, II:25–33, 1967.
60. Wes Munsil and Chia-Jiu Wang. Reducing Stack Usage in Java Bytecode Execution. *Computer Architecture News*, 1(7):7–11, March 1998.
61. 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.
62. Advancel Logic Corporation. TinyJ Processor Core Product Datasheet. Datasheet, July 1999.
63. Patriot Scientific Corporation. Psc1000/a microprocessor datasheet. Patriot Scientific, 1997. http://www.ptsc.com/downloads/ psc1000/specs/datasheet.pdf.