Pipelined Java Virtual Machine Interpreters

Jan Hoogerbrugge and Lex Augusteijn

Philips Research Laboratories,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
{jan.hoogerbrugge,lex.augusteijn}@philips.com

Abstract. The performance of a Java Virtual Machine (JVM) interpreter running on a very long instruction word (VLIW) processor can be improved by means of pipelining. While one bytecode is in its execute stage, the next bytecode is in its decode stage, and the next bytecode is in its fetch stage. The paper describes how we implemented threading and pipelining by rewriting the source code of the interpreter and several modifications in the compiler. Experiments for evaluating the effectiveness of pipelining are described. Pipelining improves the execution speed of a threaded interpreter by 19.4% in terms of instruction count and 14.4% in terms of cycle count. Most of the simple bytecodes, like additions and multiplications, execute in four cycles. This number corresponds to the branch latency of our target VLIW processor. Thus most of the code of the interpreter is executed in branch delay slots.

1 Introduction

One of the most important properties of the Java Platform is its ability to support mobile code, i.e., code that runs on a large variety of machines, often transfered over a network [1]. This is realized by an object file format called *class files* which contains machine instructions called *bytecodes* for a virtual machine called Java Virtual Machine (JVM). JVM is designed so that it can be efficiently implemented on a large variety of platforms. JVM can be implemented directly in hardware as a JVM processor [2], by means of a software interpreter, or by means of a just-in-time compiler [3,4] which compiles bytecodes to native machine code just before the code is executed and caches it for future execution.

A JVM processor can potentially provide excellent execution speed performance, but they usually cannot execute other software that has to run on the system such as video and audio decoders. Such software would have to be ported to the JVM processor.

A software interpreter is usually slow, about 5-20 times slower than native code, but it is portable and has low memory requirements for both code and data.

A just-in-time compiler could approach the execution speed of optimized native code but has high memory requirements. A just-in-time compiler will be larger than an interpreter, and require RAM to store generated native code. A just-in-time compiler is also more complex to develop, and is not portable. Furthermore, just-in-time compilation requires an investment in compilation time

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 35–49, 2000.

which will only be profitable when the code is executed repeatedly. A comparison between JVM processors, interpreters, and just-in-time compilers for JVM can be found in [5].

In this paper we describe a JVM interpreter for the Philips TriMedia very long instruction word (VLIW) media processor, which is targeted for embedded systems in consumer products [6,7]. For these products a dedicated JVM processor, unable to perform media processing, will not be cost-effective. Furthermore, the memory requirements of a just-in-time compiler are expected to be too high, especially for VLIWs, which require instruction scheduling and have a low native code density. That is why we have opted JVM interpreter for TriMedia.

State-of-the-art compiler technology is not able to sufficiently exploit instruction-level parallelism (ILP) in a JVM interpreter. All steps required to execute a bytecode – fetch, decode, and execute – are dependent on each other, which results in low ILP. Although there is little ILP within a bytecode, there is much more ILP between successive bytecodes. Exploiting this implies pipelined execution of the interpreter, a well-known technique employed in all performance-oriented processors. Although software pipelining [8] has been well-known for many years, the algorithms developed for it are not able to pipeline the complex, irreducible [9] control flow structure of an interpreter. In this paper we will show how we managed to pipeline a JVM interpreter with modest compiler support.

This paper is organized as follows. Section 2 provides background information on JVM and TriMedia. Section 3 describes pipelining and how we implemented it. Section 4 describes the experiments we conducted. Section 5 discusses related work. Finally, section 6 ends the paper with conclusions.

2 Background

In this section we give a brief description of JVM and TriMedia, necessary to understand the remainder of the paper.

2.1 Overview of JVM

The Java Virtual Machine is a stack-based architecture, which means that all common instructions such as additions and multiplications get their operands from the evaluation stack and put their result on the evaluation stack. An array of local variables is available for fast accessible storage of temporary scalar data. Load and store instructions move data between the evaluation stack and the local variables. The memory of the JVM consists of a heap for dynamic storage of objects and arrays. Instructions are available for allocation of objects and arrays, and for accessing and modifying them. Various kinds of control flow instructions are available for conditional branching and method invocation.

Whereas real machines have relatively simple instructions, JVM has both simple as well as several very complex instructions. Examples of the latter are

instructions for memory allocation, thread synchronization, table jumps, and method invocation.

The instructions of JVM are called bytecodes. A bytecode consists of a one-byte opcode followed by zero or more operands. Each operand consists of one or more bytes. This structure, unlike the structure of a real processor, makes JVM easy to implement by a software interpreter.

2.2 The TriMedia VLIW Media Processor

The TriMedia VLIW media processor[1] is a five-issue VLIW processor with an operation set and peripherals optimized for the processing of audio, video, and graphics. 28 functional units are available; among which are 2 load/store units, 3 jump units, 5 ALUs, and 4 floating point units. Each instruction holds up to five operations that can be issued to a subset of the 28 functional units under certain combination restrictions. All functional units are connected to a central register file of 128 32-bit registers. The processor has no interlocks and stalls in cache misses. TriMedia has a 16 kbyte data cache and a 32 kbyte instruction cache.

All multi-cycle functional units, except the divide functional unit, are fully pipelined. The load/store units have a latency of three cycles, and the jump units have a latency of four cycles. A jump latency of four cycles means that three delayed instructions are executed before a jump operation becomes effective. Fifteen operations can be scheduled in these three delayed instructions and another four operations can be scheduled in parallel with the jump operation. For more information on TriMedia the reader is referred to [6,7].

3 Pipelined Interpreters

A JVM interpreter is a complex piece of software, usually written in (ANSI) C. In our case, we used both the freely available Kaffee [10] and Sun Microsystems' Personal Java [11] for the application of our pipelining techniques. In fact, these techniques are generally applicable to all types of interpreters, and not just to JVM interpreters.

We assume that an interpreter is written in C and has the following structure:

```
bytecode bc;              /* Current bytecode */
bytecode *pb;             /* Bytecode pointer */

for (;;) {
  bc = *pb;               /* Fetch */
  switch (bc) {           /* Decode */
    case ADD:
       ... = ... + ...    /* Execute bytecode */
```

[1] Whenever we mention TriMedia in this paper, the reader should read TriMedia TM1000 or TM1100. Future generations may have a different architecture.

```
        pb += 1;          /* Increment bytecode pointer */
        break;
      ...                 /* Other cases */
    }
  }
```

An optimizing C compiler will implement the switch statement by a table jump including a range check to test whether the value is in the range of the table. Assuming that the last case of the switch is the default one, this gives a control-flow graph as shown in figure 1a for such an interpreter.

3.1 Threaded Interpreters

An indirect threaded[2] interpreter [12,13], is an interpreter with which the code to implement a bytecode jumps directly to the code to implement the next byte to be executed, rather than to a shared join point. So each bytecode performs a table jump to its successor. Such a threaded interpreter exhibits the control flow shown in figure 1c.

We have modified the TriMedia C/C++ compiler so that it can transform the control flow graph of figure 1a into that of figure 1c. This transformation comprises of two steps.

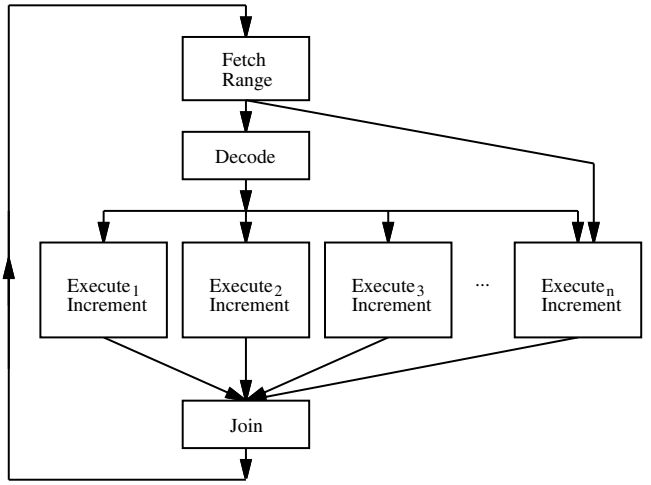
Removal of the range check. If the switch expression is of type `unsigned char` and the number of cases is large enough (≥ 128), the switch statement is made complete by adding the missing cases and removing the default case. This results in a 256-entry jump table and the elimination of the range check. The resulting control flow graph is shown in figure 1b.

Code duplication. The switch block S in the control flow graph of figure 1b and the join block J are duplicated for each of the 256 cases. This effectively yields the threaded interpreter. The compiler is instructed by pragmas at the end of each switch case where to perform this code duplication. As the blocks involved contain only a jump (for J) (which disappears) and two loads and a table jump (for S), the resulting increase in code size is limited. Note that the 256 jumps to J disappear.

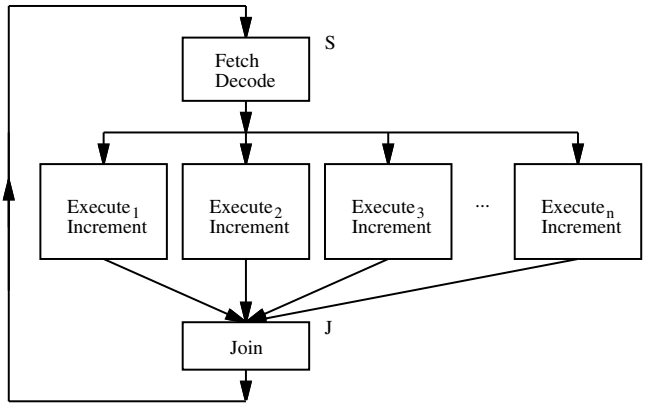
Another way to create a threaded interpreter is to use first-class labels, as found in GNU C. In this method, each block ends with a computed goto to the next block. However, first-class labels are not part of ANSI C and are not supported by the TriMedia compiler.

The control flow graph of a threaded interpreter as shown in figure 1c is irreducible [9]. This means that there is no single loop header and control flow edges cannot be partitioned into forward and backward edges such that the forward edges form an acyclic graph and heads of backward edges dominate their tails. The authors are not aware of software pipelining algorithms that can handle irreducible control flow graphs. Therefore, we have developed our own technique for pipelining threaded interpreters.

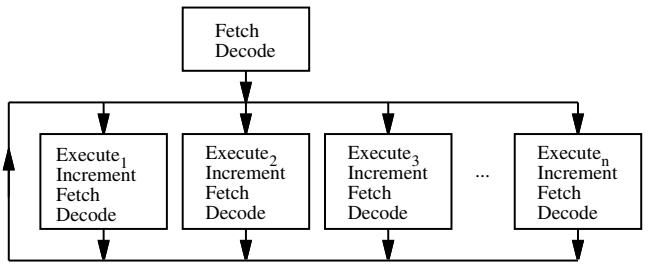
[2] In the remainder of the paper we will speak about a threaded interpreter whenever we mean an indirect threaded interpreter.



(a) Control flow graph of a table–jump–based interpreter



(b) Control flow graph after range check removal



(c) Control flow graph after duplication of Fetch and Decode

Fig. 1: Control flow graphs of bytecode interpreters

3.2 Pipelined Interpreters

In the threaded interpreter shown in figure 1c the execute$_i$ and the increment/fetch/decode (IFD) parts are usually independent, because the increment value of the bytecode pointer is a constant for most bytecodes, jumps being the exception. The IFD part is expressed in pseudo C as:

```
pb += n;          /* Increment */
bc = *pb;         /* Fetch next bytecode */
t = table[bc];    /* Fetch target address from jump table */
goto t;           /* Jump to next bytecode */
```

This consists of a sequence of two loads and a jump, which can be executed parallel to the execute part of the bytecode itself on a VLIW processor like the TriMedia. For many simple bytecodes, the latency is determined by the IFD part. Therefore, a key issue in the speeding up of an interpreter on a VLIW is the decrease of this latency. The solution is to pipeline the two loads, i.e. `*pb` and `table[bc]`, and the jump. The first of these loads is under the control of the programmer, so we obtain its pipelining by restructuring the source of the interpreter. The second load is compiler-generated from the switch statement. We have modified the compiler to take care of its pipelining. Without pipelining, the latency of the IFD part is *2*load+jump = 2*3+4 = 10* cycles on a TriMedia.

The pipelining can also be explained by means of the following diagrams. Figure 2a shows the sequential execution of bytecodes. The figure illustrates that the execution time of a bytecode is determined by the IFD part, which is executed in parallel with the execute part. Figure 2b shows the pipelined execution. The pipeline comprises three stages: increment/fetch, decode, and execute/jump. While bytecode i is in its execute/jump stage, bytecode $i + 1$ is in its decode stage, and bytecode $i + 2$ is in its fetch stage.

Pipelining the Bytecode Fetch The bytecode fetch is pipelined by fetching bytecodes in advance and keeping these prefetched bytecodes in local variables which will be allocated to machine registers. Because the pipeline has to be two stages deep (we need to pipeline two loads), the bytecode must be fetched far enough in advance. Because bytecode instructions are of variable length, several bytes may need to be fetched in advance. The majority of the bytecodes have a length of at most three bytes. Since the bytes must be fetched two bytecode executions in advance, we decided to fetch a total of $2 * 3 = 6$ bytes in advance. We depict them by `b1` through `b6`.

The pipelined interpreter code looks like this, for an instruction length of two bytes.

```
pb += 2;          /* Increment */
bc = b2;          /* Fetch next bytecode */
b1 = b3; b2 = b4; b3 = b5; b4 = b6; b5 = pb[5]; b6 = pb[6];
t = table[bc];    /* Fetch target address from jump table */
goto t;           /* Jump to next bytecode */
```

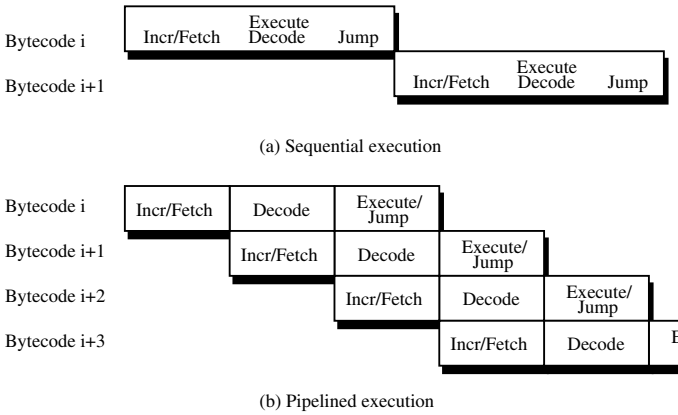


Fig. 2: Sequential vs. pipelined bytecode interpretation. In the sequential case (figure 2a), the operations of the IFD part are dependent on each other while the IFD part can be executed in parallel with the execute part. Figure 2a illustrates this.

The result of this pipelining is that the decode load (of the jump target) is no longer dependent on any bytecode fetch. The latency of the IFD part is thus reduced to $load{+}jump = 7$ cycles on a TriMedia.

Pipelining Decode Load Recall that the decode load of the branch target `table[bc]` is compiler-generated. The compiler can move this load from each of the bytecode blocks into each of its predecessors. Since they are each others' predecessors, each bytecode block gets 256 of these decode loads. Fortunately, they are all equal apart from a difference in the instruction length `n`. So, for each value of `n`, a version is copied to each predecessor block. In practice, this is limited to 3 versions. This corresponds to the speculated decoding of bytecodes due to the variable instruction length.

The result in pseudo C looks like this:

```
pb += 2;          /* Increment */
bc = b2;          /* Fetch next bytecode */
b1 = b3; b2 = b4; b3 = b5; b4 = b6; b5 = pb[5]; b6 = pb[6];
t  = t2;
t1 = table[b1];  /* Fetch target address 1 from jump table */
t2 = table[b2];  /* Fetch target address 2 from jump table */
t3 = table[b3];  /* Fetch target address 3 from jump table */
goto t;          /* Jump to next bytecode */
```

This code can be optimized by combining the shifting of the bytecodes with the table loads, so that they can be shifted too. To this end, the compiler per-

forms an extensive analysis to prove that `t1` always equals `table[b1]`, etc. It then transforms the code as follows:

```
pb += 2;          /* Increment */
bc = b2;          /* Fetch next bytecode */
b1 = b3; b2 = b4; b3 = b5; b4 = b6; b5 = pb[5]; b6 = pb[6];
t  = t2;
t1 = t3;          /* Shift target address 1 */
t2 = table[b2];   /* Fetch target address 2 from jump table */
t3 = table[b3];   /* Fetch target address 3 from jump table */
goto t;           /* Jump to next bytecode */
```

The net result of this pipelining process is that, on a VLIW, a simple bytecode can start with a jump to its successor, and perform its actual computation in parallel with the jump. This reduces the minimum latency of a bytecode on a TriMedia to 4 cycles, the value of the branch delay.

Pipelining also increases the required number of registers to store the variables, and increases the code size of the interpreter loop. For TriMedia, the interpreter loop uses fewer than 20 registers and 30 kbyte of code. Recall that TriMedia has 128 registers and a 32 kbyte instruction cache.

3.3 Stack Caching

After pipelining, the execution time of the control flow part (IFD) of a bytecode is reduced from the sum of the latencies of two loads and a jump to the maximum of those latencies. In the case of TriMedia, the reduction is from 10 to 4 cycles. After this reduction however, the execute part usually has a longer latency, especially due to the evaluation stack accesses in a Java interpreter, which are always memory accesses.

This latency is easily reduced by caching the top s elements of the stack in local variables, which are mapped onto registers by the compiler. The best value for s is a trade-off between limiting the number of memory accesses to the evaluation stack on the one hand, and minimizing the register pressure and book-keeping operations relating to pipelining and stack-caching on the other, because driving these techniques too far results in too much code to implement it, which fills up the available issue slots. In practice, a value of $s = 2$ performs best on a TriMedia.

The following C code illustrates stack caching for the `iadd` bytecode, which takes two integers from the stack, adds them, and puts the result back on the stack:

```
tos = tos + nos;  /* Add top two elements on stack */
nos = sp[2];      /* Fetch new data into stack cache */
sp += 1;          /* Update stack pointer */
...               /* IFD part */
```

The two local variables holding the top two stack elements are called `tos` (top of stack) and `nos` (next on stack). Unlike in previously published stack

caching techniques [14], the primary goal of stack caching for us is to reduce the dependence height in the execute part of the bytecode and not to reduce the number of operations. Unlike [14], we do not maintain how many cache elements are valid by means of interpreter state. In our case, `tos` contains valid data when there is at least one element on the stack, and `nos` contains valid data when there are at least two elements on the stack.

After execution of a bytecode that consumes a variable number of stack elements, such as the invoke bytecodes, the cache is refilled by executing:

```
tos = sp[0];       /* load top-of-stack into cache */
nos = sp[1];       /* load next-on-stack into cache */
```

After using pipelining and stack caching, sufficient parallelism has been exposed that can be exploited by the global instruction scheduler of the TriMedia compiler [15].

3.4 Control Flow Bytecodes

In the case of control flow bytecodes, the IFD part is dependent on the execute part. In the case of a non-taken jump the pipeline is updated normally, but in the case of a taken jump the pipeline has to be flushed and has to be restarted at the jump target. The result is that taken jumps require more execution time than non-taken jumps. Pipelined processors without delay slots or branch prediction exhibit the same behavior in this respect.

3.5 Example TriMedia Assembly Code

Figure 3 shows the TriMedia assembly code for the `iadd` bytecode. It was produced by our compiler and slightly modified to improve clarity. Figure 3 shows four instructions, each containing five operations, in which empty issue slots are filled with `nop` operations. Immediate operands appear between parentheses. Register moves are implemented by an `iadd` operation with `r0`, containing a hard-wired zero value, as first operand.

Note that the jump operation is scheduled in the first instruction of the block and that the other operations are scheduled in parallel with the jump or in the three delay slots of the jump operation. To understand the code in figure 3, the reader should be aware that all operations in an instruction are issued in parallel and that the results of three-cycle latency load operations are written to registers three instructions after being issued. For example, the values that are moved from `r14` and `r21` in instruction 3 are not the values that are loaded in instruction 2, but the values that will be overwritten by the two loads, in the cycle after instruction 4.

4 Evaluation

In this section we describe several measurements to evaluate the presented techniques. As our evaluation platform we used Sun's Personal Java [11] running on

```
(* instruction 1 *)
iadd r22 r23 -> r22,        /* tos = tos + nos */
ijmpt r1 r13,               /* goto t1 */
iadd r0 r16 -> r9,          /* b1 = b2 */
ld32d(4) r12 -> r23,        /* nos = sp[1] */
nop;

(* instruction 2 *)
iadd r0 r17 -> r16,         /* b2 = b3 */
iadd r0 r10 -> r17,         /* b3 = b4 */
iadd r0 r20 -> r10,         /* b4 = b5 */
ld32x r18 r20 -> r14,       /* t3 = table[b4] */
uld8d(6) r11 -> r21;        /* b6 = pb[6] */

(* instruction 3 *)
iadd r0 r15 -> r13,         /* t1 = t2 */
iadd r0 r14 -> r15,         /* t2 = t3 */
isubi(4) r12 -> r12,        /* decrement stack pointer */
iadd r0 r21 -> r20,         /* b5 = b6 */
nop;

(* instruction 4 *)
iaddi(1) r11 -> r11,        /* increment bytecode counter */
nop, nop, nop, nop;
```

Fig. 3: TriMedia assembly code for the `iadd` bytecode

top of the pSOS real-time operating system. Furthermore, we used a cycle-true simulator of TriMedia that is configured for 128 MByte of main memory. We use SPEC JVM98 for benchmarking. The seven benchmarks of SPEC JVM98 are listed in table 1.

4.1 Execution Speed

First we are interested in the execution times of individual bytecodes. We determined this by inspecting the TriMedia assembly code. Table 2 shows the number of VLIW instructions required to execute a bytecode. In the absence of cache misses, the number of VLIW instructions corresponds to the number of machine cycles required to execute the bytecode. Since pipelining is intended to reduce the interpretation overhead of simple bytecodes, we only list the simple bytecodes in table 2. Long integer and double precision floating point are not supported by the TriMedia architecture and are implemented by library calls. They are therefore regarded as complex bytecodes and have consequently not been included in table 2.

Table 2 clearly shows that most simple bytecodes are executed by four VLIW instructions or close to it. Taken jumps require more cycles because the inter-

Benchmark	Description	Input description	Bytecodes
_201_compress	LZH data compression	SPEC JVM98 -s1 option	27,993,191
_202_jess	Expert shell system	SPEC JVM98 -s1 option	8,151,873
_209_db	Data management	SPEC JVM98 -s1 option	1,773,230
_213_javac	The JDK Java compiler	SPEC JVM98 -s1 option	6,151,375
_222_mpegaudio	MPEG-3 audio decode	SPEC JVM98 -s1 option	115,891,206
_227_mtrt	Raytracer	SPEC JVM98 -s1 option	33,842,764
_228_jack	Parser generator	SPEC JVM98 -s1 option	176,706,788

Table 1: Benchmarks used for evaluation

Bytecode type	Instructions	Example bytecodes
Simple arithmetic	4	iadd, fadd, isub, fsub, imul, fmul, ineg, fneg, ishl
Taken/non-taken jump	15/7	ifeq, ifne, iflt, ifge, if_icmpeq, if_icmpne
Load constant	4	aconst_null, iconst_m1, iconst_0, iconst_1, fconst_0
	5	dconst_0, dconst_1, ldc_quick
	6	ldc_w_quick
	8	ldc2_w_quick
Push constant	4	bipush
	6	sipush
Load	5	iload, lload, fload, dload, aload
	4	iload_0, iload_1, iload_2, fload_0, fload_1, fload_2
Store	5	istore, lstore, fstore, dstore, astore
	4	istore_0, istore_1, istore_2, dstore_0, dstore_1
Array load	9	iaload, laload, faload, daload, aaload, baload
Array store	16	iastore, lastore, fastore, dastore, bastore
Stack manipulation	4	pop, pop2, dup, dup_x1, dup_x2, dup2, swap
Local increment	6	iinc
Control flow	12	goto, jsr, goto_w, jsr_w
Field access	7	getfield_quick, putfield_quick
	8	getfield2_quick, putfield2_quick
	21	getstatic_quick, putstatic_quick, getstatic2_quick

Table 2: Overview of execution times of simple bytecodes

preter pipeline has to be flushed and reloaded with bytecodes at the jump target, as described in section 3.4.

Various heap access bytecodes are relatively slow because of the internal data structures used by Sun Personal Java. All accesses to heap objects (class objects as well as array objects) are performed via handles. Although this facilitates heap compaction, an extra indirection, i.e. a load operation, is required to access data.

Static field accesses are relatively slow because each reference requires an expensive test to determine whether static initializers have to be called. Several sequential loads are required to perform this test in Personal Java.

Array stores are also relatively slow. This is caused by the run-time checks that have to be performed (null pointer check and bound check) and by the fact that the array reference operand is the third stack operand which is not cached in the stack cache. This operand is the most time-critical of the three operands of array store bytecodes (array reference, index, and data to be stored). A stack cache of three elements will improve the speed of array stores, but several other bytecodes are likely to take more time.

4.2 Individual Contributions of Different Techniques

In order to measure the individual contributions of the techniques described in section 3, we compiled the JVM in three different ways: (1) no threading and pipelining, (2) threading but no pipelining, and (3) threading and pipelining. In all three cases stack caching was enabled. Table 3 shows the instruction count reduction: on average, threading reduces the instruction count by 16.6%, pipelining gives a further reduction of 19.4% over the threaded version. The combined reduction is 32.5%. Furthermore, table 4 shows the cycle count reduction: on average, threading reduces the cycle count by 14.4%, pipelining gives a further reduction of 14.6% over the threaded version. The combined reduction is 26.4%.

The reduction in cycle count is obviously less than the reduction in instruction count because threading and pipelining does only reduce the number of compute cycles and not the number of stall cycles where the processor is waiting on the memory. In fact, threading and pipelining may increase the number of memory cycles because they increase the code size of the interpreter, which leads to more instruction cache misses, and pipelining results in speculative decoding, which causes more data cache references and misses.

The results listed in tables 3 and 4 show a clear correlation between the effectiveness of threading and pipelining. These techniques are most effective for applications in which the majority of the execution time is spent by interpretation of the simple bytecodes. Profile analysis shows that this is indeed the case: _202_jess, _209_db, _213_javac, and _227_mtrt spend a lot of their time in complex bytecodes, the run-time system, garbage collector, and long integer arithmetic emulation code.

5 Related Work

Ertl suggests in his thesis to move part of the dispatch code of the interpreter to earlier bytecodes for machine with sufficient instruction level parallelism [16]. This is a first step into the direction of pipelined interpreters which results in a pipeline of two stages.

Proebsting proposes superoperators to reduce the interpretation overhead [17]. Several of operators are combined together into one superoperator. The interpretation overhead is shared between several operations.

The same idea is used by Piumarta and Riccardi with the difference that their interpreter is dynamically extended with superoperators [18]. They implemented

Benchmark	Threading off/ Pipelining off	Threading on/ Pipelining off		Threading on/ Pipelining on	
_201_compress	548,294,965	419,622,175	(23.4%)	291,935,454	(30.3%)
_202_jess	315,140,012	278,841,909	(11.5%)	242,821,721	(12.9%)
_209_db	64,614,881	56,152,260	(13.0%)	48,126,498	(14.2%)
_213_javac	235,200,152	206,235,014	(12.3%)	178,900,872	(13.2%)
_222_mpegaudio	2,785,393,145	2,157,813,532	(22.5%)	1,623,357,369	(24.7%)
_227_mtrt	1,319,173,260	1,163,465,508	(11.8%)	1,023,474,854	(12.0%)
_228_jack	3,676,668,870	2,882,897,278	(21.5%)	2,056,280,297	(28.6%)
Average			(16.6%)		(19.4%)

Table 3: The effects of threading and pipelining on instruction count. The instruction count reduction with respect to the previous column is shown between parentheses. The combined reduction of threading and pipelining is 32.5%.

Benchmark	Threading off/ Pipelining off	Threading on/ Pipelining off		Threading on/ Pipelining on	
_201_compress	582,876,139	455,567,117	(21.8%)	334,036,387	(26.7%)
_202_jess	424,461,155	390,515,530	(8.0%)	364,608,239	(6.6%)
_209_db	92,906,063	84,712,746	(8.1%)	78,876,026	(6.9%)
_213_javac	303,094,306	270,041,792	(10.9%)	248,756,893	(7.9%)
_222_mpegaudio	2,860,981,158	2,247,040,013	(21.4%)	1,718,591,626	(23.5%)
_227_mtrt	1,471,970,773	1,319,425,388	(10.4%)	1,195,740,789	(9.3%)
_228_jack	4,135,817,332	3,333,366,670	(19.4%)	2,624,348,279	(21.2%)
Average			(14.4%)		(14.6%)

Table 4: The effects of threading and pipelining on cycle count. The cycle count reduction with respect to the previous column is shown between parentheses. The combined reduction of threading and pipelining is 26.4%.

this by a 'cut-and-past' style of code generation where the code to implement a superoperator is generated by concatenating the code of the elementary operators.

We have also used pipelined interpreter technology for other purposes than JVM. We implemented a code compaction system which compiles code to a very dense bytecode-based application-specific virtual machine [19]. Like the JVM interpreter, we used a three-stage pipeline for this.

In another project, we studied a pipelined interpreter for the MIPS instruction set; using only four pipeline stages it was capable of executing many of the MIPS instructions in six clock cycles on TriMedia without any pre-decoding. The first pipeline stage fetches the MIPS instruction, the second stage maps all opcode fields of a MIPS instruction onto one bytecode, the third stage performs the decode by means of a table lookup, and the fourth stage is the jump/execute stage.

We expect that pipelining of threaded interpreters will be useful for superscalars and explicitly parallel instruction computers (EPIC), such as the IA64 architecture [20], as well. In case of IA-64, the rotating register file can be used for shifting bytecodes and the prepare-to-branch instructions can be used to improve the predictability of the jump to the next block.

6 Conclusions

The paper has describes how the execution speed of a JVM interpreter on a VLIW processor can be improved substantially by pipelining. While one bytecode is in its execute stage, the next bytecode is in its decode stage, and the next bytecode of that is in its fetch stage. Pipelining is implemented by both source code rewriting and compiler modifications. Rewriting of the interpreter source code requires compiler expertise. This is not a severe problem because compiler and interpreter technology are very related to each other.

On the TriMedia VLIW processor, with load and jump latencies of three and four cycles, respectively, pipelining makes it possible to execute many of the simple bytecodes, such as additions and multiplications, in four clock cycles.

References

1. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996. 35
2. Harlan McGhan and Mike O'Conner. PicoJava: A Direct Execution Engine for Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998. 35
3. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Canada, 1998. 35
4. Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, France, October 1998. 35
5. Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM Implementation: Compilers versus Hardware. In John Morris, editor, *Computer Architecture 98 (ACAC '98)*, Australian Computer Science Communications, pages 101–110, Perth, 1998. Springer. 36
6. Gert A. Slavenburg. *TM1000 Databook*. TriMedia Division, Philips Semiconductors, TriMedia Product Group, 811 E. Arques Avenue, Sunnyvale, CA 94088, `www.trimedia.philips.com`, 1997. 36, 37
7. Gerrit A. Slavenburg, Selliah Rathnam, and Henk Dijkstra. The TriMedia TM-1 PCI VLIW Mediaprocessor. In *Hot Chips 8*, Stanford, California, August 1996. 36, 37
8. Vicky H. Allan, Reese B. Jones, Randall M. Lee, and Stephan J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3), September 1995. 36
9. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985. 36, 38

10. The Kaffe Homepage: `www.kaffe.org`. 37
11. The Personal Java Homepage: `java.sun.com/products/personaljava`. 37, 43
12. James R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, 1973. 38
13. Paul Klint. Interpretation Techniques. *Software — Practice & Experience*, 11(9):963–973, September 1981. 38
14. M. Anton Ertl. Stack Caching for Interpreters. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, La Jolla, California, June 1995. 43
15. Jan Hoogerbrugge and Lex Augusteijn. Instruction Scheduling for TriMedia. *Journal of Instruction-Level Parallelism*, 1(1), February 1999. 43
16. M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996. 46
17. Todd A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 322–332, 1995. 46
18. Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998. 46
19. Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A Code Compression System Based on Pipelined Interpreters. *Software — Practice & Experience*, 29(11):1005–1023, September 1999. 47
20. Intel, Santa Clara, CA. *IA-64 Application Developer's Architecture Guide*, 1999. 48