Improving Java Performance Using Hardware Translation

Ramesh Radhakrishnan, Ravi Bhargava and Lizy K. John
Laboratory for Computer Architecture
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{radhakri, ravib, ljohn}@ece.utexas.edu

ABSTRACT

State of the art Java Virtual Machines with Just-In-Time (JIT) compilers make use of advanced compiler techniques, run-time profiling and adaptive compilation to improve performance. However, these techniques for alleviating performance bottlenecks are more effective in long running workloads, such as server applications. Short running Java programs, or client workloads, spend a large fraction of their execution time in compilation instead of useful execution when run using JIT compilers. In short running Java programs, the benefits of runtime translation do not compensate for the overhead.

We propose using hardware support to perform efficient Java translation coupled with a light-weight runtime environment. The additional hardware performs the translation of Java bytecodes to native code, thus eliminating much of the overhead of software translation. A *translated code buffer* is used to hold the translated code, enabling reuse at the bytecode level. The proposed hardware can be used in any general purpose processor without degrading performance of native code. The proposed technique is extremely effective for short running client workloads. A performance improvement of 2.8 times to 7.7 times over a software interpreter is achieved. When compared to a JIT compiler all SPECjvm98 benchmarks except one show a performance improvement ranging from 2.7 times to 5.0 times. A performance degradation (0.58 times) is observed for one benchmark which is long running. Allowing hardware translation to perform optimizations similar to JIT compilers and Java processors will execute long running programs more efficiently and provide speedups similar to that of client workloads.

1. INTRODUCTION

Java is widely used to develop software in different environments, ranging from embedded systems to high-end server applications. The popularity of Java is fueled by features like its platform independence, object-oriented nature, and security. Platform independence is attained by compiling Java to an intermediate "bytecode" format, which is the instruction set architecture (ISA) of the Java Virtual Machine (JVM) [1]. Java bytecodes, which are stack-based instructions, can be executed on any platform which has an implementation of the JVM. The JVM dynamically interprets each bytecode into platform-specific machine instructions to execute the Java program.

Interpretation and other forms of software emulation of bytecodes are very slow. The interpreter is typically slow because fetch and decode functionalities of normal program execution (reading and updating program counters, decoding the instruction, transferring control to activities that correspond to the opcode of the instruction just decoded, etc) are performed in software. While executing natively compiled code, the microprocessor hardware performs the fetch and decode actions. Therefore natively compiled code executes much faster compared to software translated code.

Instead of dynamically interpreting each bytecode at runtime, optimized JVMs use a Just In Time (JIT) compiler [2, 3, 4] to compile Java methods (i.e. functions) to native code. JIT compilers compile bytecodes into native code at run time, thereby adding compilation time to the total execution time. Initial JIT compilers were quick and dirty, generating unoptimized code for any method which was invoked. Current Java run time environments are more sophisticated and mix interpretation and compilation using profiling information [5, 6]. This allows the compiler to spend more time on optimizing frequently executed methods, which lowers the execution time spent in the compiled methods. This method of execution which mix interpretation and JIT compilation is called adaptive or mixed-mode compilation. An example of a compiler that uses adaptive compilation is the Jalapeno [7] dynamic optimizing compiler which uses a compile-only approach to program execution. The Jalapeno compiler compiles the code with a low level of optimization at the beginning. As hot regions of the program are recognized, higher levels of optimization are used.

1.1 Limitations of JIT compilers

JIT and adaptive compilers do not improve performance for all workloads. Programs which touch a lot of code and do not spend too much time in loops can get better performance when executed using an interpreter. We used two micro-benchmarks to compare the execution times of different execution modes. Table 1 shows the execution cycles,

ICS '01 Sorrento, Italy

instructions executed and IPC seen when executed using an interpreter, JIT compiler and an adaptive compiler, HotSpot [5]. The interpreter executes faster than a JIT compiler or the HotSpot compiler for both the micro-benchmarks. The overhead of compiling a method occurs at run-time and is a more expensive option than interpretation. Server applications which are typically long running exhibit significant code reuse are ideal workloads for JIT compilers.

A breakdown of execution time shows that compilation constitutes a major fraction of total execution time in certain Java applications. Figure 1 shows the time spent translating and executing for five benchmarks from the SPECjvm98 [8] benchmark suite and the *hello world* micro-benchmark. It is observed that short running programs such as *db* and *Hello World* spend more time in translation than program execution when using JIT compilers. *Compress* is a long running benchmark displaying characteristics similar to server workloads, and benefits the most from JIT execution, spending only a small fraction of the total execution time in translation.

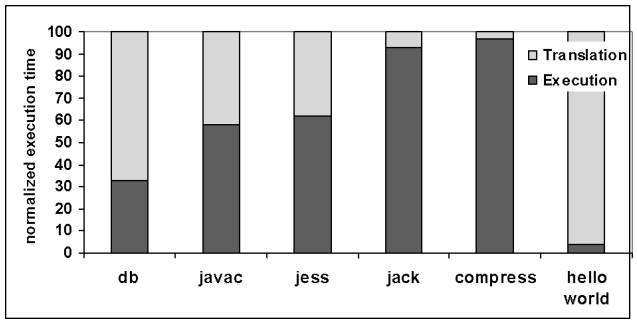

Figure 1: Breakdown of execution time

Another noted problem with JIT compilers is an increase in data cache misses. Data cache miss rates during execution with a JIT compiler are dominated by compulsory write misses, which are caused by installation of the translated code [9]. Another observed phenomenon is the depositing of translated code into the data cache and subsequent fetching of the same code from the instruction cache, resulting in avoidable data transfer and double-caching [9]. Due to all these issues, JIT compilers are fruitful only in certain environments.

1.2 Bridging the architectural gap

Interpreted execution of Java workloads using a native interpreter requires 35 SPARC instructions on average to emulate each bytecode [10]. The average number of SPARC instructions required to emulate a bytecode is approximately 20 when using a JIT compiler [11]. These instruction counts include the interpretive or translative overhead of the interpreter and JIT. The actual instructions required for the program to execute is only a small subset of the operations that happen during Java program execution. Software emulation is easy to implement for new platforms but cannot offer a solution for fast execution of Java bytecodes.

Using hardware support to assist the execution of bytecodes eliminates the requirement of a software layer to emulate the bytecodes. The execution of Java programs can be improved by a hardware accelerator or coprocessor that works in conjunction with a standard microprocessor. Essentially the effort is to bridge the semantic gap that exists between the bytecodes and the native instructions. The bytecode ISA is stack-based and uses few registers. General purpose RISC and CISC machines are register-based, and the *architectural gap*[1] between the JVM and these general purpose processors is large. If dedicated Java hardware, such as the picoJava processor, is used as a coprocessor, Java execution can be relegated to this Java processor and execution of other languages can be performed on the main processor.

By using a dedicated Java processor, the emulating microarchitecture is brought up to the level of the ISA being emulated, bridging the architectural gap. However, our studies show that this is not the best way to bridge this gap, especially for high performance environments. In modern high performance processors, much of the performance is obtained through instruction level parallelism (ILP). The stack nature of the emulating architecture can significantly limit the available parallelism. Figure 2 compares the available parallelism of Java programs to SPECint95 and C++ applications for a Machine Level Parallelism (MLP) of 8, 16, 32, 64, 128 and infinite.

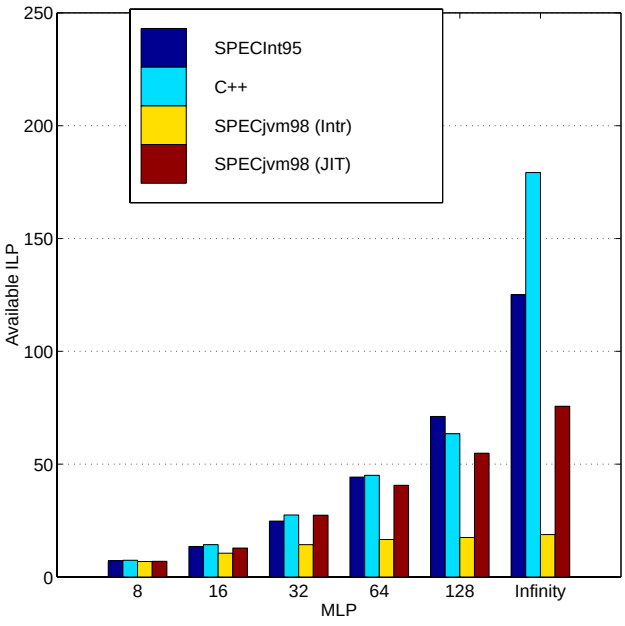

Figure 2: Available ILP in Java workloads

For this ILP study, only true dependencies are considered while scheduling instructions. Perfect branch prediction and

[1]The architectural gap refers to the gap that exists between the emulated architecture (JVM) and the emulating architecture (the general purpose processor).

Execution Mode	hello world			AddClass		
	Cycles	Inst	IPC	Cycles	Inst	IPC
JDK 1.1.6 Interpreter	7.29M	21.76M	2.98	7.24M	21.64M	2.98
JDK 1.1.6 JIT	9.67M	27.85M	2.87	8.72M	25.50M	2.92
JDK 1.2 Interpreter	19.58M	46.44M	2.37	19.69M	46.95M	2.38
JDK 1.2 JIT	22.64M	55.80M	2.46	22.55M	55.79M	2.47
JDK 1.2.2 HotSpot	38.19M	88.23M	2.31	37.64M	88.45M	2.35

Table 1: Comparison of different software translation modes

memory disambiguation are assumed. The latency of all operations is set to one cycle. The harmonic mean of the available parallelism for the various benchmarks are presented. The extremely low ILP of the interpreted Java programs can be attributed to the stack-based implementation of the JVM which imposes a strict ordering on the execution of the bytecodes.

An alternative to a coprocessor is to provide support in the microprocessor to decode the bytecodes to native code. This is similar to the approach taken in recent IA32 processors [12] where X86 instructions are decoded dynamically in hardware to simpler instructions. Original X86 instructions are complex and variable-length taking multiple cycles to execute. To alleviate this problem, the complex X86 instructions are decoded or translated into one or more fixed-length RISC instructions called micro-ops. The micro-ops are then executed in an aggressive RISC-style core. An X86 instruction that translates to four or fewer micro-ops is decoded in hardware. If an X86 instruction is more than seven bytes or translates to more than four micro-ops it is sent to the micro instruction sequencer (MIS). The MIS is a microcode ROM that contains the series of micro-ops associated with each complex X86 instruction. Thus the RISC-style core is the emulating architecture and the original x86 ISA is the emulated ISA.

1.3 Overview of the paper

We propose an approach similar to the Pentium Pro for translating Java bytecodes to the native instruction set of the processor by using hardware decoding. A hardware translator for the bytecode incurs a lower overhead than software translation and thereby improves Java execution. The proposed architecture also stores the recently executed bytecodes to capture bytecode reuse, whereas a pure software interpreter does not reuse previous translations. Section 2 describes the implementation of the proposed architecture which uses a hardware translator. The experimental methodology is explained in Section 3. The performance results and analysis are presented in Section 4. Simulation studies based on SPECjvm98 benchmarks show that the proposed architecture improves performance by 2.8 to 7.7 times relative to interpreters and 0.58 to 5.08 times relative to JIT compilers. Benefits from the proposed scheme come from many sources (i) performing bytecode to native code conversion in hardware as opposed to software, (ii) translation of bytecodes in a pipelined fashion concurrently with the execution of the native code, and (iii) capturing bytecode reuse using the translated code buffer. Related work is presented in Section 5 and concluding remarks are offered in Section 6. The main contributions of our paper include: (i) proposal and implementation details of hardware translation for Java bytecodes, (ii) comparison of hardware translation with state of the art JITs and interpreters, and (iii) detailed performance evaluation illustrating sources of improvement.

2. IMPLEMENTATION OF THE HARDWARE TRANSLATOR

The execution of a bytecode requires the execution of a short sequence of native machine instructions. Software interpretation uses an execution mechanism which consists of repeatedly executing the following steps:

- **step 1.** Locating the next bytecode to be executed, fetching the bytecode and decoding it. This involves:
  - reading and updating a program counter for the bytecodes
  - reading the contents of memory addressed by the bytecode PC
  - decoding the bytecode to get opcode, operands etc
  - transferring control to a routine which corresponds to the opcode just decoded
- **step 2.** Executing the semantics that correspond to the decoded bytecode. This involves executing a short sequence of machine instructions.
- **step 3.** Transferring control back to step 1.

The JIT compiler also uses a similar mechanism, but when it comes across a method call it transfers control to the compiler. The compiler checks if a translated routine exists for the method and if it does not, proceeds to compile the method. In a sense, the JIT compiler creates a semantic routine for each method invoked in the Java program at runtime, whereas the interpreter uses a semantic routine for individual bytecodes.

Step 2 actions are called *executive actions* and form the *execution phase* of the software translation. Steps 1 and 3 are responsible for fetching and decode of bytecodes and perform the *mapping actions*. In natively compiled code the mapping actions are performed in hardware, which is significantly faster. A significant slowdown is observed for software translated code, since the mapping actions are performed in software. Using semantics different from native code precludes the use of existing hardware in a general purpose processor to perform the mapping actions.

Our proposal is to use additional hardware in a general purpose processor to perform the mapping actions involved in execution of Java bytecodes. The additional hardware will have functionality similar to a software interpreter. We use the term Hard-Int (Hardware Interpreter) in the rest of the paper to denote a general purpose architecture that uses the proposed hardware. The Hard-Int architecture performs the mapping from bytecodes to native code using hardware, instead of software. The translated instructions are fed to the processor core which executes them in the normal fashion. A high-level block diagram of the Hard-Int system is shown in Figure 3. In addition to using a hardware translator, a *translated code buffer* is added to facilitate reuse of the translated bytecodes.

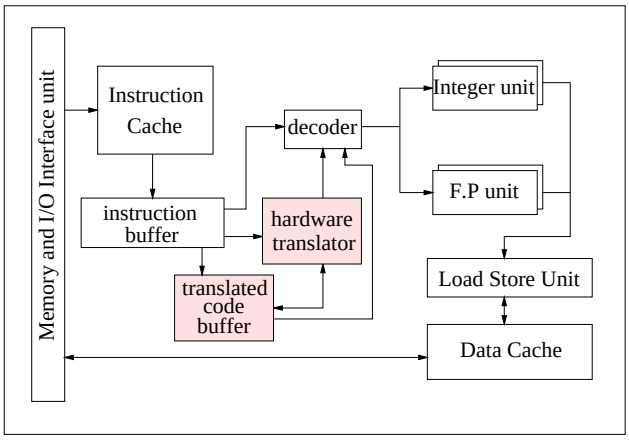

Figure 3: The Hardware Interpreter (Hard-Int) architecture. Proposed additions are shaded in grey.

Figure 4 illustrates the proposed Hard-Int units. The hardware translation unit fetches the bytecodes from the instruction cache and predecodes them before performing the translation using the microcode ROM, which contains the translated code for that bytecode. Complex bytecodes (which form less than 5% of the dynamic instructions) do not contain entries in the ROM. A dispatch table contains the starting address of corresponding interpreter routine for these complex bytecodes. Software translation is used to execute these less frequently used bytecodes. The bytecode PC is stored in a register (B-PC) and is read when fetching the next bytecode to be executed. After a bytecode is decoded the corresponding native instructions are sent to the decoder from the ROM and are also stored in the translated code buffer. During a fetch, the translated code buffer is checked in parallel with the instruction cache. In the event of a translated code buffer hit, the native instructions are fed to the decoder from the translated code buffer. By providing the instructions from the translated code buffer, an instruction cache access and the hardware mapping (using the microcoded ROM) is bypassed. A translated code buffer will enable further optimization of bytecode sequences, similar to the peephole optimizations performed by a JIT compiler.

The proposed architecture is very similar to an instruction path coprocessor [13]. Typically coprocessors have been used in general purpose processors to speedup specific tasks. Coprocessors have been used for performing floating point calculations, graphics and memory management. These co-

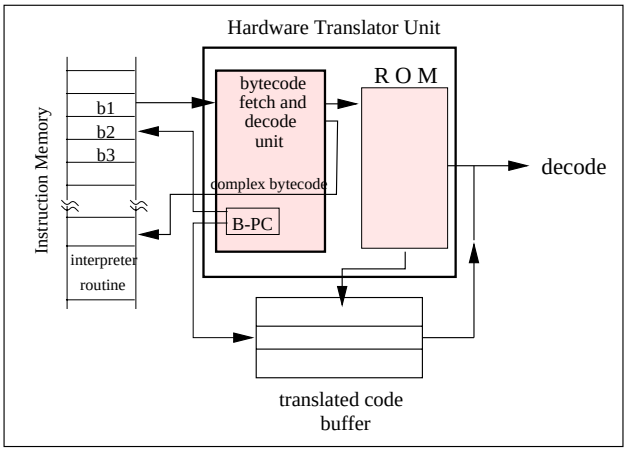

Figure 4: Translating bytecodes in the Hard-Int architecture

processors exist in the data path of the processor and do not help in fetching or decoding instructions. Coprocessors which exist on the instruction path of the processor, obtain instructions from memory, and execute them are called *instruction path coprocessors* [13]. The *instruction path coprocessor* was introduced as a way to provide hardware support to speed up interpretive applications by executing the interpreter mapping concurrently with the CPU using a coprocessor. A programmable instruction path coprocessor is used to perform dynamic transformations to the object code in [14].

The proposed architecture may alternatively be viewed as a translate coprocessor. Along the style of decoupled access-execute architectures [15], it can constitute a decoupled translate-execute architecture, with the translate processor performing the conversion from bytecode to native sequence. The execute processor is responsible for only the executive actions pertaining to the program, whereas the translative or mapping actions corresponding to the program are handled by the translate processor.

Examples of bytecode translation

To illustrate the process, we provide a few simple examples of the translation of bytecodes to native code (with SPARC as the native ISA). We will translate the following bytecodes: (i) *swap* which swaps the top two words on top of the operand stack, (ii) *dup* which duplicates the top word on the operand stack and pushes it into the operand stack and (iii) *lstore* which stores the word on the top of the operand stack, of type *long* into the local variable area. The offset into the local variable is obtained from an unsigned byte following the *lstore* opcode.

The mapped instructions that are stored in the microcoded ROM are shown in SPARC assembly, and variable names are used in place of explicit register numbers. Variables used here are: (i) SP - pointer to the topmost entry of the operand stack (ii) TOSA and TOSB - registers to hold temporary computation values, (iii) NextByte - parameter for current bytecode (obtained by doing a fetch ahead of the instruction stream) and (iv) Varbase - pointer to array of local variable.

BYTECODE	SPARC ops	comments
swap	ld [SP], TOSB	;load TOS entry to register
	ld [SP-4], TOSA	;load TOS-1 entry to register
	st TOSB, [SP-4]	;swap entries in the stack
	st TOSA, [SP]	
dup	ld [SP], TOSA	;load TOS into register
	st TOSA, [SP+4]	;store register value to TOS+1
	inc 4, SP	;increment stack top pointer
lstore	ld [SP-4], TOSA	;load TOS-1 entry to register
	sll NextByte, 2, T2	;get offset value from NextByte, store in register
	dec 4, SP	; decrement stack pointer
	ld [SP+4], TOSB	;load TOS into register
	st TOSA, [Varbase+T2]	;store 1st register value to local variable area
	add Varbase, 4, T1	;point to next location
	st TOSB, [T1+T2]	; store 2nd register value to local variable area

Table 2: Examples of translation routines used for *swap*, *dup* and *lstore*

A similar mapping scheme is used for most of the bytecodes and the SPARC instructions are stored in the microcoded ROM. The translated code contains stack references and accesses to temporary registers. To avoid register usage interfere with the registers used in the JVM, we use the same register mapping model used in the JDK 1.2 interpreter (implemented in SPARC assembly language) to minimize register usage interference between the JVM and the translated code. All accesses to the Java stack are treated as memory references, unless they have been optimized and stored in temporary registers previously.

3. EXPERIMENTAL EVALUATION

The performance of the proposed model is evaluated using the SPECjvm98 benchmarks [8]. A description of the benchmarks is given in Table 3, along with the number of methods compiled and invoked at runtime. The benchmarks are run with the *s1* dataset, since it generates workloads with contrasting characteristics (both server-like and client-like). A characteristic of a client application is execution time spread somewhat evenly across all the methods. The method information provided in Table 3 shows that benchmarks like *db*, *jess* and *javac* show characteristics of client applications. In benchmarks like *mtrt* and *mpegaudio*, the majority of the execution time is concentrated in a few methods. This is typical of server workloads which attain good performance using JIT compilers.

The performance of the Hard-Int architecture is evaluated by comparing it to different JVMs running on a 4-way superscalar processor machine as well as an aggressive 16-way superscalar processor machine. A detailed, cycle-accurate microprocessor simulator using Shade [16] as the functional execution engine is used to compare the performance of the Hard-Int architecture against the Sun JDK 1.1.6 and Sun JDK 1.2 JVMs (which are run using an interpreter and JIT compiler). The trace to the simulator includes class loading, verification, garbage collection, and synchronization, in addition to the instructions executed for translation of the bytecodes. Therefore all aspects of the Java execution are captured in the data that is presented in Section 4. Table 4 shows the configuration of the 4-way superscalar and 16-way superscalar machines used in this study.

The runtime environment for JDK 1.2 is more optimized compared to JDK 1.1.6 and uses more efficient synchronization and garbage collection. The JIT compiler used in JDK 1.2 is more mature compared to the JDK 1.1.6 JIT, performing more optimizations and selective compilation. Recently the Java HotSpot Client Virtual Machine was introduced as a replacement for the classic virtual machine in the previous versions of the Java 2 SDK to offer improved runtime performance for client applications and applets. The Java HotSpot Client VM has been specially tuned to reduce application start-up time and the memory footprint, making it particularly well suited for client environments. However, at the time of this experiment the Client VM was not available and hence we do not have any studies comparing our hardware approach to the client VM. It may be observed at the SPEC web site that for the SPECjvm98 benchmarks, the HotSpot Client VM outperforms the Server VM by approximately 8% for the first run, and the Server VM outperforms the Client VM by 48% in the subsequent runs. Since we consider only first runs in our simulations, i.e we include the start up time and overhead of initializing the VM, the HotSpot Client VM would perform better than the Server VM in our simulation environment. Therefore we speculate that speedups observed using the hardware approach and Client VM will be 20-50% lower than what we present for the JDK 1.2 JVM.

To model Hard-Int architecture, the simulator is modified to incorporate the hardware translator and a table lookup scheme to decode the bytecodes. A mapping from bytecode to SPARC native instructions as described in the previous section is used to emulate the bytecodes. The mappings are stored in a microcoded ROM and a latency of 3 cycles is at-

benchmark	description	no. of methods compiled	no. of methods invoked	bytecodes executed
compress	A popular LZW compression program	577	17,330,744	954,990,234
db	Data management benchmarking software written by IBM	642	65,379	2,035,798
javac	The JDK Java compiler from Sun Microsystems	1,384	213,243	5,958,654
jess	Java version of NASA's CLIPS rule-based expert systems	1,222	414,349	8,126,332
mpegaudio	The core algorithm that decodes an MPEG-3 audio stream	843	954,605	115748387
mtrt	A dual-threaded program that ray traces an image file	781	1,906,112	50683565
jack	A parser-generator from Sun Microsystems	1,230	2,318,110	175,740,325

Table 3: Description of the method frequency in the SPECjvm98 benchmarks

tributed to decode the bytecode and fetch the corresponding mapping from the ROM. A 64-entry, 4-way translated code buffer with an access time of two cycles is used to cache the SPARC instructions associated with recently decoded bytecodes. Each line of the translated code buffer can hold up to 16 SPARC instructions associated with one or more bytecodes. If a bytecode maps to more than 16 SPARC instructions, the translated code buffer is accessed in the next cycle to retrieve the remaining instructions. In the event of a translated code buffer miss, the bytecodes are fetched from the instruction cache and decoded before obtaining the corresponding entries from microcoded ROM. This suffers the latency of an instruction cache access in addition to three cycles to access the microcoded ROM. The rest of the configuration of the Hard-Int architecture remains the same as in Table 4. Our simulator does not implement the whole JVM specification, just the translation of the bytecodes. To allow fair comparison we calculate the cycles spent to perform other tasks in the form of system calls and Java Network Interface for each benchmark (during interpreted execution), and add that as part of the execution time for the Hard-Int model.

4. PERFORMANCE RESULTS

The performance of the proposed model is compared to both the interpreter and JIT execution modes using JDK 1.1.6 and JDK 1.2 JVM runtime environments for the SPECjvm98 benchmarks (except for *compress* and *jack* which did not run to completion due to problems in the simulation environment).

4.1 Execution speedup

Figure 5 shows the number of execution cycles needed to run the SPECjvm98 benchmarks on a 4-way processor machine for the two interpreters, the two JIT compilers and the Hard-Int architecture. The Hard-Int architecture performs better than the interpreter for all the benchmarks, resulting in speedups of 2.81 for *db* to 7.7 for *jess*. The translation overhead for a bytecode in the Hard-Int architecture is smaller than the overhead incurred in interpreters. The interpreter must fetch each bytecode from the data cache, and perform certain bookkeeping tasks (like incrementing the PC and stack pointers) before executing each bytecode. By lowering the translation overhead for executing each bytecode, the Hard-Int architecture performs better than the interpreter for all the benchmarks.

On comparing the performance of the Hard-Int architecture to that of the JIT compiler, Hard-Int provides speedup ranging from 2.68 for *mtrt* to 5.1 for *jess*, except in the case of *mpeg*. For the *mpeg* benchmark, the JIT compiler has a speedup of 1.72 over the Hard-Int architecture. The JIT compiler performs better for this benchmark since most of the execution time is spent in a few methods which were compiled by the JIT compiler during the initial phases. The JIT compiler obtains a speedup of 7.3 over the interpreter for *mpeg* illustrating the exploitation of method reuse. The high speedup obtained by Hard-Int for *db*, *jess* and *javac* shows that the Hard-Int model performs consistently better for client workloads, where the JIT results in little or no improvement over an interpreter. For certain long running workloads (like *mpeg*) the Hard-Int model performs better than the interpreter. However, there is performance degradation in Hard-Int when compared to a JIT compiler. The performance of the Hard-Int architecture can be improved further by lowering the overhead associated with the fetch and decode of bytecodes. A hardwired decoder for the simpler and more frequently executed bytecodes will reduce the decode overhead to one cycle. Using a bigger translated code buffer and allowing for optimizations on the stored bytecodes will result in generation of more optimized sequences of bytecodes, similar to the code generated by JIT compilers.

Figure 6 shows the number of cycles executed for the benchmarks when run on a 16-way machine. The Hard-Int architecture outperforms both interpreters in all cases, with speedups in the 2.86 to 7.57 range. There is improvement over JITs in all benchmarks except *mpeg*. The average speedup (geometric mean) over the five benchmarks is 2.26 (over JDK 1.2 JIT), 2.07 (over JDK 1.1.6 JIT), 5.07 (over JDK 1.2 interpreter) and 3.95 (over JDK 1.1.6 interpreter). The current implementation of the hardware translator converts one or more bytecodes at a time, and stores the translated instructions in the translated code buffer. If we store traces of bytecodes, similar to a trace cache, we can improve the ILP and increase performance when going to wider-issue machines. Another option to improve performance is to store the translated code for methods together, and fetch them on a method call which will increase the ILP and help the Hard-Int architecture to be effective in wide issue machines.

4.2 Cache performance

The cache performance of the SPECjvm98 benchmarks for the various execution modes is shown in Table 5. When using a hardware translator the bytecodes are stored and fetched from the instruction cache and the translated code is written to the translated code buffer. We see better instruction and data cache performance using this approach.

DATA MEMORY

	Configuration-1	Configuration-2
- L1 Data Cache:	4-way, 64KB, 1-cycle access	4-way, 64KB, 1-cycle access
- L2 Unified cache:	4-way, 1 MB, +7 cycles	4-way, 1 MB, +7 cycles
- Non-blocking	2 MSHRs and 1 port	8 MSHRs and 4 port
- D-TLB	128-entry, 1-cycle hit, 30-cycle miss	128-entry, 1-cycle hit, 30-cycle miss
- Store buffer:	32-entry w/load forwarding	128-entry w/load forwarding
- Main memory:	Infinite, +22 cycles	Infinite, +22 cycles

FETCH ENGINE

	Configuration-1	Configuration-2
- L1 Instruction Cache:	4-way, 64KB, 1-cycle hit	4-way, 64KB, 1-cycle hit
- Branch Predictor:	16k gshare predictor 8-cycle misprediction penalty	16k gshare predictor 8-cycle misprediction penalty
- Indirect Branch target buffer:	512 entries	512 entries

EXECUTION CORE

	Configuration-1			Configuration-2		
Int/FP ROB entries	64 each			144 each		
Decode Width	4 instructions			16 instructions		
Issue Width	4 instructions			16 instructions		
Execute Width	4 instructions			16 instructions		
Retire Width	4 instructions			16 instructions		
- Functional unit	#	exec. lat.	issue lat.	#	exec. lat.	issue lat.
Load/store	2	1 cycle	1 cycle	6	1 cycle	1 cycle
Simple Integer	3	1	1	9	1	1
Int. Mul/Div	2	3/20	1/19	6	3/20	1/19
Simple FP	2	3	1	4	3	1
FP Mul/Div/Sqrt	2	3/12/24	1/12/24	4	3/12/24	1/12/24

Table 4: Configurations of simulated processor

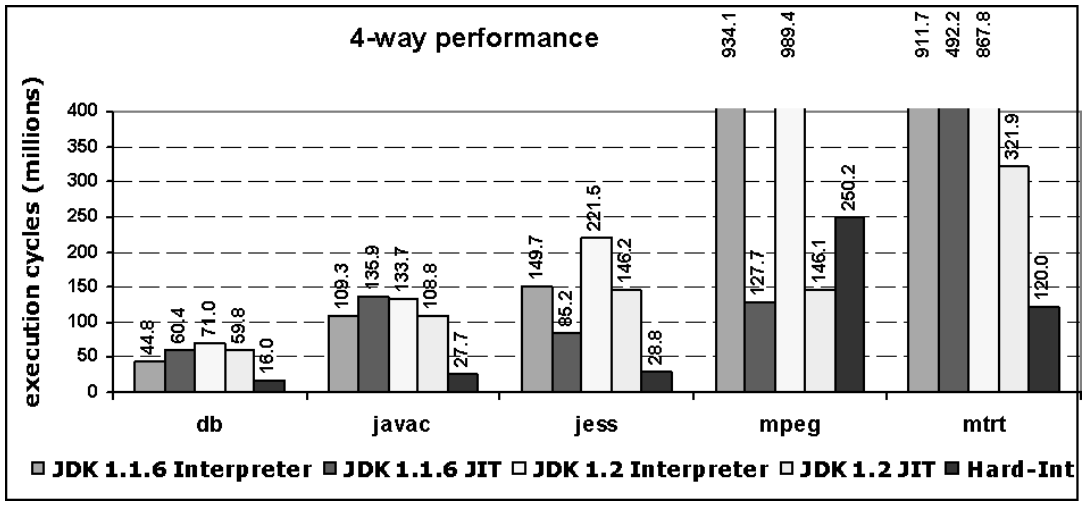

Figure 5: Execution cycles for different execution modes on a 4-way machine

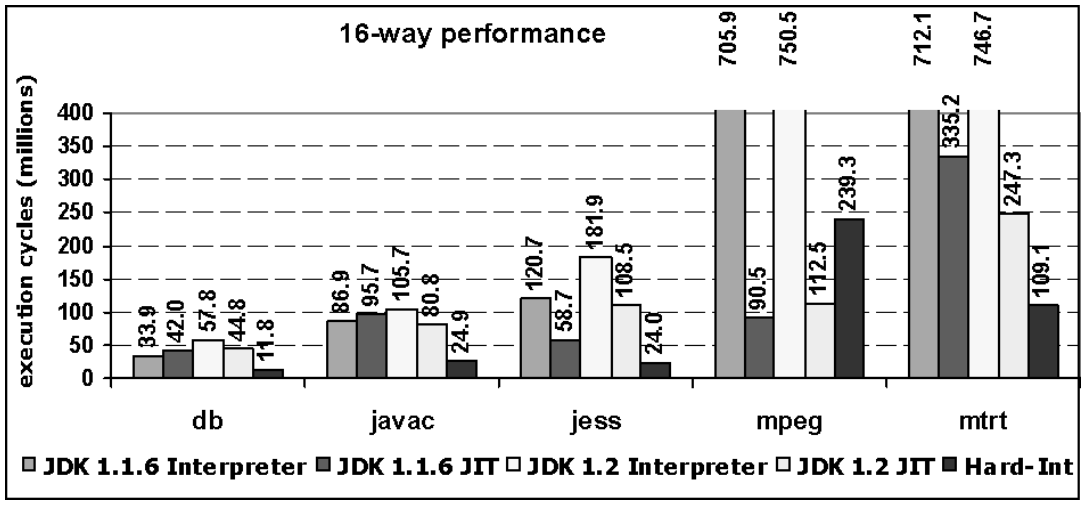

Figure 6: Execution cycles for different execution modes on a 16-way machine

The better data cache performance is the result of not storing the bytecodes, the translated code for the bytecodes, or the large data structures required by the JIT compiler in the data cache. Fewer instruction and data cache references are generated when using the Hard-Int architecture. There are fewer cache misses in a majority of cases, except a few benchmarks where the JDK 1.1.6 interpreter shows impressive instruction cache performance. The interpreter executes a basic loop that fetches and decodes a bytecode fetched from the data cache. The instruction footprint of the interpreter is small and we see very few misses in the instruction cache. The JIT compiler on the other hand has a larger instruction footprint than the interpreter and also generates references to the compiled native code. The working set switches between the compiled code and the code for the JIT compiler often, resulting in more instruction cache misses.

The difference in number of misses is more prominent in the case of the data cache, where the JIT compiler has higher misses that the interpreter for all the benchmarks. This is the case even when the interpreter has more data references in certain benchmarks such as *mtrt* and *javac*. Loads to fetch bytecodes are the main source of data cache references for the interpreter. In the JIT compiled execution, the compiled code has to be written to memory and cause write misses in the data cache. Data cache misses are also caused by frequent accesses to large data structures used by the JIT compiler. When that code is executed for the first time it also causes misses in the instruction cache.

4.3 Performance of the translated code buffer

The translated code buffer is a 64-entry, four-way associative buffer which is used to store the corresponding SPARC instructions associated with recently executed bytecodes. It is a means of enabling some reuse at the bytecode level. A JIT compiler stores the translated code for methods, and thereby making it unnecessary to translate that method when it is invoked in the future. Similarly, but at a finer granularity, we store the translated code for each bytecode in a buffer and provide the translation from the buffer if the bytecode is executed again. If the translated code for a bytecode does not exist in the translated code buffer, the mapping is provided by the microcoded ROM after the bytecode is fetched and decoded.

Table 6 shows the references and misses encountered by the translated code buffer. Using a small buffer of only 64 entries, a hit rate of 54.7% to 81.3% is observed. The translated code buffer used here is a very simple structure. Optimizations can also be done on the stored entries in the translated code buffer which will allow the performance of the hardware translator to be more on par with the JIT performance for server workloads.

4.4 Emulation cost

The larger the architectural gap that exists between the JVM and the emulating processor, the higher the number of cycles taken to emulate a bytecode. Figure 7 shows the average number of cycles to emulate (translate and execute) a single bytecode on a 4-way processor. Using an interpreter, the overhead associated with fetching and decoding each bytecode in software is high resulting in a large translation cost ranging from 8.0 cycles per bytecode executed (for *mpeg*) to 35 cycles per bytecode (for *db*).

In the Hard-Int architecture the fetching and decoding of the bytecodes are done in hardware (taking two cycles for each bytecode). The cost seen in Figure 7 ranges from 2.2 cycles per bytecode for *mpeg* to around 8.0 cycles per bytecode for *db*. The emulation cost for the JIT ranges from 1.1 cycles per bytecode for *mpeg* to 29.7 cycles per bytecode for *db*.

The cost is low for benchmarks which spend relatively large amounts of time in a few methods, and higher if the execution time is distributed across a lot of methods. The JIT does a good job of exploiting the architectural features using optimizations, but the compilation time is added to the translation overhead. If the compiled code is not executed frequently and more time is spent in compiling methods than executing them, the cost may be higher than when executed using an interpreter. This is observed in Figure 7 for *db* and

Benchmark	JVM	Execution mode	I-Cache		D-Cache	
			References	Cache-misses	References	Cache-Misses
db	JDK 1.1.6	interpreter	13.52M	4304	21.9M	225425
	JDK 1.1.6	JIT	19.37M	125768	27.67M	454032
	JDK 1.2	interpreter	19.51M	77772	35.03M	440581
	JDK 1.2	JIT	17.50M	189801	28.69M	615089
	Hard-Int		7.62M	35820	11.91	299405
javac	JDK 1.1.6	interpreter	31.01M	14058	53.71M	596469
	JDK 1.1.6	JIT	41.82M	265825	61.16M	1.08M
	JDK 1.2	interpreter	35.82M	205450	61.31M	979362
	JDK 1.2	JIT	31.81M	456543	48.07M	1.24M
	Hard-Int		13.2M	54169	17.62M	439674
jess	JDK 1.1.6	interpreter	40.36M	17012	73.54M	791518
	JDK 1.1.6	JIT	29.48M	140802	43.09M	885864
	JDK 1.2	interpreter	55.79M	264831	107.4M	1.53M
	JDK 1.2	JIT	42.43M	634877	67.23M	1.88M
	Hard-Int		10.10M	36659	11.98M	314735
mpeg	JDK 1.1.6	interpreter	206.73M	12314	451.62M	511866
	JDK 1.1.6	JIT	37.23M	197381	91.45M	765343
	JDK 1.2	interpreter	222.47M	125611	485.7M	917192
	JDK 1.2	JIT	37.7M	302112	110.68M	1.01M
	Hard-Int		36.03M	77292	54.24M	522781
mtrt	JDK 1.1.6	interpreter	218.35M	17162	487.38M	5.15M
	JDK 1.1.6	JIT	162.69M	172562	229.64M	5.56M
	JDK 1.2	interpreter	189.17M	105299	502.25M	4.06M
	JDK 1.2	JIT	81.9M	266588	164.41M	4.50M
	Hard-Int		17.82M	76356	58.21M	518483

Table 5: Cache Performance for the SPECjvm98
This table shows the number of references and misses for the instruction and data cache. Cache size= 64K bytes, block size= 32 bytes, I-cache is 2-way and D-cache is 4-way set-associative. M - indicates million.

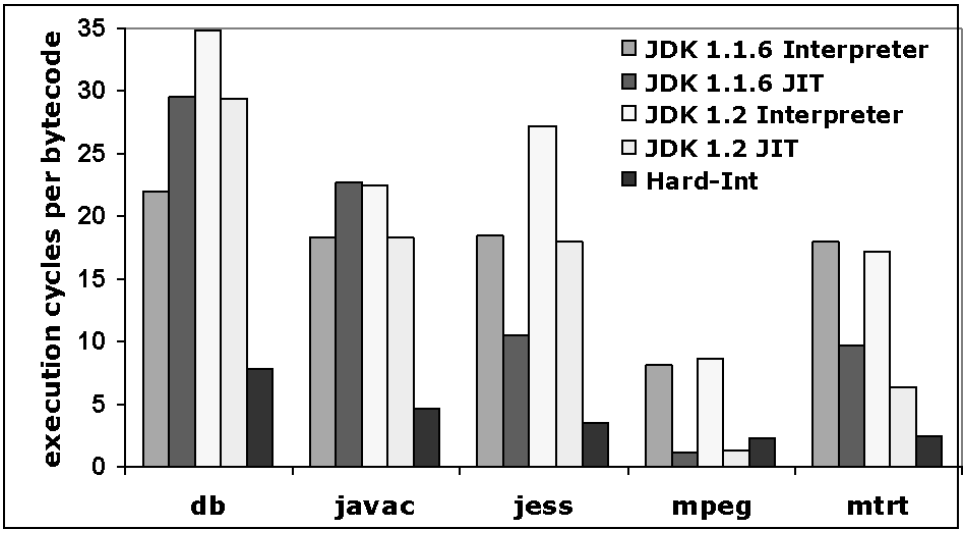

Figure 7: Cycles executed per bytecode on a 4-way machine

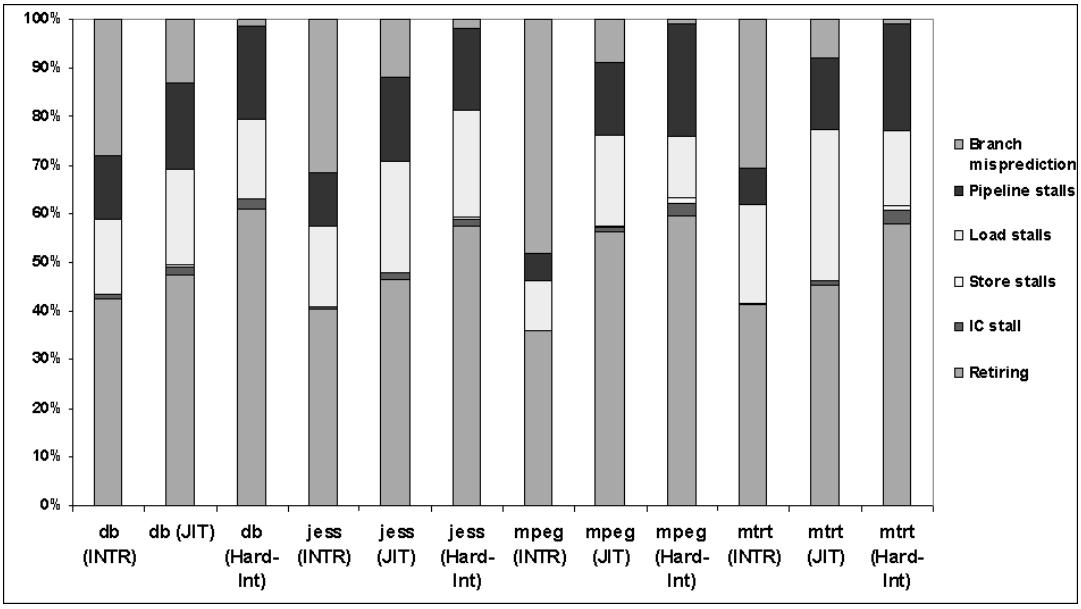

Figure 8: Hard-Int IPC Breakdown for JDK 1.2

javac when executed using the JDK 1.1.6 interpreter and JIT.

Benchmark	References	Hits	Hit Rate
db	13.28M	927392	69.79%
javac	3.86M	2.86M	74.32%
jess	5.40M	4.39M	81.34%
mpeg	72.97M	55.77M	76.42%
mtrt	34.94M	19.11M	54.70%

Table 6: Translated code buffer performance This table shows the number of references and hits for the translated code buffer. Buffer size is 64 entries and 4-way set-associative. `M indicates million.`

4.5 Performance bottlenecks

Figure 8 graphically illustrates the retire-time bandwidth for each benchmark. In the 4-way configuration, four instructions can potentially be retired each cycle. This figure breaks down the utilization of these retire slots. The bottom section of the column is the percentage of retire slots that are filled with instructions that successfully retire. The section above that represents the percentage of slots that are empty due to instruction cache misses. The next portion of the bar (and generally the smallest for these benchmarks) is due to store instructions that can not progress due to memory resource contentions. The load stalls segment is above the store stalls. Loads stalls are caused by load instructions that are currently in the process of loading or can not progress due to memory resource contentions. The slice second from the top is due to pipeline stalls. These instructions are detained because the pipeline is backed up and they have yet to execute. The top section of the column represents retire-time bandwidth lost due to branch mispredictions. When a branch mispredicts, no other valid instructions can reach the retire slots until the branch has resolved and a new fetch has occurred.

The top slice of the breakdown, which shows the stalls due to branch misprediction is the most striking difference between the different execution modes. The performance of the branch predictor is fairly consistent across the different modes, but it is the number of indirect branches and BTB misses which account for the difference. The interpreter executes a large number of indirect branches to implement the decode loop and large number of BTB misses occur when predicting the targets. This effect is mitigated in the JIT but not completely eliminated. The Hard-Int does not execute a branch to decode the bytecode and branch misprediction forms a small percent of the retire bandwidth. Stalls due to store waiting are more prominent for the Hard-Int architecture. Benchmarks *mtrt* and *mpeg* show up to 1.3% stalls due to stores that could not complete. This is due to memory operations becoming a more prominent percentage of the instruction mix in the Hard-Int architecture. The percentage of load stalls are similarly higher in the Hard-Int architecture compared to the interpreter. The load stalls are seen to be fairly large across all three execution modes and is the worst for the JIT, where the stall is as high as 32% in *mtrt*. The pipeline stalls are more of a bottleneck in performance for the Hard-Int architecture since other stalls are not as prominent in the other execution modes. It is evident that eliminating BTB misses mode would result in substantial performance improvement for the interpreter. Load stalls are seen to limit the JIT performance. In the case of the Hard-Int architecture, it is mainly memory performance that is the bottleneck.

5. RELATED WORK

Several language specific processors have been designed in the past. These processors would show considerable speedup over a general purpose processor for languages which they target. In a very specialized processor the assembly language is the high level language and no semantic gap exists. The disadvantages of such a processor is that it requires

complex hardware compared to conventional machines, and it is also very inflexible. The Pasdec architecture [17] and the SYMBOL machine [17] are examples of such specialized processors. Architectures which provide support for a language by raising the level of the machine language also exist. Ideally for such machines, there exists a one to one correspondence between the high level language and the machine instructions. and a compiler translates each instruction. The DELtran [18] and Scheme-79 chip [19] are examples of such architectures.

Past research also includes a large body of work towards providing architectural support for efficient Java execution. Java processors such as picoJava [20] from Sun Microsystems, JEM1 [21] from Rockwell Inc, and Patriot Scientific corporations PSC1000 [22] used in the embedded market are examples of architectures providing support for direct execution of Java bytecodes. Hardware support for Java stack processing, object manipulation and method invocation was proposed by Vijaykrishnan et al. [23, 24, 25]. Dynamic translation of bytecodes to the DELFT-JAVA RISC instruction set and a link translation buffer to help in dynamic method invocation was proposed by Glossner et al. [26, 27]. The MAJC (Microprocessor Architecture for Java) chip [28] proposed by Sun Microsystems uses thread level parallelism and speculation to improve performance of Java applications. Java methods are mapped to threads and executed speculatively to reduce execution time.

The JSTAR accelerator [29] recently announced by Nazomi Communications is a Java coprocessor for general purpose embedded processors. JSTAR is a coprocessor that interfaces to the native microprocessor core and its cache or memory subsystem. JSTAR fetches bytecodes from memory and executes them in conjunction with the native processor. A speedup of 5.5 over an interpreter was achieved when on a MIPS R3000 processor core [29]. While available literature mentions that JSTAR performs translation using hardware, details on the architecture or implementation have not been revealed.

The HotShot architecture from Chicory Systems [30] uses a hardware engine to compile and optimize Java code. The hardware engine is not a coprocessor or part of the processor core. Instead, it is implemented as a standard peripheral device. The hardware engine performs optimizations that include branch elimination, branch folding, loop transformations, etc., similar to software compilers. Technical details about the implementation of the HotShot architecture have not been published. However, we speculate that their approach is similar to the one proposed in this paper. The main difference between HotShot and Hard-Int is that we implement the hardware engine as a part of the microprocessor core, whereas HotShot implements it as a peripheral chip.

Kent and Serra [31] study the feasibility and design issues related to a hardware/software codesign of the JVM. They propose a coprocessor implemented in FPGA, that will work in unison with a general purpose micro-controller to increase Java performance. This is similar to the approach we propose in this paper. Their implementation includes a software partition (which supports the execution of the Java coprocessor) for performing the object-oriented operations, and a hardware partition which implements the other instructions that exist in the JVM. Their work talks about the design issues involved in performing some of the tasks of the JVM in hardware (to improve performance) however no performance results are provided.

6. SUMMARY AND CONCLUSIONS

The primary bottleneck in Java execution is the overhead of translation or mapping. In interpretation, it is the overhead of software fetch and decode. In JITs, it is the overhead of (i) performing compilation and compiler optimizations at run time, (ii) installing the compiled code into data caches/memory at run time, and (iii) using extra memory for the JIT itself. The JIT overheads are more prominent for short running, client-like workloads, where the execution time is not concentrated in a small number of methods. Using additional hardware in general purpose processors to perform the mapping actions lowers the overhead and improves performance of client-like workloads. We proposed the Hard-Int architecture which dynamically translates bytecodes to native machine instructions using a hardware translation unit and a translated code buffer. The hardware translation unit fetches the bytecodes from the instruction cache and predecodes them before performing the translation using the microcoded ROM. We compared the performance of SPECjvm98 benchmarks when executed using software translation (interpreter and JIT) to hardware translation using the Hard-Int architecture.

We summarize the important observations from the performance study below:

- Hardware translation results in an average[2] speedup over the JDK 1.2 (JDK 1.1.6) interpreter of 5.43 (4.39) on a 4-way processor. The average speedup attained in a 16-way processor is 5.07 (3.95).
- The Hard-Int architecture performs consistently better than a JIT compiler for client workloads, showing an average speedup over the JDK 1.2 (JDK 1.1.6) JIT compiler of 2.59 (2.58) on a 4-way machine. The average is significantly affected by *mpeg*, which is the only program that shows a performance degradation. Excluding *mpeg*, the average speedup is 3.76 (3.87) over the JIT compiler on a 4-way machine. On a 16-way machine, the Hard-Int architecture shows a speedup of 2.26 (2.07) over the JIT compiler. With *mpeg* excluded, the average speedup obtained is 3.35 (3.18) on a 16-way machine.
- The emulation cost per bytecode varies from 2.2 cycles per bytecode to 7.8 cycles per bytecode for the SPECjvm98 benchmarks when executed on a 4-way processor using the Hard-Int architecture. In the interpreter the cost ranged from 8.1 to 34.9 cycles per bytecode, and from 1.1 to 29.7 cycles per bytecode in a JIT.
- When analyzing performance bottlenecks, we find that the cache performance of the Hard-Int is better than

[2]all averages for speedup use the geometric mean

the JIT. This is because the translated code is not written to memory, and therefore write misses in the data cache and read misses in the instruction cache are not incurred as frequently. In addition, Hard-Int results in less speculative execution due to fewer branch direction and target mispredictions.

As certain workloads become dominant general purpose computer architectures have added support to execute them efficiently. This was the case with numerical coprocessors being integrated into the processor core as more transistors could fit onto a single die. MMX technology was introduced when media workloads became a dominant part of the desktop workloads. Java technology is emerging as a force in the software industry and Java workloads are becoming an important part of general processor workloads. Current execution modes of Java using software translation suffer from poor performance compared to compiled native code. Providing hardware support in general purpose processors is one way to bridge this performance gap between the execution of Java applications and natively compiled applications. The scheme proposed in this paper shows that this performance gap can be greatly reduced using the Hard-Int architecture support.

ACKNOWLEDGMENTS

This research is supported in part by the National Science Foundation under CAREER Award CCR-9796098 and grant EIA-9807112. The authors also wish to acknowledge the research support from the State of Texas Higher Education Coordinating Board under ATP grant #403, and from Corporations such as IBM, Tivoli, Sun Microsystems, Intel, Microsoft, AMD and Dell.

7. REFERENCES

[1] F. Y. T. Lindholm, *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[2] A. Krall and R. Grafl, "CACAO- a 64 bit JavaVM Just-In-time Compiler," in *Concurrency: Practice and Experience, 9(11):1017-1030*, 1997.

[3] A.-R.Adl-Tabatabai, M.Ciernaki, G.-Y.Lueh, V.M.Parikh, and J.MStichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler," in *Proceedings of Conference on Programming Language Design and Implementation*, pp. 280–290, 1998.

[4] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, " Compiling Java Just In Time ," *IEEE Micro*, vol. 17, pp. 36–43, May-June 1997.

[5] HotSpot: A New Breed of Virtual Machine, http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html?030998.

[6] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler," *IBM Systems Journal*, vol. 39, no. 1, pp. 175–194, 2000.

[7] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, and H. Srinivasan, "The Jalapeo Dynamic Optimizing Compiler for Java," in *ACM Java Grande Conference*, pp. 129–141, June 1999.

[8] The SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/.

[9] R. Radhakrishnan, N. Vijaykrishnan, L. K. John and A. Sivasubramanium , "Architectural Issues in Java Runtime Systems," in *Proceedings of the Intl. Symposium on High Performance Computer Architecture (HPCA-6)*, pp. 387–398, January 2000.

[10] R. Radhakrishnan, J. Rubio, and L. John, "Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level," in *Proceedings of International Conference on Computer Design*, October 1999.

[11] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, , and J. Sabarinathan, "Java Runtime Systems: Characterization and Architectural Implications," *IEEE Transactions on Computers*, pp. 131–146, Feb 2001.

[12] T. Shanley, *Pentium Pro and Pentium II System Architecture*. Addison-Wesley, 1998.

[13] E. Debaere and J. Campenhout, *Interpretation and Instruction Path Coprocessing*. MIT Press, 1990.

[14] Y. Chou and J. P. Shen, "Instruction Path Coprocessors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 270–281, June 2000.

[15] J. E. Smith, " Decoupled Access/Execute Computer Architecture," in *ACM Transactions on Computer Systems*, pp. 289–308, November 1984.

[16] Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling, SMLI TR-93-12," tech. rep., Sun Microsystems Inc, 1993.

[17] A. Silbey, V. Milutinovic, and V. Mendoza-Grado, "A Survey of Advanced Microprocessors and Hll Computer Architectures," vol. 19, pp. 72–85, 1986.

[18] M. J. Flynn and L. W. Hoevel, "Execution Architecture: The Deltran Experiment," vol. C-32, pp. 156–175, 1983.

[19] G. J. Sussman, J. Holloway, J. lxx Steel, and A. Bell, "Scheme-79 Lisp on a Chip," vol. 14, pp. 10–21, 1981.

[20] J. Michael O'Connor and M. Tremblay, "Picojava-I: The Java Virtual Machine in Hardware," *IEEE-MICRO*, vol. 17, pp. 45–53, Mar/Apr 1997.

[21] A. Wolfe, " First Java-specific chip takes wing ," *Electronic Engineering Times*, 22 April 1997. http://www.techweb.com/.

[22] R. B. Slack, "A Java chip available now," *Gamelans Java Journal*, April 1999. http://softwaredev.earthweb.com/java.

[23] N.Vijaykrishnan, N.Ranganathan, and R.Gadekarla, "Object-Oriented Architectural Support for a Java Processor," in *Proceedings of ECOOP'98, the 12th European Conference on Object-Oriented Programming*, pp. 330–354, 1998.

[24] N.Vijaykrishnan, *Issues in the Design of a Processor Architecture*. PhD thesis, University of South Florida, 1998.

[25] N. Vijaykrishnan and N. Ranganathan, "Tuning Branch Predictors to Support Virtual Method Invocation in Java," in *Proc. of COOTS'99*, pp. 217–228, May 1999.

[26] J. Glossner and S. Vassiliadis, "The Delft-Java Engine: An Introduction," in *Proceedings of the Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pp. 766–770, August 1997.

[27] J. Glossner and S. Vassiliadis, "Delft-Java Link Translation Buffer," in *Proceedings of the 24th EUROMICRO conference (EuroMicro 98)*, pp. 221–228, August 1998.

[28] M. Tremblay, "An Architecture for the New Millenium," in *Proceedings of Hot Chips 11*, August 1999.

[29] H. Shiffman, "JSTAR: Practical Java Acceleration For Information Appliances." http://www.nazomi.com/.

[30] Chicory Systems, "A Comparison of Java Acceleration Technologies." White Paper, Dec 2000.

[31] K. B. Kent and M. Serra, "Hardware/Software Co-Design of a Java Virtual Machine," in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, June 2000.