Allowing for ILP in an Embedded Java Processor

Ramesh Radhakrishnan, Deependra Talla and Lizy Kurian John
Laboratory for Computer Architecture
Electrical and Computer Engineering Department
The University of Texas at Austin, Austin, Texas 78712
{radhakri,deepu,ljohn}@ece.utexas.edu

Abstract

Java processors are ideal for embedded and network computing applications such as Internet TV's, set-top boxes, smart phones, and other consumer electronics applications. In this paper, we investigate cost-effective microarchitectural techniques to exploit parallelism in Java bytecode streams. Firstly, we propose the use of a fill unit that stores decoded bytecodes into a decoded bytecode cache. This mechanism improves the fetch and decode bandwidth of Java processors by 2 to 3 times. These additional hardware units can also be used to perform optimizations such as instruction folding. This is particularly significant because experiments with the Verilog model of Sun Microsystems picoJava-II core demonstrates that instruction folding lies in the critical path. Moving folding logic from the critical path of the processor to the fill unit allows to improve the clock frequency by 25%. Out-of-order ILP exploitation is not investigated due to the prohibitive cost, but in-order dual-issue with a 64-entry decoded bytecode cache is seen to result in 10% to 14% improvement in execution cycles. Another contribution of the paper is a stack disambiguation technique that allows elimination of false dependencies between different types of stack accesses. Stack disambiguation further exposes parallelism and a dual in-order issue microengine with a 64-entry bytecode cache yields an additional 10% reduction in cycles, leading to an aggregate reduction of 17% to 24% in execution cycles.

1 Introduction

Java is steadily increasing in popularity in the embedded and network chips arena, fueled primarily by the convenience and elegance of its "write-once run-anywhere" motto. The Java technology, which consists of the Java language, Java Virtual Machine (JVM) and the Application Programming Interface (API), achieves its platform independence by compiling Java source code into machine independent 'bytecodes' which are executed on the JVM. These bytecodes are typically executed by an interpreter [1], or a Just-In-Time (JIT) compiler [2], or executed directly by specialized Java processors [3, 4, 5].

Java is used in a broad range of applications from high end servers to low-end hand-held gadgets and house-hold appliances. The server class Java applications are typically executed using JIT compilers to achieve high performance. When interpreted, bytecodes have been seen to be 30x slower than optimized C code, whereas JIT compilers could provide up to 20x speedup (or more) with respect to interpreted code. However, the memory requirement of JIT compilers is prohibitively expensive for embedded systems and pervasive computing applications. Chips based on dedicated Java processors are favored for embedded applications due to their small memory requirements, low power consumption and low cost. Java processors are ideal for Internet information appliances such as digital set-top boxes, Internet TV's, smart phones, personal digital assistants (PDAs), and other consumer electronics applications.

Java processors such as Sun Microsystems picoJava cores [4, 5] and JEM [3] are low-cost hardware engines optimized to directly execute Java bytecodes. A glance through the SPEC JVM resources [6] shows that the picoJava-II offers performance in between that of general purpose embedded processors and general purpose processors, such as the ARM and the UltraSPARC respectively. A 120 MHz ARM processor with 32MB of memory provides a SpecJVM98 metric of 1.65 and the 300 MHz Sun Ultra AXi with 48MB yields a performance of 9.8. The performance of the picoJava-II core at 120 Mhz is estimated to be 4.2 [7]. For comparison, on the server side a Sun Ultra 60 with 512MB yields a performance metric of 31.9 and Compaq's Alpha server workstation with 2048MB yields a SpecJVM98 metric of approximately 75 [6]. Actual performance, clock frequency, area and power achieved by picoJava-II will depend on the cell li-

ISCA 2000 Vancouver BC Canada

braries used for synthesis. The performance numbers quoted here have been cited to merely indicate the approximate complexity of Java bytecode engines.

The Java Virtual Machine ISA is stack based and provides the advantage of dense code, which is important for the embedded environment. However, direct execution of bytecodes on stack based embedded processors is invariably constrained by the limitations of the stack architecture for accessing operands. Folding [8] is an optimization implemented in such architectures, (for example, picoJava-I and picoJava-II) to coalesce multiple stack based instructions to a single RISC-style instruction with optimized data accessing. High end Java platforms such as servers can implement equivalent optimizations in the JIT compiler. Microarchitectural enhancements in modern ILP processors can also contribute to exploiting the parallelism in Java. Space-time computing and multithreaded execution on clustered hardware as in Sun's MAJC [9] architecture will exploit both ILP and thread level parallelism in Java. However, at the low end of the Java horizon, Java chips for embedded and networked arena need to be specially tailored to exploit fine-grain parallelism. Many ILP techniques used at the server end will not be feasible to implement at the embedded end.

The basic goals of this study are to investigate cost-effective microarchitectural techniques to exploit the available parallelism in the bytecode stream and enhance the performance of Java processors for the embedded environment. Inexpensive hardware techniques to enable efficient fetching, decoding and execution of bytecodes are explored for applications which require high performance, but are constrained in terms of area, memory and power requirements (and therefore cannot use dynamic compilation or aggressive ILP techniques). We explore these issues adopting the picoJava-II processor core from Sun Microsystems as the reference processor. This choice is well justified as the picoJava-II core implements several optimizations including folding and is the highest performing specialized Java processor for the embedded environment [6]. This allows us to gauge our enhancements in comparison to a competitive baseline model.

Synthesis of the Verilog source code model of picoJava-II provided by Sun Microsystems in their Community Source Licensing effort [1], revealed that instruction folding performed by picoJava-II (in the decode stage) falls in the critical path. We propose to remove instruction folding from the decode stage, using a fill unit [11, 12] and a decoded bytecode cache (DB-Cache). By moving pattern checking and folding logic from the decoder to the fill unit, we effectively remove it from the critical path. The fill unit dynamically groups decoded bytecodes into a DB-Cache line, which can be fetched and executed in parallel when the same bytecodes are executed next time. The fill unit and the decoded bytecode cache also exploit method reuse in Java applications. In the picoJava-II, patterns have to be detected and folding performed for a sequence of bytecodes irrespective of prior use of the same sequence. However, using the fill unit we store the folded instructions in a DB-Cache line. The folded instruction is directly executed from the DB-Cache, when the same bytecode sequence is encountered again. The JVM uses a variable length instruction set and the DB-cache eliminates the need to dynamically identify instruction boundaries.

Another contribution of the paper is the *stack disambiguation* technique to exploit parallelism in the bytecode sequence. Use of the Java stack as an operand scratch pad for computations and as a local variable storage area results in enforcement of false dependencies while exploiting parallelism. We propose the use of special logic to distinguish between the operand stack (at the top of the stack) and the local variable area (at the bottom of the stack). Such a logical distinction between references to these distinct areas in the stack exposes more parallelism, which can be exploited by simple in-order multiple issue engines.

Our studies demonstrate that use of a fill unit and decoded bytecode cache can improve the decode bandwidth by 2x to 3x and the clock speed by approximately 25%. In-order paired (dual) execution in a simple RISC-style pipeline can utilize the increased decode bandwidth and reduce execution time by 10% to 15%. The proposed *stack disambiguation* technique can result in an additional 10% improvement, yielding an overall performance improvement of 17% to 24%.

1.1 Related Work

The fill unit and decoded instruction cache were hardware assists proposed by Patt et. al. as part of the HPS design philosophy [13, 14]. They describe the fill unit, which compacts instructions generated from a serial instruction stream into a decoded instruction cache. The use of these hardware assists on an HPS version of the DEC VAX processor was seen to result in significant performance improvements [14, 15]. The idea of a fill unit was revisited by Franklin and Smotherman in 1994, to dynamically group RISC instructions into a VLIW like instructions, and showed up to two times improvements over a single issue processor [12]. In a later paper, Smotherman and Franklin showed that collecting decoded X86 instructions us-

[1] In late 1998, Sun Microsystems instituted community source licensing for picoJava-II enabling tool vendors, chip developers, universities and research organizations to experiment with the Java processor RTL source code in verilog. The source code distributed is the same source RTL code Sun Microsystems used for their picoJava-II chip [10].

ing a fill unit and storing them in a decoded instruction cache can improve the decode rate as compared to a P6-like decoding structure [16]. They also presented a register allocation and renaming scheme using the fill unit approach to alleviate the excessive bandwidth placed on the P6 register renaming hardware. The picoJava-II does not require additional renaming stage or excessive ports to the Stack Cache register file, as it already has multiple ports to facilitate instruction folding optimizations.

Fill unit based optimizations for trace caches [17] and trace cache based preprocessing [18] improves performance by transforming instructions within a trace line. However, trace caches have been proposed for machines which can execute up to 8 or 16 instructions in parallel and hence store multiple basic blocks in a trace cache line. In marked contrast, we store only one basic block of bytecodes in the DB-Cache, as we are limited by the hardware resources which we can add in an embedded environment. We perform *instruction folding* within a DB-Cache line, instead of across basic blocks to reduce complexity. As an example, the designers of picoJava-II decided against adding a sophisticated dynamic branch predictor to the picoJava-II and instead opted for a static branch predictor. Investing the hardware resources for dynamic predictor was considered an overkill in terms of area and power. For high end server applications, dynamic branch predictors and aggressive ILP techniques are relevant. Object oriented architectural support in hardware is another approach taken to improve performance in Java processors, which tries to provide support for manipulating objects, specialized caches and other such optimizations [19, 20]. Several tradeoffs concerning the design of caches, branch predictors, and exploitation of parallelism using ILP techniques are explored in [21]. The dynamic execution time behavior of Java investigated in [22, 23] are also useful in understanding issues related to improving Java performance.

1.2 Outline of the paper

The rest of the paper is organized as follows: Section 2 describes the decoding and folding stages in the picoJava-II processor. Experiments using the Verilog model of the picoJava-II are also presented, showing that the folding logic falls in the critical path of the decode stage. Section 3 describes the addition of the fill unit and DB-Cache to the picoJava-II core to improve embedded Java performance. We also explain how *instruction folding* can be performed in the fill unit. Section 4 quantifies the performance improvement that can be attained using the fill unit and the DB-Cache. In section 5, cost-effective and restricted multiple instruction issue is investigated. We also propose an optimization, namely *stack disambiguation*, that exposes more parallelism resulting in additional performance improvements. Finally, we summarize and conclude in Section 6.

2 Background and Motivation

To motivate the research effort in the rest of the paper, we present an analysis of the decode and folding stages of the picoJava-II processor pipeline in detail. The picoJava-II is a stack based processor, since it is a silicon version of the stack-based Java Virtual Machine (JVM). We show that the decode stage is the longest stage in the pipeline (using critical path analysis), and the delay can be reduced by removing the folding logic from the decode stage.

2.1 Decoding instructions in picoJava-II

The picoJava-II core illustrated in Figure 1 consists of an integer execution unit, a stack cache, instruction folding hardware and optional instruction and data caches [24]. The instruction and data caches can be configured to be between 0 to 16 KBytes. The stack cache is implemented as a 64-entry register file with random access. The picoJava-II core implements the commonly executed bytecodes in hardware and handles the remaining ones using microcode and software traps. The pipeline in the picoJava-II is implemented in six stages as shown in Figure 2, similar to a simple RISC pipelines (with modifications to reflect the stack based nature of the bytecode ISA). The core fetches the instructions from the instruction cache into an instruction buffer in the fetch stage. The next stage decodes the instruction and performs folding optimizations. The stack cache is read for operands in the register stage, if necessary. The instruction executes in one or more cycles in the execute stage and goes through the memory access stage if it needs data from the data cache. Results are written back to the stack cache during the final stage in the pipeline.

A 16 byte instruction buffer is used to decouple the instruction cache from the rest of the pipeline. The processor can write a maximum of 8 bytes into the buffer at one time, where as it can read 7 bytes from the buffer at once. The bytecode ISA has variable length instructions, and most bytecodes consist of a 1 byte opcode, followed by 0, 1 or 2 operands. Since the average length of bytecode instructions is approximately 1.8 bytes [20], the processor can potentially read more than one instruction in a cycle. The processor can read up to four instructions, depending on the length of the instructions in the buffer.

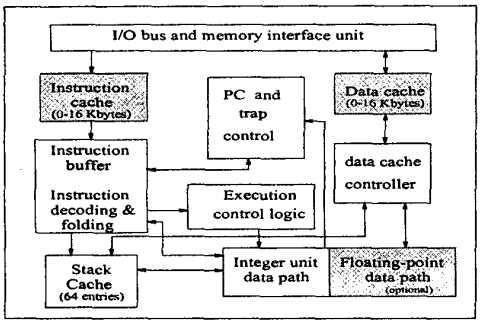


Figure 1: Block diagram of the picoJava-II microprocessor core. The caches and the floating point unit are optional.

FETCH	DECODE	REGISTER	EXECUTE	CACHE	WRITEBACK
Fetch fixed size cache lines from IC to the I-Buffer	Predecode & group instructions.	Access the register file (stack cache) for operands	Execute for one or more cycles	Access the data cache	Write back results into the operand stack. Forward results.

Figure 2: Basic Pipeline of the picoJava-II core

2.2 Folding optimizations performed in picoJava-II

The design choice of a stack based machine facilitates the advantage of small instruction (bytecode) size, but causes the generation of abundant load and store operations to the stack. For example, to compute an expression `a+b=c`, the sequence of bytecodes generated are:

- iload_a ; load local variable a into TOS (Top Of Stack)
- iload_b ; load local variable b into TOS
- iadd ; Pop top 2 elements and add them. Push result to TOS
- istore_c ; Store the element on TOS to local variable c

The picoJava-II takes 4 cycles to compute `a+b=c`, due to the requirement that bytecodes operate only on the elements at the top of the stack. The processor tries to eliminate the unnecessary loads and stores to the stack by utilizing an *instruction folding* technique. This is possible because the stack is implemented as a register file in the picoJava-II with 3 read ports, two write ports and a single cycle access to any entry of the stack. When a pattern like the one shown earlier is detected, the picoJava-II replaces it with a three-operand instruction `add c← a, b` which takes advantage of the single-cycle random access to the stack cache.

The folding optimization is implemented in the picoJava-II as follows:

1. Folding logic is added to the decoder design to detect patterns of instructions which can be folded together. The pipeline stages for a picoJava-II processor which implements the folding optimization is shown in Figure 3. Since there are innumerable folding patterns, only those patterns which occur with a high frequency are checked for using the decoder logic. The most common foldable patterns consist of instructions that are between 2 and 4 bytecode instructions.

2. Up to four instructions are decoded in the decoder and checked for foldable patterns consisting of 4 instructions. If no 4 instruction pattern is detected, then the instructions are checked for a three instruction pattern. In the event that no three instruction pattern is detected, potential for a two instruction pattern is explored, in the absence of which one instruction is sent to the next pipeline stage. If a pattern is detected, then the instructions are *folded* together and one (RISC style register-based) instruction is constructed for that pattern.

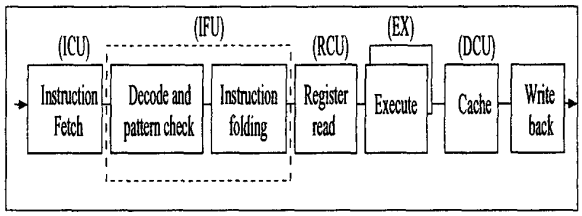


Figure 3: Pipeline stages in the picoJava-II processor to implement instruction folding. The stages within the dotted line are responsible for detecting patterns and folding.

2.3 Critical path analysis

Folding in the picoJava-II core is performed by the Instruction Folding Unit (IFU) where the instructions are grouped and pre-decoded. The picoJava-II documentation names the decode stage as the IFU, although both decoding and instruction folding is performed in this stage. The IFU classifies instructions into different types and groups them depending on their opcodes. Each of the bytes in the I-Buffer have other information associated with them, namely, a 4-bit length information corresponding to that opcode, a valid bit and a dirty bit. The folding logic examines the top 7 bytes of the instruction buffer and determines the number of instructions that can be folded depending on the bytecode, and length information

Stage/Unit	description of the pipeline stage or functional unit	delay (ns)
ICU	Instruction cache unit: fetches instructions and writes it to the I-Buffer	4.00
IFU (D)	Instruction Folding Unit: decodes and folds instructions	12.74
IFU (D')	Modified decode stage, after removing folding logic from the IFU	8.68
RCU	Register cache unit: access the register file (stack cache) for operands	9.33
EX	Execute stage, multicycle instructions are executed using microcode datapath	2.58
DCU	Data cache unit: handles sourcing and sinking of data for loads and stores	5.48

Table 1: Critical path delay for the picoJava-II datapath and control units

(associated with that bytecode) from the Instruction Cache Unit (ICU).

We analyzed the timing of all the single-cycle units in the picoJava-II core using the Verilog hardware description language model obtained from Sun Microsystems. Based on maximum timing optimizations in Synopsys targeting LSI Logic's 3.3V G10TM-p Cell-Based 0.29 micron ASIC library [25], we synthesized the Verilog code. Table 1 shows the results of the synthesis of various stages of the picoJava-II core. The pipeline stage that is the slowest is the IFU (decode+folding) stage.

Figure 4 shows the logic elements in the critical path in the Decode pipeline stage (IFU). The length decoder computes the lengths of prospective instructions to be folded using special adders called *index adders, Iadd.* These adders use length information of each byte (l_0 to l_6) and the position of the byte in the I-Buffer. Symbols (L_0 to L_3) represent the cumulative length of one to four instructions. For example, L_2 represents the sum of lengths of the first three instructions out of the four prospective foldable instructions. To compute L_3, L_2 needs to be computed which in turn depends on the earlier cumulative length, elongating the critical path.

The folding decoder consists of an array of folding decode (*FDEC*) blocks, each of which examines two consecutive bytes and generates information about prospective foldable instructions. Two bytes are used by each block due to the fact that the picoJava-II opcodes can either be one byte or two bytes long[2]. Using the accumulated lengths and the *FDEC* block outputs, folding patterns can be identified. Associated folding logic uses these patterns to output the number of foldable instructions and also a shift signal to the I-Buffer to move the contents of the buffer for decoding in the next cycle.

We isolated the folding logic from the decode stage and present the timing analysis of the modified decode stage (D') in Table 1. Isolating the folding logic from the critical path brings down the maximum delay from 12.74 ns to 8.68 ns, thus allowing to drive the clock at a faster rate than it would have been possible with folding logic in the critical path. We propose to move the instruction folding logic from the decode stage to the fill unit. The fill unit is outside the critical path of the pipeline, and does not affect the clock as it can take multiple cycles to operate without affecting the performance. After the folding logic is removed from the critical path of processor and added to the fill unit, the register cache unit (RCU) becomes the longest stage in the pipeline (having a delay of 9.33 ns). Therefore, by moving the folding logic to the fill unit we can increase the clock rate of picoJava-II by approximately 25%, which is a significant improvement. Increase in clock frequency will however result in increase in processor power. Embedded applications with power as the primary design constraint may want to make use of only the other proposed microarchitectural enhancements.

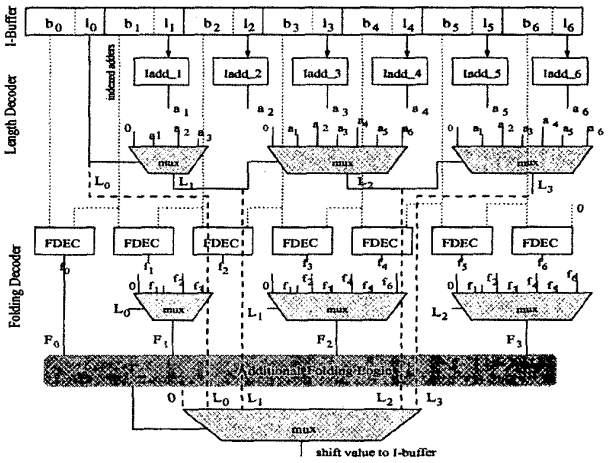


Figure 4: Logic along the critical path of the decode stage.This figure shows the logic that contributes to the critical path , in the decode stage (additional logic occurring in parallel is not shown).

[2]The picoJava-II uses special two byte opcodes for implementation specific extensions to the JVM instruction set [4]

3 Fill unit and DB-Cache operation: decoding and folding

3.1 Collecting decoded bytecode instructions

Figure 5 shows the fill unit approach that we propose in this paper. The fill unit collects decoded bytecodes and stores them into the lines of a decoded bytecode cache (DB-Cache). During instruction access, if the instruction fetch address matches an entry in the DB-Cache, the entry in the DB-Cache is given priority over the normal instruction cache and decoding path, and decoded bytecodes are supplied directly to the operand fetch stage. It should be noted that by storing the decoded bytecodes in the DB-Cache, there is no need to repeat fetching of instructions from the normal instruction cache nor finding the instruction boundaries[3]. However, operand fetch must still occur before the instructions can be fed to the processor core for execution. The advantages of the fill unit are that it can collect more than two decoded instructions into the same DB-Cache line and pass to the operand fetch stage in one cycle. Thereby, we increase the instruction decode bandwidth without increasing the decoder width, the fetch bandwidth and the instruction buffer size.

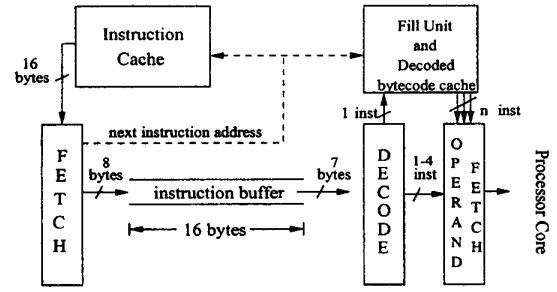


Figure 5: Increasing decode bandwidth using a fill unit and DB-Cache. The processor core is fed decoded instructions from the normal decode path, or from the DB-Cache (when there is hit in the DB-Cache).

Since a variable number of decoded bytecodes form a DB-Cache line, each line should also include a next-address field and supply it to the branch unit. Bytecodes are filled into the DB-Cache line only if they represent more than one instruction, as it is more efficient to fetch single instructions from the instruction cache. The DB-Cache line could potentially include as many bytecodes as possible, but we restrict it to 5 decoded bytecodes for this study. We place an additional restriction that only bytecodes of length 3 bytes and below are stored in the DB-Cache.

[3]The picoJava-II has variable length instructions, and instruction boundaries are calculated in the IFU stage while decoding the bytecodes.

We are able to capture more than 99% of the bytecodes executed (even with this restriction), as most of the longer length bytecodes are executed very infrequently. Based on these constraints we would require 15 bytes (5 times 3) in each DB-Cache line to store up to 5 decoded bytecodes.

Data dependencies will not prevent filling of the decoded bytecodes, but will be resolved by the processor core similar to instructions fetched along the normal path. Control dependencies, such as branches will stop the filling of a DB-Cache line in our design. We could use a more complex design where we fill past branches (to store multiple basic blocks), however, that would prove to be too expensive in terms of resources (additional memory and branch prediction logic) for the picoJava-II processor. The next-address field in each DB-Cache line is used to provide correct instruction sequencing. If there is no branch or control-changing instruction in a DB-Cache line, the next-address field is set to the first instruction that was not filled into the current line. A branch is represented in the DB-Cache line by two fields - a condition and a branch address. When the last instruction to be filled is a conditional branch, the next-address field serves as the branch-untaken address, and the branch address field serves as the branch taken address. This is to facilitate pre-fetching of lines from the DB-Cache by using the branch predictor to predict the outcome of the branch. The predictor output is used to generate the next instruction address and sent to the instruction cache and the DB-Cache. The total DB-Cache line size for this design will be 23 bytes (15 bytes for decoded bytecodes and 8 bytes to store next-address and the branch address).

3.2 Performing folding operations

In the previous section we saw how the DB-Cache can be used to increase the decode bandwidth for the picoJava. We now investigate some optimizations which can be performed by the fill unit while filling DB-Cache lines. One such optimization is *instruction folding*. As explained in the Section 2.3, implementing the folding optimizations in hardware requires adding additional logic to the decode stage. This could have a serious impact, especially if we want to detect many patterns, and patterns consist of large instructions. Since the folding logic falls in the critical path of the processor, it has to be highly optimized and kept to a bare minimum (in terms of functionality). Instead, we can use the fill unit to perform the folding operations while it is filling the decoded instructions into the DB-Cache, and this in turn has its own advantages. We explain how the fill unit performs the folding operations using Figure 6.

The fill unit collects instructions as described ear-

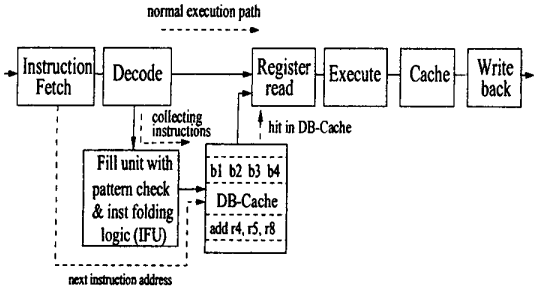


Figure 6: Storing basic blocks and *foldable patterns* of bytecodes in the DB-Cache. The DB-Cache will store basic blocks of bytecodes as well as RISC-style instructions (folded bytecodes).

lier and checks for instructions that can be stored in a DB-Cache line. We incorporate the pattern check and folding logic in the fill unit so that it checks for patterns in the bytecodes that it collects. If it detects a pattern, then it can store the synthesized (register-based) instruction and substitute it for the pattern of bytecodes. This is then stored in the DB-Cache instead of the bytecodes themselves. In Figure 6, [b1, b2, b3, b4] is a basic block of bytecodes which is collected and filled into the DB-Cache. The bytecodes are stored without modification (folding) if they do not form a foldable pattern. During the execution of the program if the fill unit came across a 4 bytecode sequence which matched a foldable pattern, the fill unit fills the DB-Cache line with the synthesized RISC-style instruction. A DB-Cache line could contain a mix of both a synthesized instruction and bytecodes as long as they do not exceed the DB-Cache line size (and are part of the same basic block). The advantages of incorporating the pattern checking and instruction folding in the fill unit are as follows:

1. By moving the pattern checking and folding logic from the decoder to the fill unit, we are effectively removing it from the critical path of the decode stage. Now that the folding logic is not in the critical path we can implement other patterns (less frequent) that were not included because of cost and timing considerations.

2. The decode width of the processor need not be increased to identify patterns which are wider than the decode width. Since the picoJava-II has a decode width of 7 bytes(1-4 bytecodes depending on their length), it can identify patterns of 4 bytecode instructions. If it has to identify instruction patterns of 5 instructions, the decode width will have to be increased from 4 to 5 (by increasing the I-Buffer and decode logic). If we move the pattern checking and folding logic to the fill unit, we still need to only have a decode bandwidth of 4, as the pattern check is done by the fill unit when collecting the decoded instructions (the DB-Cache line holds up to 5 instructions). Along the same lines, the picoJava-I core [4] has a decode width of two, and can fold only up to 2 bytecodes. If we add a fill unit and DB-Cache (similar to the one proposed in this paper), and move the folding logic to the fill unit we will be able to detect patterns of 3 and 4 in picoJava-I without increasing the decode width.

3. In the picoJava-II we have to check for patterns and perform folding for a sequence of bytecodes, no matter how many times it is executed. However, using the fill unit we replace the pattern with the synthesized RISC-style instruction and store it in the DB-Cache line, and then we can issue the synthesized instruction when the pattern is repeated (assuming we have a DB-Cache hit). In the picoJava-II (without the DB-Cache) the bytecodes would have to go through the decode stage and perform the folding operation again, before it can be issued thereby taking more cycles.

4 Effectiveness of the fill unit and DB-Cache

4.1 Experimental Methodology

The effectiveness of the proposed techniques are evaluated using a picoJava-II simulator that accepts bytecode traces extracted using a tracing JVM [26]. The simulator models the picoJava-II pipeline on a cycle by cycle basis using instruction latencies[4] of each bytecode. Single cycle instruction and data caches with 100% hit ratio are assumed. The branch predictor used in the picoJava-II is a static predictor, and has a penalty of 3 cycles for mis-predicted branches (not taken branches).

We use 6 benchmarks from the SpecJVM98 suite [28] to evaluate the proposed architectural additions to the picoJava-II core design. A description of the benchmarks is given in Table 2. The benchmarks were run with with the *s1* data set. The number of bytecodes simulated range from 2 million for *db* to 176 million for *mpegaudio*. These benchmarks do not include any graphics, networking or AWT, and therefore do not represent a whole spectrum of Java applications. They however, do provide us with a starting point to evaluate the effectiveness of microarchitectural enhancements for Java processors.

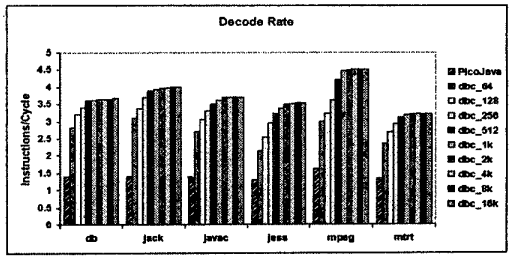
[4]The instruction latencies for the bytecodes are obtained from [27]. We assume a 30 cycle latency for software traps. These traps account for less than 0.1% of the total instructions.

benchmark	Description	count
jess	NASA's CLIPS rule-based expert system	8M
db	Data management software from IBM	2M
javac	Sun JDK Java compiler	5M
mpegaudio	software algorithm to decode an MPEG layer 3 audio stream	115M
mtrt	A program that ray traces an image	50M
jack	A parser-generator from Sun	176M

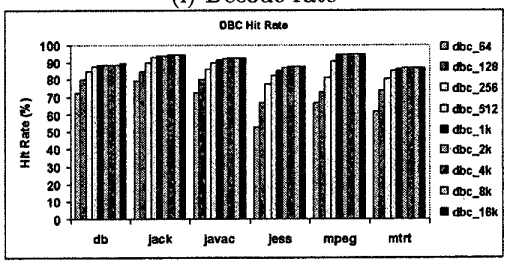
Table 2: Description of the SpecJVM98 Benchmarks used in this study

4.2 Instruction decode rate

Figure 7 shows the increase in decode rate which can be achieved using the fill unit and the DB-Cache. The decode rate increases as we increase the number of entries in the DB-Cache from 64 to 16K. DB-Cache with more than 1K entries are shown only for comparison purposes, since it would not be feasible to have such a big cache structure in the picoJava-II core (due to its power and area impact). The picoJava-II processor could potentially decode up to four instructions (due to folding), and we see a decode rate of 1.4 to 1.7 across the different benchmarks (first bar in Figure 7 (i)). We can achieve a two-fold increase in decode rate even with a DB-Cache of only 64 entries. The decode rate increases to up to three-fold for a 1K entry DB-Cache. The increase in decode rate corresponds to the hit rate that is seen for the different DB-Cache configurations (Figure 7 (ii)).



(i) Decode rate



(ii) Hit rate

Figure 7: Trends in decode rate and Hit rate for different DB-Cache sizes

4.3 Folding efficiency and performance

In this section, we look at the folding coverage and percentage of instructions that are eliminated when moving the folding logic to the fill unit. As described in the previous section, the number of bytecode patterns detected would depend on the hit rate of the DB-Cache, and certain other factors which we describe in this section.

Figure 8 shows the absolute number of patterns detected for each benchmark in the picoJava-II processor with and without the fill unit and DB-Cache. The total number of folded two instruction patterns (2-fold), three instruction patterns (3-fold) and four instruction patterns (4-fold) are shown for each benchmark. The last bar in the graph (named as picoJava-II) shows the number of patterns detected when the pattern check is done at the decode stage of the picoJava-II processor. The number of three instruction and four instruction patterns detected in the fill unit is seen to be lower compared to the normal picoJava-II processor. This is mainly due to two reasons; (i) We fail to detect a pattern the first time the fill unit encounters it, since it follows the normal decode and execute path, which does not have pattern checking and folding logic, and (ii) some of the larger patterns, especially four instruction patterns get stored in two lines due to overflow. For example, if a 4 instruction pattern starts at the middle of a DB-Cache line, and continues into the next cache line (because the pattern starts at the middle of a basic block stored in that line) we would fail to detect it. We refer to this as *pattern fragmentation.* Because of pattern fragmentation, a four instruction pattern might avoid being detected when using the fill unit. However, if the split pattern forms a smaller instruction pattern (of two or three instructions) which is stored in one DB-Cache line, the pattern checker in the fill unit will detect it. Thus, there is a potential that the total number of two instruction patterns that are detected will be more when we do the pattern checking in the fill unit. It can be seen in Figure 8 that this is the case, and the two instruction patterns (2-fold) that are detected using the fill unit exceed that detected by the normal picoJava-II processor (as these 2 instruction patterns were part of a longer 3 or 4 instruction pattern, which was detected in the picoJava, but went undetected in the fill unit).

Figure 9 shows the percentage of total dynamic instructions which were folded for four of the SpecJVM98 benchmarks, based on the instruction patterns (2-fold, 3-Fold and 4-Fold) that were detected in the bytecode execution stream. The percentage of instructions eliminated when folding is

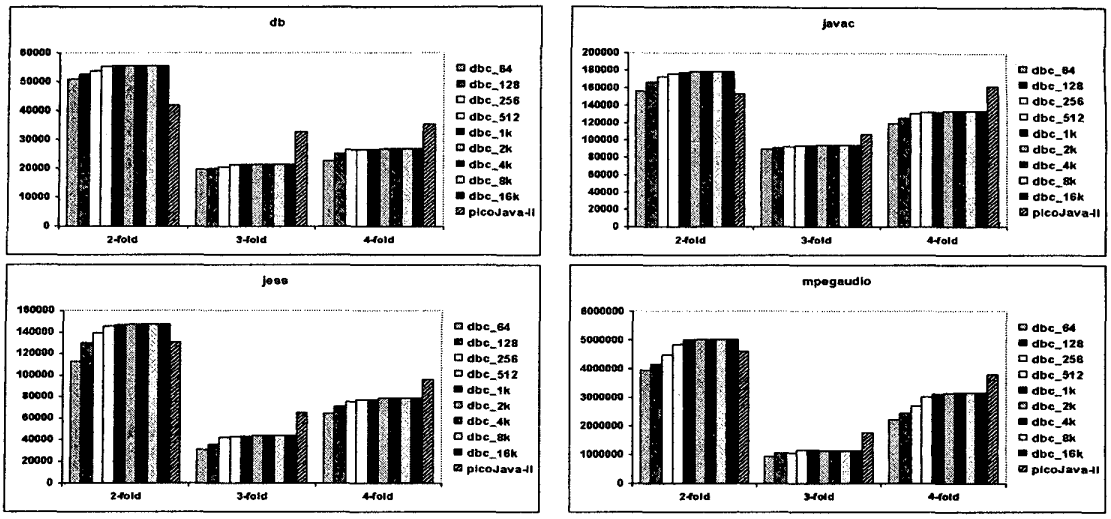


Figure 8: Folding patterns detected in picoJava-II (at decode stage) and when using the fill unit and DB-Cache.

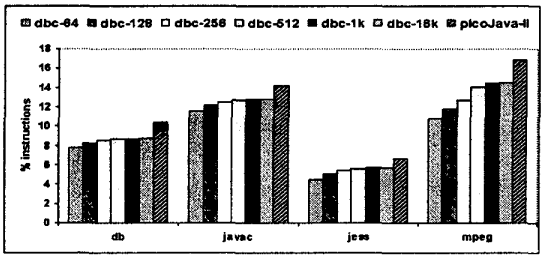


Figure 9: Percentage of total dynamic instructions that are eliminated (folded) using instruction folding

done at the decode stage is 2% higher compared to using a DB-Cache (to hold the folded patterns) of 64 entries for *db*, *javac* and *jess*. This percentage difference increases to 6% for *mpeg*, but goes down to almost 2% when using a DB-Cache entry of 1k entries. It can be inferred from the graph, that using a fill unit to capture and store foldable instruction patterns in the DB-Cache allows to eliminate almost the same percentage of instructions as when doing the folding at the decode stage. Failing to detecting a pattern the first time it is encountered, *pattern fragmentation* and misses in the DB-Cache cause the percentage of instructions to be slightly lower than what one would observe when folding is done at the decode stage.

5 Improving performance using multiple instruction issue

One straightforward technique to improve performance is to exploit parallelism in the instruction stream using ILP techniques. However, obtaining increased performance in a resource-constrained embedded processor involves detecting and executing multiple instructions in a cost-effective manner. The previous sections illustrated how a fill unit and DB-Cache can achieve up to a three-fold increase in the decode rate. In this section, we investigate the performance improvement achieved by executing multiple instructions in conjunction with the fill unit and DB-Cache which provide an abundant supply of instructions. While we want to exploit instruction level parallelism, we would like to keep the complexity manageable for embedded environments and hence restrict the exploration to in-order execution as opposed to using aggressive ILP techniques which require substantially higher amount of resources.

5.1 Effectiveness of in-order multiple instruction issue

To detect the level of exploitable concurrency in the bytecode stream, we relax the resource constraints on number of execution units and estimate the optimum number of execution units. When the execute stage is fed a basic block of instructions from the DB-Cache, it could potentially execute all of them in parallel assuming there are no dependencies and resource constraints. If any instruction in the basic block writes to the stack, then all following instructions which read from the stack will have a dependency on that instruction (and will be executed in the next cycle). We simulate the picoJava-II core with the above constraints for the different benchmarks to find the po-

tential execution rate assuming a DB-Cache of 128 entries. It should be noted that no exploration beyond basic blocks is done and what is revealed here is by no means the bound on available parallelism in the whole application.

benchmark	Instructions Executed in Parallel				
	1	2	3	4	5
db	90.12	9.88	0.00	0.00	0.00
javac	93.41	6.28	0.17	0.00	0.00
jess	92.05	7.95	0.00	0.00	0.00
mpeg	94.70	5.30	0.00	0.00	0.00
mtrt	91.22	8.78	0.00	0.00	0.00

Table 3: Percentage of instructions executed in parallel when using a DB-Cache of 128 entries

Table 3 shows the percentage of instructions that can be executed in parallel for the picoJava-II with a fill unit and DB-Cache (of 128 entries). It is seen that 90-94% of the instructions are executed without being paired with other instructions. *Javac* is one benchmark in which we see a negligible small percent of the instructions being executed at a rate of 3 instructions per cycle (0.17%). We never see 4 or 5 bytecodes being executed in one cycle, because of the dependencies to the stack. This implies that in the limited window we are investigating, most of the instructions have a dependency to the stack, and is limiting the execution width even if we are able to decode up to 5 instructions in one cycle. Since there is one unique stack, an operation which expects an operand from the stack must wait and depend on the previous load or written back data to be stored into the stack. This causes a severe limitation to the instruction level parallelism that can be exploited. Therefore, it we will be not be possible to gain much performance by adding resources to execute more than 2 instructions in one cycle, without alleviating the constraints imposed by the stack.

Figure 10 shows the relative performance of a dual issue architecture with the proposed fill unit and a few different DB-Cache configurations. This architecture can execute pairs of bytecodes if they have no dependencies. If one bytecode has a longer latency than than its pair, the pipeline stalls till the longer latency instruction completes. A DB-Cache of 64 entries yields a performance improvement of 10% to 13%, but performance does not improve beyond 15% for larger DB-Caches. Even though this is not a very significant improvement in performance, it should be kept in mind that this is in addition to the clock speed improvements made possible by the fill unit and the DB-Cache. Performance improvement is limited primarily by stack dependencies. In the next section, we show how we can alleviate some of this problem by using an optimization called the *stack disambiguation.*

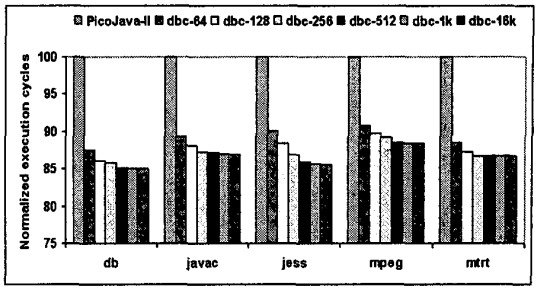


Figure 10: Performance improvement when adding a fill unit, DB-Cache (64-16K entries) and instruction execute width of two to a picoJava-II processor.

5.2 Using stack disambiguation to improve exploitable concurrency

The Java stack is used to hold the operand stack, local variable area and frame data for each method. The operand stack is used as a scratch pad to perform computations and store values. All operations on the operand stack take place at the top of the stack and the OPTOP (top of operand stack) register is used to manipulate the operand stack. The local variable area consists of values for commonly used local variables for a particular method, and occupies the bottom entries of the Java stack. A register VARS is used in the picoJava-II to mark the beginning of the local variable area. The sizes of the operand stack and local variable area varies depending on the requirements of each method. Any data access generated as an offset from the OPTOP or VARS register in the picoJava, is recognized as a stack access and all other data access go the data cache. If we use additional logic to distinguish between stack access generated as an offset to the OPTOP and VARS register, we can schedule more instructions in parallel since the operand stack is the true stack, and the local variable area is a storage area. We refer to this optimization to distinguish between operand stack and local variable access as *stack disambiguation*, since it logically separates the stack into the operand stack and local variable area.

Using additional logic to check the type of access, could affect the cycle time if it falls in the critical path of the data cache access. Instead we could use an additional bit along with each instruction stored in the DB-Cache, to mark if it is an access to the operand stack or the local variable area[5]. The percentage of instructions executed in parallel, when using *stack dis-*

[5]This would add 5 bits to each DB-Cache line for the configuration used in this study

benchmark	Instructions executed in Parallel				
	1	2	3	4	5
db	78.76	21.06	0.17	0.00	0.00
javac	81.25	18.17	0.57	0.00	0.00
jess	81.06	18.20	0.74	0.00	0.00
mpeg	87.32	12.47	0.21	0.00	0.00
mtrt	86.44	11.56	2.00	0.00	0.00

Table 4: Percentage of instructions executed in parallel when using *stack disambiguation* with a 128 entry DB-Cache)

ambiguation is given in Table 4. As expected, we see an improvement in the execution rate, and more instructions being executed in parallel, when compared to the data in Table 3 where we do not distinguish between stack and local variable access. The percentage of instructions that are executed with a width of three increases, and we see 0.17% to 2% of the instructions in this category. In Table 3 only *javac* was seen to have a small percentage of instructions which were executed with a rate of 3 instructions per cycle. Similarly, we see 2 to 3 times improvement in the percent of instructions that are executed in pairs.

The performance of the picoJava-II processor with the fill unit configuration and the *stack disambiguation* is plotted relative to the performance seen for the original picoJava-II in Figure 11. Performance improvements range from 17% to 24% for the different DB-Cache configurations. *Stack disambiguation* alone is seen to contribute approximately 10% over a naive stack access implementation.

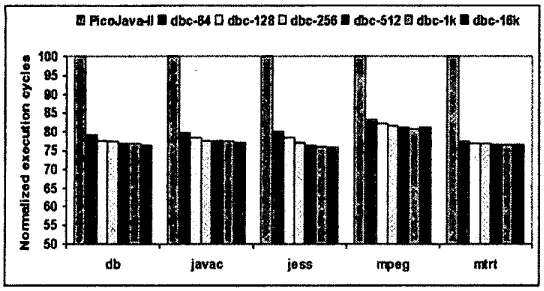


Figure 11: Relative performance of picoJava-II using the fill unit, DB-Cache (64-16K entries), execution width of two and *stack disambiguation*

6 Summary and Conclusions

Due to their low cost and low power requirements, achieving high performance in Java processors necessitates a balance of complexity and performance. In this study we extended the concept of fill unit for Java bytecode engines, and explored the effectiveness of fill unit based optimizations. We show that a fill unit coupled with a relatively small decoded bytecode cache (64 entries) improves performance, showing that cost-effective techniques exist to exploit limited ILP for the low-end embedded Java processors. The major contributions of this paper are summarized below:

- *Performing instruction folding using the fill unit and DB-Cache allows to increase clock frequency by approximately 25%.*

  The picoJava-II core performs instruction folding in the decode stage. Based on maximum timing optimizations in Synopsys targeting LSI Logic's 3.3V G10TM-p Cell-Based 0.29 micron ASIC library [25], synthesizing the Verilog code for the picoJava-II core showed that instruction folding *does* elongate the critical path.

- *Adding a fill unit and DB-Cache improves decode bandwidth by 2x to 3x.*

  This result is obtained from simulation studies using the SpecJVM98 benchmarks on a picoJava-II simulator with fill unit and DB-Cache. This improvement in decode width is obtained without increasing the baseline decoder width, fetch width and the instruction buffer size.

- *Instruction folding in the fill unit may not detect all foldable patterns, however the loss in performance is negligible.*

  Failing to detect a pattern during its first occurrence, detecting only a part of the pattern (*pattern fragmentation*) and misses in the DB-Cache reduce the overall number of dynamic instructions folded by approximately 2%.

- *An in-order dual issue execution core with a 64-entry DB-Cache improves performance by 10% to 14%.*

  Increasing the number of functional units to more than two is not beneficial due to stack dependencies. It is seen that 90% to 95% of the instructions cannot be paired and issued in parallel due to dependencies on the stack.

- *Stack disambiguation increases pairing of instructions by eliminating false dependencies.*

  Logically separating the stack accesses to the operand stack and the local variable area allows more instructions to be executed in parallel. Dependencies to the stack is *the* limiting factor for exploiting ILP in stack-based architectures. Using our technique of *stack disambiguation* allows

for an additional improvement of 10% over the fill unit and DB-Cache configuration, leading to an overall improvement of 17% to 24% in execution cycles over the picoJava-II processor.

It may be noted that we have not investigated any out-of-order ILP techniques. Incorporating fill unit and the DB-Cache into the picoJava-II core was seen to improve performance in many different aspects. Implementing aggressive ILP techniques to improve performance in embedded processors has to be done judicially, keeping in mind the power and area impact of the optimizations.

Acknowledgments

This research is supported in part by NSF under Grants EIA-9807112 and CCR-9796098. Laboratory for Computer Architecture is also supported by Sun Microsystems, Dell, Intel, Microsoft and IBM. We thank Manoj Franklin for providing part of the simulator code and Juan Rubio for developing the bytecode tracer. We also thank Srivatsan Srinivasan and Vivekanand Vedula for their help with the Synopsys tools. We acknowledge the comments and suggestions by Ravi Bhargava, Madhavi Valluri and the anonymous referees which helped to improve the paper.

References

[1] T.H Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J-L. Baer, B. N. Bershad and H. M. Levy, "The Structure and Performance of Interpreters," in *Proceedings of ASPLOS VII*, pp. 150–159, 1996.

[2] 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.

[3] A. Wolfe, "First Java-specific chip takes wing," *Electronic Engineering Times*, April 1997. http://www.techweb.com/wire/news/1997/09/0922java-.html.

[4] H. McGhan and M. O'Connor, " PicoJava: A direct execution engine for Java bytecode ," *IEEE Computer*, pp. 22–30, October 1998.

[5] M. O'Connor and M. Tremblay, " picoJava-I: The Java virtual machine in hardware ," *IEEE Micro*, pp. 45–53, March-April 1997.

[6] "SPEC JVM 98 Results." http://www.spec.org/osg/jvm98/results/jvm98.html.

[7] "picoJava Technology FAQ." http://www.sun.com/microelectronics/communitysource/picojava/techfaq.html.

[8] L.-C. Chang, L.-R. Ton, M.-F. Kao and C.-P. Chung, "Stack operations folding in Java processors," *IEE proceedings on Computers and Digital Techniques*, vol. 145, pp. 333–340, Sept 1998.

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

[10] "Community Source Licensing for picoJava Technology." http://www.sun.com/microelectronics/communitysource-/picojava/.

[11] W.-M. Hwu and Y.N. Patt , "HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality," in *Proc. of 13th Annual International Symposium on Computer Architecture*, pp. 297–306, 1986.

[12] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," in *Proceedings of Micro-27*, pp. 162–171, 1994.

[13] Y.N. Patt, W.-M. Hwu and M.C. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," in *Proceedings of Micro-18*, pp. 103–108, 1985.

[14] Y.N. Patt, S.W. Melvin, W.-M, Hwu and M.C. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture," in *Proceedings of Micro-18*, pp. 109–116, 1985.

[15] Y.N. Patt, S.W. Melvin, W.-M. Hwu, M.C. Shebanow, C. Chen and J. We, "Run-Time Generation of HPS Microinstructions from a VAX Instruction Stream," in *Proceedings of Micro-19*, pp. 109–116, 1986.

[16] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit ," in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO-28)*, 1995.

[17] D.H. Friendly, S.J. Patel and Y.N. Patt, "Putting the fill unit to work: dynamic optimizations for trace cache microprocessors," in *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, pp. 173–181, 1998.

[18] Q. Jacobson and J.E. Smith, "Instruction pre-processing in trace processors," in *Proceedings of 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*, pp. 125–129, 1999.

[19] N. Vijaykrishnan, *Issues in the Design of a Java Processor Architecture.* PhD thesis, College of Engineering, University of South Florida, Tampa, FL 33620, July 1998.

[20] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, "Object-oriented architectural support for a Java processor," in *Proceedings of the 12th European Conference on Object-Oriented Programming*, pp. 430–455, July 1998.

[21] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural issues in java runtime systems," in *Proceedings of 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*, pp. 387–398, January 2000.

[22] A. Barisone, F. Bellotti, R. Berta, and A. De Gloria , "Instruction Level Characterization of Java Virtual Machine Workload," in *Digest of Workshop on Workload Characterization (WWC-99)*, 1999.

[23] 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*, pp. 281–284, October 1999.

[24] Sun Microsystems, *picoJava-II Microarchitecture Guide*, March 1999.

[25] "Synopsys Online Documentation," *Guidelines and Practices for Synthesis* v.1997-08.

[26] J. Rubio, "Characterization of java application at the bytecode level," Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, May 1999.

[27] Sun Microsystems, *picoJava-II Programmer's Reference Manual*, March 1999.

[28] "SPEC JVM98 Benchmarks." http://www.spec.org/osg/jvm98/.