ICS

TECHNICAL REPORT

Improving Interpreted Execution Performance with Java Bytecode SuperOperators

Ana Azevedo, Alex Veidenbaum and Alex Nicolau
{aazevedo, alexv, nicolau}@ics.uci.edu

UCI-ICS Technical Report #02-22
Department of Information and Computer Science
University of California, Irvine, CA 92697

November, 2002

Information and Computer Science
University of California, Irvine

Improving Interpreted Execution Performance with Java Bytecode SuperOperators

Ana Azevedo, Alex Veidenbaum and Alex Nicolau
{aazevedo, alexv, nicolau}@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine, CA 92697, USA

Technical Report #02-22
November, 2001

Abstract

This paper exploits the concept of optimizing the interpreted execution of Java programs with *SuperOperators* (SOs). SOs are groups of bytecode operations used to produce interpreter engines with specialized instructions. The present work makes 3 distinguished contributions to this topic.

Firstly, we show that less than 20 SOs formed by basic blocks cover more than 50% of all bytecodes executed by an application and are enough to yield the bulk of performance improvement when optimizing interpreters with SOs. We analyze SOs formed by the most frequently executed program basic blocks and SOs formed by special sub-patterns of Java bytecode operations that compose the basic blocks. Such sub-patterns are extensions of PicoJava's stack operation folding (OF) patterns. Unlike SOs formed by basic blocks, we find OF patterns repeat across a wide range of applications.

Secondly, we compare techniques for optimizing interpreters with SOs. We show that the number of stack accesses and stack pointer updates, implicit in the bytecode semantics, is more limiting to the interpreter performance than the bytecode dispatch overhead. Our findings suggest that an interpreter that fully optimizes the top SOs formed by basic blocks, reducing both sources of overhead, yields up to fourfold performance improvement compared to previous techniques.

Finally we assess the efficiency of a software implementation of the stack operation folding mechanism. We design statically customized interpreter versions that use a limited number of non-patented Java bytecode opcodes to represent SOs formed by OF patterns valuable across applications. We also propose a dynamic scheme that is more flexible in customizing the interpreter for a particular application. Both approaches use annotation attributes in the class files marking occurrences of the most valuable SOs, dispensing with the expensive pattern search and classification at runtime. Our statically customized interpreter versions, deploying a limited subset of SOs, and our dynamically customized version improve the performance of SPEC JVM98 and Java Grande Forum benchmarks by 7% to 39%.

Contents

List of Figures

List of Tables

1 Introduction

Program interpretation is the process of emulating in software the basic tasks of fetching and decoding instructions of a normal program execution, which are usually done by a microprocessor hardware. Therefore interpretation bears inferior performance compared to direct program execution.

An interpreter is essentially structured as an infinite loop that reads in a new instruction from an array of instructions pointed by a software program counter, decodes the instruction, transfers control to code parts that handle the instruction just decoded, updates the program counter to point to the next instruction in the stream and eventually returns to the same fetch-decode-execute cycle to translate the next instruction. The implementation of an interpreter loop in a high level language like C is shown in the upper side of Figure 1, and is referred to as a *switch-based* or *bytecoded* interpreter.

There are two main sources of overhead in the interpreted execution of Java bytecode programs. We discuss the overheads by analyzing Figure 2, which shows part of LaTTe's interpreter engine [14] directly written in SPARC assembly code. LaTTe is the open-source, high performance JVM used as the underlying framework in all the experiments reported in this paper.

Each Java bytecode implementation is declared as a section of assembly code at position `_interpret_start + opcode * DISP`, where `_interpret_start` label marks the base address of the loop; `opcode` is the byte representing the bytecode opcode; and `DISP` is the maximum number of bytes (256) reserved for the native code that implements the bytecode semantics. Notice that in LaTTe's interpreter engine, the most important loop variables, as the top of the stack (`TOP`) and the logical program counter (`PC`), are kept in SPARC machine registers for improved performance.

Figure 2 also details the execution of an `ILOAD` operation, which requires 6 machine instructions. The tasks of fetching, decoding and jumping to the next bytecode to be executed define the bytecode dispatch cost. The dispatch cost requires 4 more instructions of expensive type (load and branch instructions) to be executed: a `ldub` to load the next bytecode; a `sll` to calculate the next bytecode address; a `jmp` to transfer control to that new address; and an `add` to update the program counter. In this example we can also notice that the dispatch cost is more than half of the typical size of the native code implementation of the most commonly executed Java bytecode (load from local variables on the average account for 35.5% of SPEC JVM98 total executed bytecodes [18]).

Another source of overhead exists in the execution of the bytecode semantics in which a stack machine is being emulated in software. This forces the copy of operands and results to and from the other Java memory areas (e.g., the heap and the local variables array) to the Java stack.

Any technique that reduces the cost of dispatching a new bytecode, reduces the data transfer to/from the Java stack or reutilizes the translation work of previously executed bytecodes can improve the overall performance of Java programs.

JIT compilers [3, 12, 14] eliminate the above issues altogether at the cost of more space to store the compiled methods and the compilation framework itself. A JIT compiler is not a viable solution in domains where space constraints limit the available memory.

```
void bytecoded_Interpreter_Engine{

char program[] = {ICONST_2, ICONST_2, ICONST_1, IADD,...}
char *pc = program;                          /* bytecode pointer */

/* dispatch loop implementation */
while (true){
  switch(*pc++){     /* Fetch, Decode, Update pointer */
    case ICONST_1: *++sp = 1; break;  /* Execute bytecode */
    case ICONST_2: *++sp = 2; break;
    case IADD: sp[-1] += *sp; --sp; break;
    ...  /* Other cases */
  }
}
}
```

```
void threaded_Interpreter_Engine{

void * program[] = {&&ICONST_2, &&ICONST_2, &&ICONST_1, &&IADD,...}
void **pc = program;  /* pointer to the address of bytecode implementation */

/* bytecode implementations */
goto **(pc++);                                  /* Fetch */
ICONST_1: *++sp = 1; goto **(pc++);    /* Execute bytecode, Fetch, update pointer */
ICONST_2: *++sp = 2; goto **(pc++);
IADD:  sp[-1] += *sp; --sp; goto **(pc++);
... /* Other cases */
}
```

Figure 1: Interpretation schemes.

Threaded code interpreter [2, 8] is known to be the most efficient technique for reducing the bytecode dispatch cost. The instructions in the program to be interpreted are replaced by by the address of the routine that implements them. The interpretation process consists of fetching this address and branching to the routine. An example of a threaded code interpreter is illustrated in the right portion of Figure 1. Another advantage of threaded code is that it allows interpreters to be written in a very portable way using high level languages that offer a way to produce indirect jumps.

JIT compilers produce at least 2 to 10-fold performance speedup compared to a switch-based interpreter while the threaded code technique leads to 10% speedup.

In this paper we evaluate existing techniques and propose new methods for optimizing the performance of Java interpreters with patterns of Java bytecode operations or Super-Operators. Techniques based on SOs are orthogonal to implementing threading and should further boost the efficiency of a threaded code interpreter.

The examples in Figure 3 illustrate how combining bytecode operations into patterns can lead to optimized interpreted execution. The upper side of Figure 3 shows the execution trace of an `ILOAD` operation immediately followed by an `ISTORE` operation. A total of 20 instructions are executed. The trace in the bottom of Figure 3 shows the situation in

```
DISP    .EQU    256         ! Maximum size of each opcode implementation
SDISP   .EQU    8           ! log_2 DISP
METHOD  .REG    (%i0)       ! Method structure
ORIGIN  .REG    (%i1)       ! Beginning of opcode implementations
PC      .REG    (%i2)       ! Address containing the current bytecode
TOP     .REG    (%i3)       ! Operand stack top
LOCALS  .REG    (%i4)       ! Local variables
TARGET  .REG    (%i5)       ! Next opcode to execute
FP      .REG    (%l0)       ! Java stack frame pointer
POOL    .REG    (%l1)       ! Resolved pool
FAKEI   .REG    (%l2)       ! Instruction which trampolines start with
...
.macro DECLARE opcode
\(\(.org)) _interpret_start + \opcode * DISP
.endm
...
! void interpret (Method *m, void *args, void* bcode)
! m is the method to be interpreted
! args is the memory containing arguments; the return value also goes in here.
! bcode is Bytecode address to execute

interpret:
                ...
                ! Initialize registers, e.g., %i1
                sethi %hi(_interpret_start), ORIGIN
                or ORIGIN, %lo(_interpret_start), ORIGIN
                ...
_interpreter_start:

                DECLARE 0 !NOP
                ...
                DECLARE 1 ! ACONST_NULL
                ...
                ! Load word from local variable and push onto operand stack
                DECLARE 21              !ILOAD
                ! Read next bytecode to be executed
                ldub [ PC + 2], TARGET
                ! Execute the current bytecode semantics
                ldub [PC +1], %o2                   ! Read index operand
                sll %o2, 2, %o2
                neg %o2
                ld [LOCALS + %o2], %l3              ! Read local variable at index %o2
                st %l3 [ TOP - 4 ]                  ! Save local variable on the stack
                add TOP, -4, TOP                    ! Update stack pointer
                !Transfer control to the next bytecode
                sll TARGET, SDISP, TARGET           ! Calculating address of next bytecode
                jmp ORIGIN + TARGET
                add PC, 2, PC                       ! Updating PC, delay slot
                ...
                DECLARE 201     ! JSR_W
                ...
_interpreter_end:
                ...
```

Figure 2: Bytecoded interpreter loop example.

Separate Translation: ILOAD followed by ISTORE

```
(1)   ldub [ PC + 2], TARGET          ! Read next bytecode to be executed
(2)   ldub [PC +1], %o2               ! Read ILOAD index operand
(3)   sll %o2, 2, %o2
(4)   neg %o2
(5)   ld [LOCALS + %o2], %l3          ! Read local variable at index %o2
(6)   st %l3 [ TOP - 4 ]              ! Save local variable on the stack
(7)   add TOP, -4, TOP                ! Update stack pointer
(8)   sll TARGET, SDISP, TARGET       ! Calculating address of next bytecode
(9)   jmp ORIGIN + TARGET
(10)  add PC, 2, PC                   ! Update PC, delay slot
(11)  ldub [ PC + 2], TARGET          ! Read next bytecode to be executed
(12)  ldub [PC +1], %o2               ! Read ISTORE index operand
(13)  sll %o2, 2, %o2
(14)  neg %o2
(15)  ld  [ TOP], %l3                 ! Load value from the stack
(16)  st  %l3, [LOCALS + %o2]         ! Store value in  local variable at index %o2
(17)  add TOP, 4, TOP                 ! Update stack pointer
(18)  sll TARGET, SDISP, TARGET       ! Calculating address of next bytecode
(19)  jmp ORIGIN + TARGET
(20)  add PC, 2, PC                   ! Updating PC, delay slot
```

Combined Translation: ILOAD followed by ISTORE

```
(1)   ldub [ PC + 4], TARGET          ! Read next bytecode to be executed
(2)   ldub [PC +1], %o2               ! Read ILOAD index operand
(3)   sll %o2, 2, %o2
(4)   neg %o2
(5)   ld [LOCALS + %o2], %l3          ! Read local variable at index %o2
(6)   ldub [PC + 3], %o2              ! Read ISTORE index operand
(7)   sll %o2, 2, %o2
(8)   neg %o2
(9)   st  %l3, [LOCALS + %o2]         ! Store value in  local variable at index %o2
(10)  sll TARGET, SDISP, TARGET       ! Calculating address of next bytecode
(11)  jmp ORIGIN + TARGET
(12)  add PC, 4,  PC                  ! Updating PC, delay slot
```

Figure 3: Traces of the translations of an `ILOAD` followed by an `ISTORE` operation.

which at the moment of translating the `ILOAD` operation it was known that such instruction was followed by an `ISTORE` operation. In this new trace, the dispatch cost for the `ISTORE` operation is eliminated (instructions 8, 9, 10 and 11 from the upper trace). Besides, the stack accesses and stack pointer updates in the translation of both operations are completely eliminated (instructions 6, 7 and 15 in the upper trace). The result produced by the `ILOAD` operation is kept in the machine register `l3` and reused by the subsequent `ISTORE` bytecode, with no need to access the stack. The combined translation of the two bytecodes executes 12 instructions, a 40% improvement over the separate translation.

This paper is organized as follows. Section 2 discusses related solutions to cope with program interpretation issues. In Section 3, we evaluate the importance of different types of SOs formed by basic blocks and simpler operation folding patterns. In Section 4 we carry out a comparative study of techniques for optimizing interpreters with SOs. Our SO-aware customization techniques for the direct interpretation of Java bytecode programs are

discussed in Sections 5 and 6. In Secion 7 we list the SOs implemented in different interpreter versions. Finally, our main conclusions and future work direction are summarized in Section 8.

The current report summarizes our main findings in this area. A more detailed description of this work can be found in the main author's Ph.D. dissertation [1].

2 Related Work

The breakthrough in the efficient implementation of virtual machine interpreters is the threaded code technique [2, 7, 8]. The threaded interpreter still pays the cost of an instruction dispatch for each bytecode executed. If simple bytecodes are combined into bytecode sequences, the dispatch overhead is reduced.

Optimizing interpreters with bytecode sequences has been tried in previous research. Proebsting's work on SuperOperators [16] introduces SOs as specialized instructions automatically inferred from repeated patterns in the tree-like intermediate representation produced by *lcc* compiler. His bytecoded interpreter extended with SOs runs 2 to 3 times faster with the tested benchmarks.

Ertl, Gregg et al [11, 9] have combined the advantages of threaded code interpreter with the merging of single instructions into *Superinstructions*. By inspecting traces of a program execution, patterns of instructions of length 2, 3 and up to 4 are detected. In a later phase, the behavior of the original virtual machine operations and the patterns of instructions are defined using a special syntax in C. An automatic interpreter generator takes in this specification and outputs an interpreter in C that implements the described behaviors. Their work relies on a smart C compiler to remove redundant stack accesses, unnecessary stack pointer updates and bytecode dispatch instructions within patterns. Hundreds of patterns are incorporated to the interpreter code, substantially increasing the size of the interpreter. Up to 2 fold-speedups have been reported for the indirect interpretation of Java bytecodes.

Piumarta et al propose a technique that eliminates the dispatch overhead within a basic block using *selective inlining* [15]. The code to be interpreted is first translated to threaded code and basic blocks are identified. A second pass dynamically generates macro opcodes representing the basic blocks and replaces threaded code opcodes with the macro opcodes. The implementation of each macro is a simple concatenation of the C-code implementations of the bytecodes that it replaces. The technique was applied to the Objective Caml bytecode interpreter and resulted in 50% average speedup, reaching twice as fast in some cases.

Thibault et al. [22] proposes interpreter *Specialization* as a more generic solution for optimizing interpreters than Piumarta's. An interpreter specialized for a particular program is essentially a concatenation of the implementations of all the bytecodes in the program. This technique fully eliminates the bytecode dispatch cost resulting in 4-fold speedups.

Sun designed a stack operation folding mechanism for PicoJava I [20] and PicoJava II [21] architectures that converts many cycles of stack oriented instructions into an one-cycle register based instruction, which can be implemented with a few registers. This technique groups or folds contiguous operations that have true data dependency. For example, the bytecode sequence `iload_1 iload_2 iadd istore_3` (two stack copy operations, an ALU operation

and a local variable store operation) can be transformed into a single `add R1, R2, R3` operation. Sun's folding technique is based on pattern matching with a very limited set of patterns. A special decode unit in the PicoJava processor converts the bytecode instructions into micro-operations and looks for folding patterns up to 4 consecutive instructions long. If a pattern is identified, the decoder replaces the instructions belonging to the pattern with a simple micro-operation. Other stack operation folding techniques that enhance PicoJava's simple grouping rules have been proposed in literature [4, 23, 5].

Kim [17] has proposed a software approach to stack operation folding in which he creates a *Smart Loader*, a custom class loader that finds folding patterns at load-time by applying PicoJava-like grouping rules. With this approach, the JVM hardware can be simplified yet benefit from folding.

Our work differs from previous research mainly in the concern of finding out the maximum number of SOs required to guarantee the bulk of the performance improvement, while limiting the number of SOs that need to be implemented in the interpreter. This is not done in [9]. We also propose profiling and an annotation scheme as solutions that free the runtime system from identifying the patterns of bytecodes that should be optimized and that allow the interpreter to be customized for a particular application. In [15], all program basic blocks are subject to inlining with no prioritizing. Although the work in [9] resorts to profiling to find patterns of instructions, the patterns are not custom-designed for a specific application. Another goal of ours is to assess the value of operation folding patterns in customizing software interpreters. To our knowledge this has not been attempted before.

3 Evaluating SO Types

The most intuitive way of looking for valuable patterns of bytecodes is to identify basic blocks (BBs) in Java bytecode programs, profile them, calculate their dynamic execution frequencies, prioritize them and form SOs with the top most important basic blocks.

We do not study patterns of instructions that include control transfer instructions, except when it is the last instruction in the pattern. Therefore, when identifying basic blocks, method invocations, conditional jump bytecodes (e.g., `if_icmpeq`), unconditional branch bytecodes (e.g., `goto`), compound conditional branch bytecodes (e.g., `tableswitch`, `lookupswitch`) and the bytecodes associated with the implementation of the `finally` keyword (`jsr` and `ret`) all terminate the basic blocks.

For the experiments we describe in this paper we analyze several programs from SPEC JVM98 [19] and Java Grande Forum (JGF) [6] benchmark suites. The latter suite is organized in 3 different sections. Of interest to us are Section 2 (labeled JGF S2), a collection of scientific and numerical application kernels, and Section 3 (labeled JGF S3), which is made up of full-scale science and engineering applications.

The second column in Table 1 lists the total number of SOs formed by basic blocks that are executed in each benchmark. To find out the priority among the high number of SOs in Table 1 we have to resort to heuristics. We decided to prioritize SOs by their averaged

Benchmarks	SO formed by BB	SO formed by OF pattern	SO formed by BB $dynamicF > 1\%$	SO formed by OF pattern $dynamicF > 1\%$
compress	179	331	25	45
db	170	200	15	19
jess	792	963	21	28
mtrt	510	604	15	16
Crypt	68	107	44	25
FFT	72	130	20	49
HeapSort	43	55	6	11
LUFact	92	155	2	2
Series	50	67	7	8
SOR	38	53	2	4
SparseMatmul	42	51	2	4

Table 1: SOs in SPEC JVM98 and JGF S2 benchmarks.

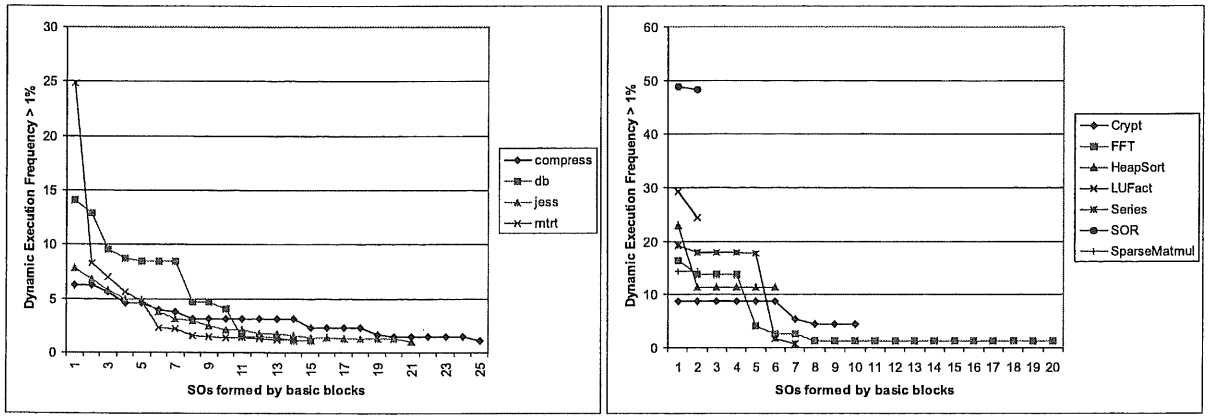

Figure 4: Dynamic frequencies for SOs formed by BBs in SPEC JVM98 and JGF S2 suites.

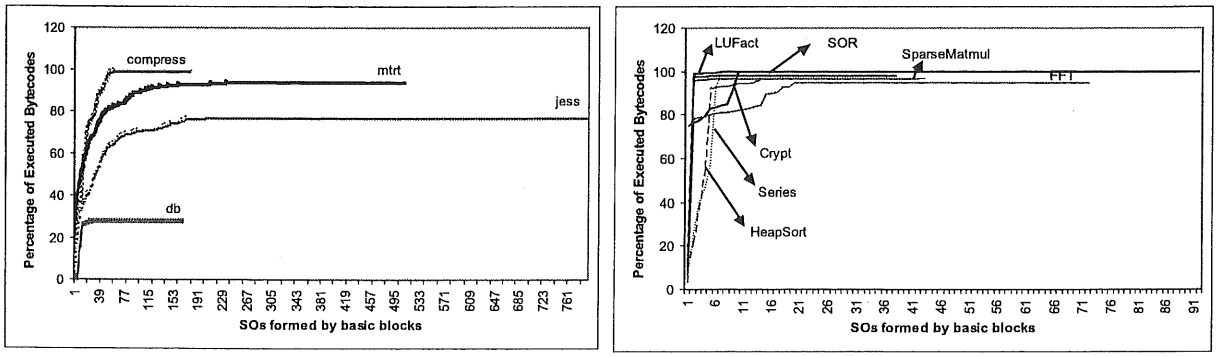

Figure 5: Bytecodes covered by all SOs formed by BBs in SPEC JVM98 and JGF S2 suites.

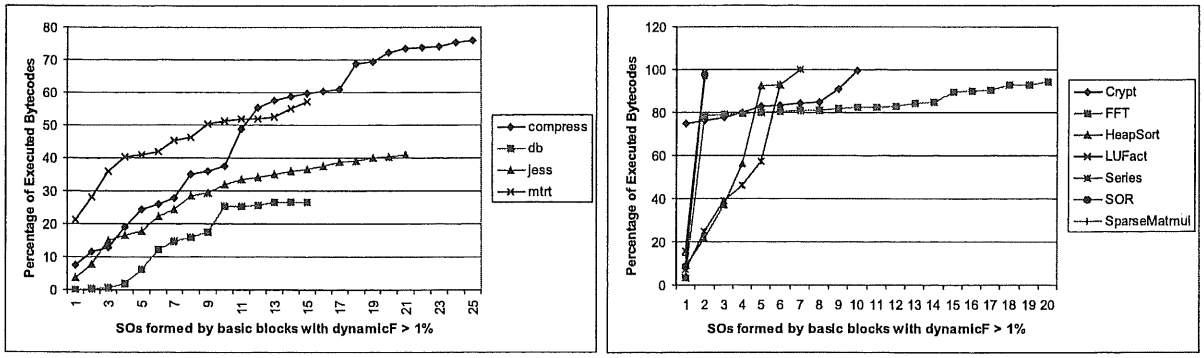

Figure 6: Bytecodes covered by top SOs formed by BBs in SPEC JVM98 and JGF S2 suites.

dynamic execution frequency $dynamicF_i$, which is is calculate as

$$dynamicF_i = \frac{\sum_1^n \frac{100 * DF_{ik}}{\sum_1^t DF_{ik}}}{n}$$

where DF_{ik} is the number of times SO i is executed during the execution of program k; t is the total number of SOs in the static code of program k; and n is the total number of programs averaged over or the total program runs for different datasets.

The graphs in Figure 4 plot all the SOs with dynamic frequencies higher than 1% for programs from SPECJVM98 and JGF S2 suites. The total number of SOs with $dynamicF_i$ higher than 1% is listed in the fourth column of Table 1. From Figure 4 we can notice that, for most of the benchmarks, the first 2 SOs bear the highest execution frequencies. The dynamic frequencies of the following SOs drop abruptly to lower than 5% when reaching SOs higher than the sixth.

The dynamic frequency value by itself only gives an initial hint for the most valuable SOs. Further insight can be obtained by calculating the percentage of the total number of bytecodes executed by an application that are covered by SOs. The total number of executed bytecodes also factors in bytecodes executed from standard library methods. However, in the next study we only account for SOs that appear in methods defined in an application program. For SPEC JVM98 suite, application code accounts for 83 to 99% of the total bytecodes executed, except `db`'s methods that account for 30%. For JGF suite, 94 to 99% of the total bytecodes executed are from methods defined in the benchmarks. We do not evaluate the contribution of SOs that exist in method bytecodes from the Java library. In some platforms, standard library methods are available in native code and do not need to be translated.

Figure 5 depict graphs with the accumulative percentage of the total executed bytecodes covered as the bytecodes corresponding to each SO are counted in. In this figure, we account for all SOs listed in the second column of Table 1. The percentage values do not reach 100% because we only formed SOs with basic blocks longer than one bytecode operation and SOs that appear in methods defined in the application programs.

The graphs in Figure 6 plot the percentage of total executed bytecodes covered by only the SOs with dynamic frequencies higher than 1%. These graphs should be read referring

back to the graphs in Figure 5. For example, the top 21 SOs in `jess` account for 41% of the total bytecodes executed by the application. If all SOs in `jess` are counted in, Figure 5 reveals that they account for a maximum of 77% of the total bytecodes executed. Therefore, the top 21 SOs in `jess` cover 53% (41/77) of that maximum.

When calculating this last percentage value for all the programs, we find values varying from 53 to 96% for SPEC JVM98 benchmarks and 96 to 100% for JGF S2 suite. This result indicates that if we optimize an interpreter with only the top 20 SOs, we are improving the execution of 53 to 100% of the bytecodes executed in the program basic blocks. It is also a relevant piece of information for restricting the interpreter size.

Our experience has shown that the interpreter size can significantly affect program performance. For example, changing the size of the bytecode implementation from 256 to 512 bytes in LaTTe's interpreter causes performance slowdown of up to 11% due to cache conflict misses. This behavior was observed on a 500MHz UltraSPARC IIe processor with a 16 Kb L1 instruction and data caches and 256 Kb L2 unified cache. Therefore, adding a high number of SOs to the interpreter loop can actually degrade performance when the savings the SOs produce are overweighed by the performance degradation caused by cache misses.

If we can find SOs that are valuable across several benchmarks we can further reduce the number of SOs that need to be implemented and yet optimize the interpreter for a large range of applications. When searching for SOs that match across all the studied benchmarks in SPEC JVM98 suite and across all programs in JGF S2 suite we found very few exact matches. A more detailed look into the sub-patterns that form the basic blocks revealed that basic blocks are formed by simpler patterns that can be detected by stack operation folding techniques, usually implemented in hardware JVMs. Our experiments showed that such sub-patterns repeat across benchmarks more frequently than the larger, basic-block based SOs. Therefore we decided to investigate the importance of such patterns.

Figure 7 shows the main loop in the kernel of JGF S2 SOR benchmark with some examples of OF patterns. It also lists part of the disassembled bytecodes 79 to 123 that correspond to the array operations in the innermost loop. These bytecodes form the SO with the second highest execution frequency in SOR. The 5 occurrences of PicoJava-like operation folding patterns are highlighted in boldface.

The third column of Table 1 lists the large number of OF patterns that were detected in each benchmark while the last column shows the total number of OF patterns with dynamic frequency higher than 1%.

We further investigated the value of SOs formed by OF patterns by analyzing the percentage of executed bytecodes they cover. Figure 8 shows the percentage of executed bytecodes when accumulatively accounting for the bytecodes corresponding to the SOs formed by OF patterns for both benchmark suites. In SPEC JVM98 suite, all SOs formed by OF patterns cover 13 to 50% of the total executed bytecodes. In JGF S2 suite, the percentage value varies from 19 to 57%. These results are roughly half of the percentage values when using SOs formed by basic blocks, shown in Figure 5.

In Figure 9 we plot the percentage of executed bytecodes covered by SOs formed by OF patterns that are found as sub-patterns in the earlier examined basic blocks with dynamic frequencies higher than 1%. For SPEC JVM98 benchmarks, from Figure 9 we can calculate that up to 45 SOs account for 62 to 92% of the maximum total executed bytecodes covered

Java Code

```
void SORrun(double, double[][], int) public static final
...
for (int p=0; p<num_iterations; p++) {
  for (int i=1; i<Mm1; i++){
    double [] Gi = G[i];
    double [] Gim1 = G[i-1];
    double [] Gip1 = G[i+1];
    for (int j=1; j<Nm1; j++){
    Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1]
          + Gi[j+1]) + one_minus_omega * Gi[j];
    }
  }
}
...
```

Java bytecode	
79: aload 14	
81: iload 17	
83: dload 6	
85: aload 15	**SO1**
87: iload 17	
89: daload	
90: aload 16	**SO1**
92: iload 17	
94: daload	
95: dadd	
96: aload 14	
98: iload 17	**SO2**
100: iconst_1	
101: isub	
102: daload	

103: dadd	
104: aload 14	
106: iload 17	**SO3**
108: iconst_1	
109: iadd	
110: daload	
111: dadd	
112: dmul	
113: dload 8	
115: aload 14	**SO1**
117: iload 17	
119: daload	
120: dmul	
121: dadd	
122: dastore	
123: iinc vindex: 17 by: 1	

Figure 7: Example of top SO in SOR benchmark highlighting OF patterns.

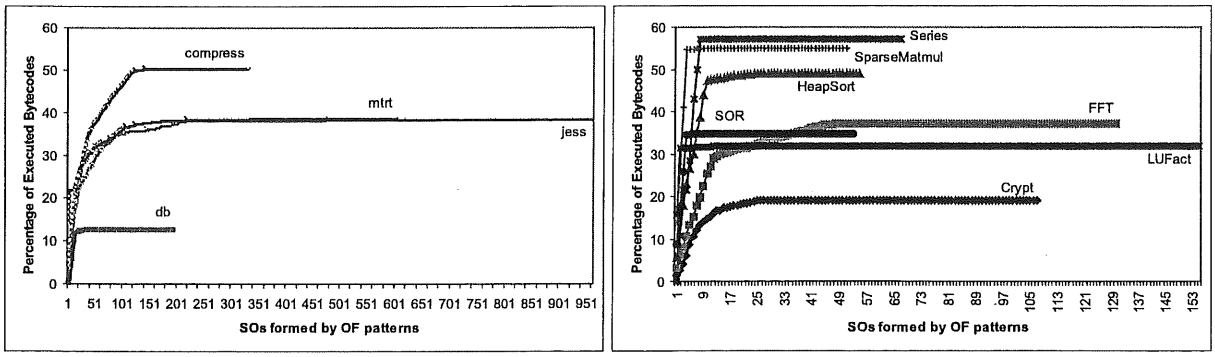

Figure 8: Bytecodes covered by all SOs formed by OF patterns in SPEC JVM98 and JGF S2 suites.

by all OF patterns shown in Figure 8. For JGF S2 benchmarks, the graph reveals that almost 100% of the maximum total executed bytecodes shown in Figure 8 are covered by less than 10 SOs, except for `Crypt` and `FFT`.

The results reported in this section demonstrate that an interpreter optimized with the top 20 SOs formed by basic blocks per application will improve the translation of 50% to almost all executed bytecodes. If we are interested in SOs that are valuable across applications, looking for operation folding patterns that form the basic blocks is an alternative that produces more matches. However, the total executed bytecodes such SOs cover is up to 57% of the total bytecodes the applications execute. Although an interpreter optimized with SOs formed by OF patterns can improve the performance of a broader range of applications, the speedups it will produce will be lower compared to techniques that deploy sets of SOs formed by basic blocks.

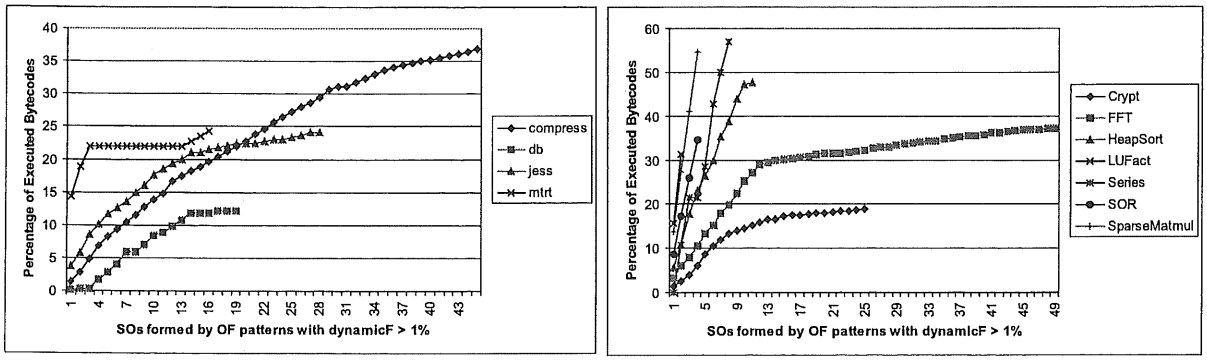

Figure 9: Bytecodes covered by top SOs formed by OF patterns in SPEC JVM98 and JGF S2 suites.

4 Comparing SO-based Techniques

Previously proposed techniques for optimizing interpreters with patterns of instructions have been implemented in either different virtual machines other than JVMs or use different intermediate languages other than Java bytecode. In order to compare the techniques under the same virtual machine engine we designed three Java SO annotation-aware interpreter versions, building on top of LaTTe's interpreter. The description of the optimizing techniques implemented by each interpreter version is given below.

1. **Version BBOpt: Optimizing top SOs formed by basic blocks**
   In this interpreter version, which we label BBOpt, the most valuable SOs are fully optimized, i.e., the bytecodes are concatenated eliminating the dispatch cost. Additionally, we implement common sub-expression elimination and copy propagation at basic block level to eliminate unnecessary stack accesses and stack pointer updates.

2. **Version BB: Concatenating top SOs formed by basic blocks**
   The interpreter version labeled BB simply combines the translation of the individual bytecodes that compose the most important SOs. It eliminates unnecessary operations that implement bytecode dispatch.

3. **Version OF: Optimizing top SOs formed by Operation Folding patterns**
   Finally, the interpreter version labeled OF fully optimizes SOs as in the BBOpt version, but SOs are formed by operation folding sub-patterns that appear in the most executed basic blocks.

The techniques we designed can be seen as counterparts of Piumarta's and Ertl's that enable direct interpretation of programs in Java bytecode. Piumarta's technique is equivalent to what we implement in the BB version while Ertl's technique falls between what is implemented in the BB and BBOpt versions.

In each interpreter version we deploy only the top 3 SOs for some of the benchmarks studied in Section 3. The SOs are added to the interpreter loop using the non-patented

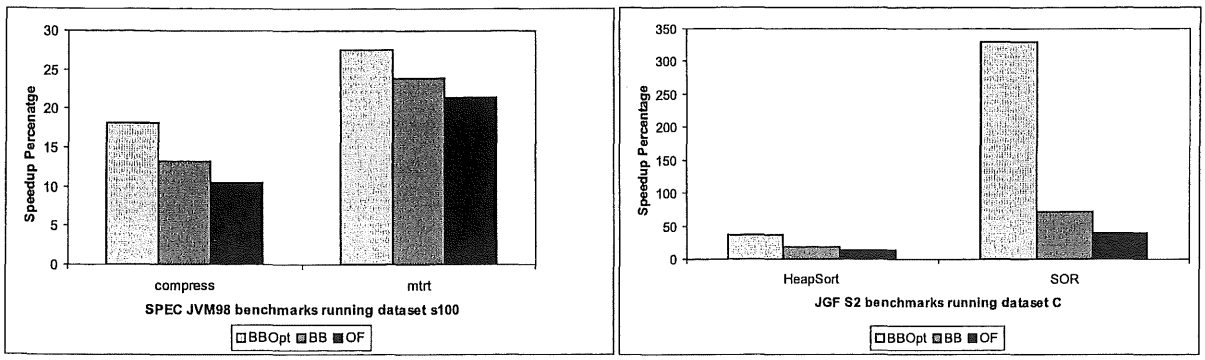

Figure 10: Comparing Techniques for optimizing interpreters with SOs.

Java bytecode opcodes. The graphs in Figure 10 show how faster the SO-aware interpreter versions are compared to the performance of LaTTe's original interpreter. The speedups are shown when running the largest datasets (s100 for SPEC JVM98 suite and C for JGF suite).

The 3 SOs formed by basic blocks in `compress`, `mtrt`, `HeapSort` and `SOR` cover 13, 36, 37 and 97.5% of the total executed bytecodes in each program respectively. The 3 SOs formed by OF patterns for the same benchmarks account for 5, 22, 18 and 26% of the total executed bytecodes.

As expected, interpreter version BBOpt bears the best performance. The interpreter version BB also outperforms OF. Benchmark `SOR` yields the best speedups in all versions, running 330% faster on BBOpt. It produces such good results because the top 3 SOs in this benchmark cover 97.5% of the total executed bytecodes. The other benchmarks, which deploy SOs that cover much less of the total executed bytecodes, we do not notice markedly distinct performance levels among the interpreter versions. The performance level observed with the more complex BBOpt and BB versions is almost all achieved with the simpler OF interpreter version. For example, executing `mtrt` on BBOpt, BB and OF produces close speedups of 27.5%, 24% and 21.5%.

The main difference between BBOpt version and BB is the fact that we removed the redundant stack accesses that the semantics of Java bytecode operations require to copy values onto the stack and to store values to local variables and on the heap. As can be noticed in Figure 10 these operations compromise performance much more than the interpreter trips the BB version is able to eliminate. The OF version corresponds to a software implementation of well known stack operation folding techniques. Though bearing the lowest speedups it can produce performance improvement comparable to the other more expensive techniques when SOs are customized per application, as implemented in this experiment.

We conclude that a highly efficient technique for optimizing interpreters with patterns of bytecodes should also optimize stack accesses and stack pointer updates. As the BBOpt version implies more work, specially if patterns are translated at runtime, applying it to up 20 of the most valuable SOs should guarantee performance superior to that of current art with lower overhead.

5 Statically Customizing an Interpreter

In an interpreter that directly translates Java bytecode programs we are limited in the number of new operations we can add to the interpreter loop. A total of 202 opcodes are taken by Sun's bytecodes opcodes, and 3 more are reserved for special use. The rest non-patented bytecode opcodes are free for the interpreter internal use. Some of these opcodes are already used to implement *quick variants* [13] of normal JVM operations. In LaTTe's interpreter design, we are left with 21 free opcodes that we use to represent our SOs.

Our technique for statically customizing an interpreter adds a limited number of SOs to the interpreter loop as new pseudo-instructions which are only visible from within the interpreter code. We built interpreter versions with a SO set that is valuable across both SPEC JVM98 and JGF benchmark suites, with a SO set customized per benchmark suite and a third kind that implements a SO set customized per application.

We chose to implement sets of SOs formed by the top most executed OF patterns in the applications. Such patterns compose program basic blocks and repeat across benchmark suites more often than whole basic blocks. We extended the folding pattern classes supported in PicoJava architecture to include floating-point operations and other complex operations (e.g. loads from the Constant Pool [13]) that have been excluded in hardware implementations.

Although we know interpreters optimized with SOs based on OF patterns offer limited performance improvement, this kind of implementation has not been tried before and no bound on speedups have been published, except for hardware JVMs.

We designed an annotation scheme that uses an extra code attribute to carry the information on SO occurrences in the bytecode stream. In this attribute we store an annotation table that associates the pc addresses of a method, represented as short values, with their corresponding byte-long SO identification number. The annotations do not hinder the portability of Java class files as it can be safely ignored by JVM engines that do not understand the attribute. Such annotations encoding leads to 5.5, 6.8 and 10.5 code size increase for SPEC JVM98, JGF S2 and S3 suites.

We built 4 interpreter versions with SOs customized for different sets of application programs. In Section 7 we list the sets of SOs implemented in each version. LaTTe's original interpreter code is used as the baseline when calculating speedups.

Version SOOFDF implements the top 21 SOs across *all* benchmark suites. Its performance is depicted in Figure 11. Average speedups of 7.2, 7.7 and 6.2% are noticed for SPEC JVM98, JGF S2 and JGF S3 suites. The maximum speedups are 15, 20.3 and 13.5% respectively for each suite. Slowdown situations occur for `db`, `LUFact` and `Series`. Except for `LUFact`, these are the benchmarks with largest data cache miss rates, which indicates that our 16 Kb L1 data cache and 256 Kb L2 unified caches do not handle well these applications demands.

Versions SOOFDFJVM98 and SOOFDFJGFS2 are customized per benchmark suite and implement the top 21 SOs in SPEC JVM98 and JGF S2 suites respectively. Figure 12 shows the performance results. SOOFDFJVM98 interpreter version produces average speedup of 5.2%, and a maximum speedup of 17%. SOOFDFJGFS2 version leads to average speedup of 7.6% and a maximum of 29%. These average speedups are comparable to that of SOOFDF

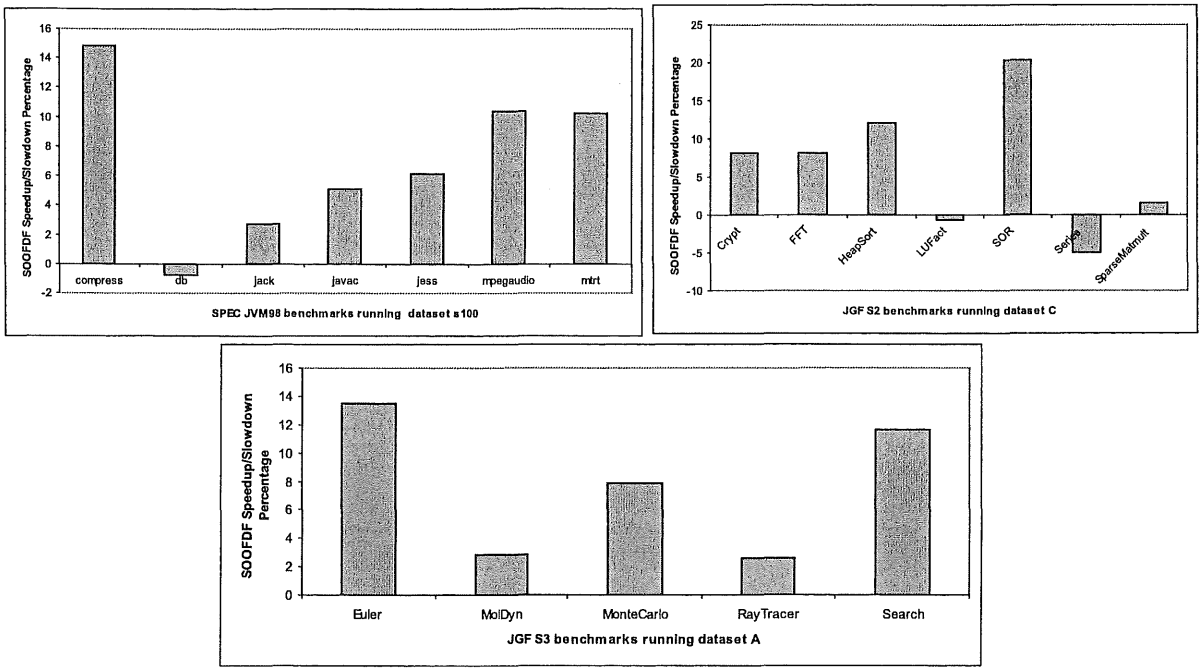

Figure 11: SOOFDF performance running SPEC JVM98 and JGF suites.

version. Nevertheless, the fact that customizing the set of SOs per suite does help individual applications significantly, it should be preferred in the design of SOs. For example, instead of finding the most frequent SOs across all applications concomitantly, as we did when building the SOs in SOOFDF version, a better heuristic would first find the top most frequent SOs for each suite, and then use the selected ones per suite to generate a combined set of SOs.

Versions SOOFDFSOR and SOOFDFSeries are customized with the top 21 SOs in `SOR` and `Series` from JGF S2 benchmark suite. `SOR` is among the benchmarks that produced the highest speedups when running on SOOFDFJGFS2 version, while `Series` led to insignificant performance improvement running on that same version. In Figure 13 we compare the performance of the SOOFDFSOR and SOOFDFSeries versions against the speedups obtained with the interpreter version customized across all applications (SOOFDF) and the version specialized for the execution of JGF S2 benchmark suite (SOOFDFJGFS2). `SOR`'s maximum speedup is 39.6%, almost doubling the performance we first obtained when running it on SOOFDF version. Despite the highly customized interpreter built for `Series`, the program still performs poorly, producing marginal speedup of 3%. We attribute the lack of performance to limitations of the hardware we execute the application on.

In this section we showed the performance speedups one can expect from optimizing an interpreter with a limited number of SOs formed by operation folding patterns. Such SOs yield speedups varying from 5% to 39%, depending on the customization level.

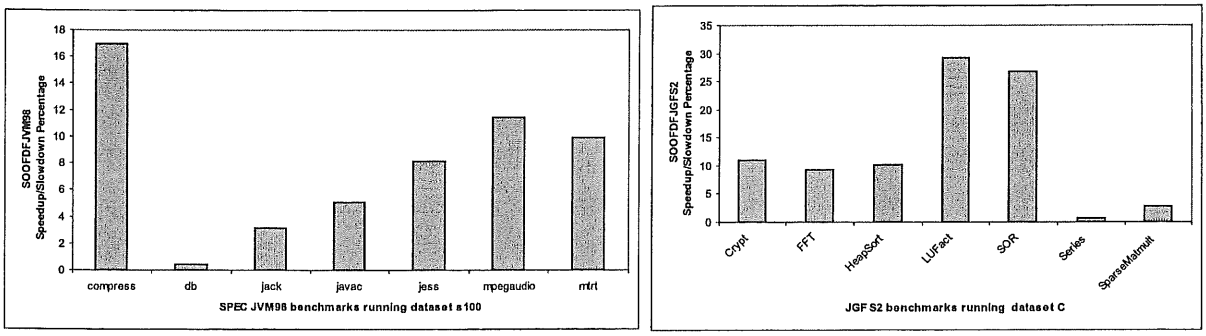

Figure 12: SOOFDFJVM98 and SOOFDFJGFS2 performance running SPEC JVM98 and JGF S2 suites.

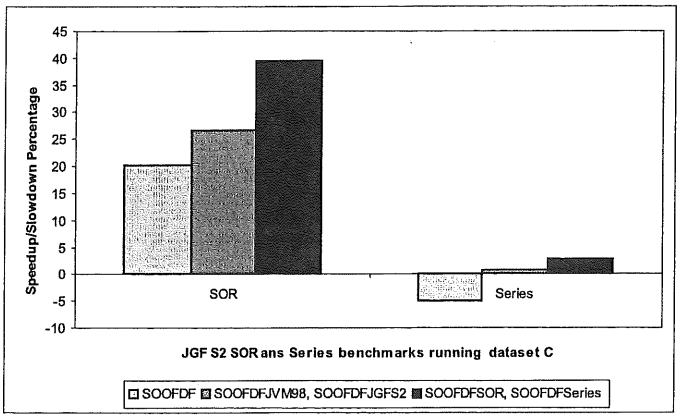

Figure 13: SOOFDFSOR and SOOFDFSeries performance running `SOR` and `Series`.

6 Dynamically Customizing an Interpreter

In Section 5, the best interpreter version optimized with SOs formed by OF patterns is the one customized per application. To statically customize an interpreter to the level of an individual application is very restrictive and makes sense for some specialized embedded systems. To try to combine the per-application customization advantages with the convenience of a SO-aware interpreter that is efficient across several classes of applications, we created the dynamically customized interpreter. With this technique SOs are generated for each application individually and are annotated ahead of time. Another added advantage over the statically customized versions is the fact that the SOOFDYN interpreter can be specialized independently of the non-patented opcodes available in the interpretation engine.

The annotation scheme in SOOFDYN version is similar to that of the static case, with an added byte that represents the type of each SO. This extra field results in a slightly higher code size increase than that reported in Section 5. The type information guides the SO translation at runtime. We wrote special functions for translating the 9 OF pattern classes defined for PicoJava architecture. These functions fully optimize stack accesses and stack pointer updates, and eliminate unnecessary bytecode dispatches.

The interpreter loop in the dynamic case, depicted in Figure 14, is kept essentially the same as in the original LaTTe's code (except for some extra operations to control the dynamic scheme). We only need one non-patented opcode to represent all possible SOs (opcode 203 in Figure 14). At runtime, as a SO is identified in the bytecode stream, a call to a software translation function produces the machine instructions that implement the SO semantics. The translated code is stored in a native code table, pointed by SONCODE, which is also indexed by the SO identification number. This table is shared by all the methods in a program. Each bytecode in a method that corresponds to a SO occurrence is associated with a pointer to an entry in the native code table. Future executions of instances of a previously translated SO can skip the translation process and jump straight to this pointer to execute the SO implementation. The mechanism described above is represented in Figure 15. Figure 14 illustrates how a dynamically customized interpreter invokes a SO after it has been translated.

The graphs in Figure 16 plot the performance of the SOOFDYN interpreter version. In this experiment the annotations mark *all* SO occurrences. Average speedups of 5, 11 and 8% were measured for SPEC JVM 98, JGF S2 and S3 suites. Maximum speedups are 16, 19 and 24% for each suite respectively. Slowdown situations occur when executing `mtrt`, `Series` and `Euler`.

For SPEC JVM98, the average speedups with the dynamic approach are comparable to the performance levels obtained when running on SOOFDF and SOOFDFJVM98 statically customized interpreter versions. JGF S2 and S3 benchmarks perform slightly better when running on the dynamically customized interpreter version than on either SOOFDF or SOOFDFJGFS2 interpreters. The performance results can be further explained by understanding the 2 sources of overhead in the dynamic case.

There is a time overhead associated with translating SOs at runtime. In this experiment we show the worst case scenario in which SOOFDYN version translates all the annotated SOs in the methods that are loaded. Not all translated SOs are executed, but the percentage

```
DISP        .EQU   512    ! Maximum size of each opcode implementation
SDISP       .EQU   9       ! log_2 DISP
METHOD      .REG   (%i0)   ! Method structure
ORIGIN      .REG   (%i1)   ! Beginning of opcode implementations
PC          .REG   (%i2)   ! Address containing the current bytecode
TOP         .REG   (%i3)   ! Operand stack top
LOCALS      .REG   (%i4)   ! Local variables
TARGET      .REG   (%i5)   ! Next opcode to execute
FP          .REG   (%l0)   ! Java stack frame pointer
POOL        .REG   (%l1)   ! Resolved pool
FAKEI       .REG   (%l2)   ! Instruction which trampolines start with
BCBASE      .REG   (%l5)   ! base of bytecode stream
SONCODE .REG   (%l6)    ! SO native code table
...
.macro DECLARE opcode
\(\(.org)) _interpret_start + \opcode * DISP
.endm
...
! void interpret (Method *m, void *args, void* bcode)
! m is the method to be interpreted
! args is the memory containing arguments; the return value also goes in here.
! bcode is Bytecode address to execute

interpret:
                ...
                ! Initialize registers, e.g., %i1
                sethi %hi(_interpret_start), ORIGIN
                or ORIGIN, %lo(_interpret_start), ORIGIN
                ...
_interpreter_start:

                DECLARE 0  !NOP
                ...
                DECLARE 1  ! ACONST_NULL
                ...
                DECLARE 201     ! JSR_W
                ...
                DECLARE 203             ! opcode identifying a SO occurrence
                sub PC, BCBASE, %l3     ! displacement from bytecode begin
                sll %l3, 2, %l4
                add SONCODE, %l4, %l4  ! calculating index into native code table
                ld [%l4], %l4           ! address of the translated code
                jmp %l4                 !  jump to the SO translated code
                nop
                ...
_interpreter_end:
                ...
```

Figure 14: Dynamically customized interpreter.

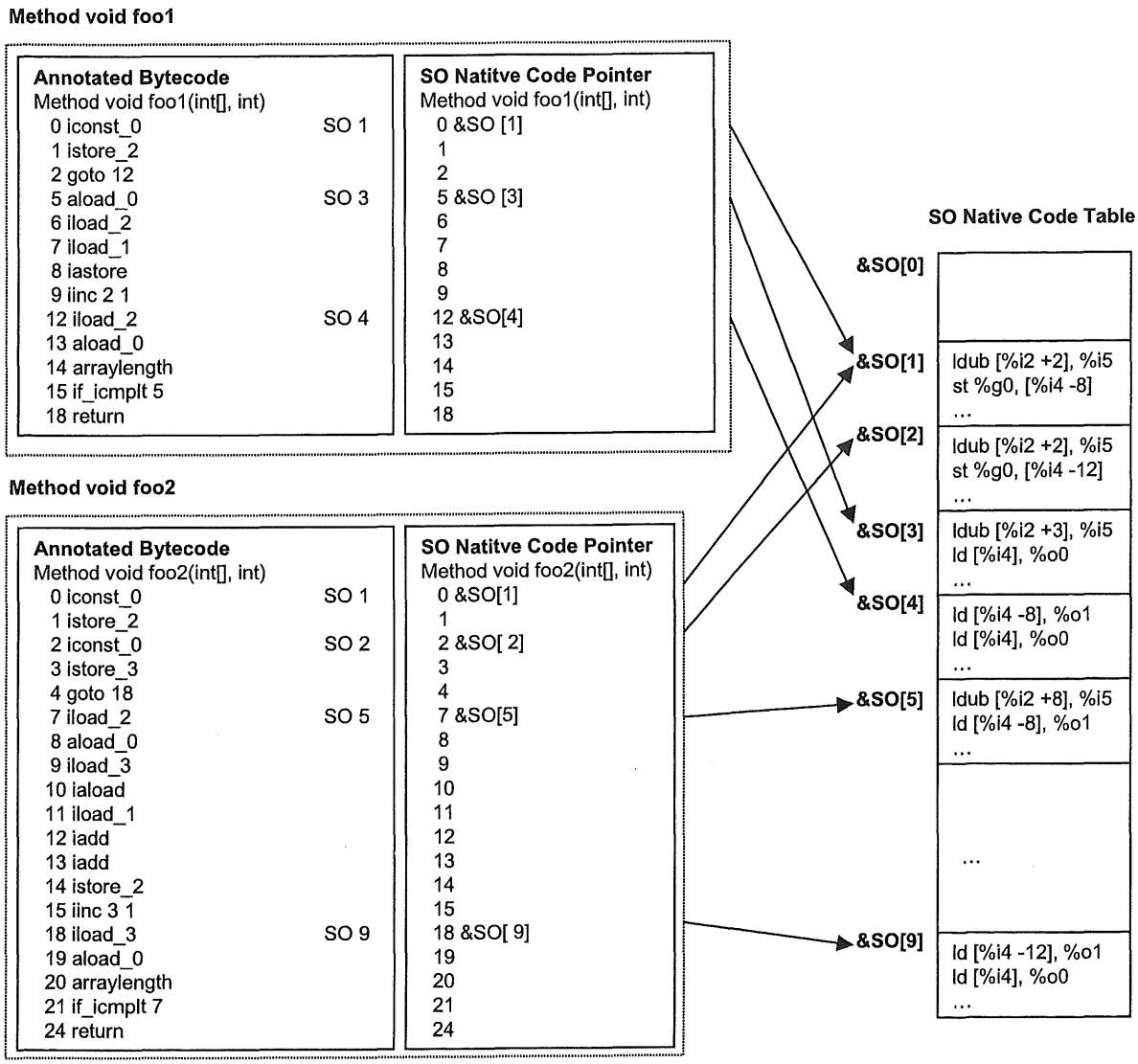

Figure 15: Data structures used in the dynamic approach.

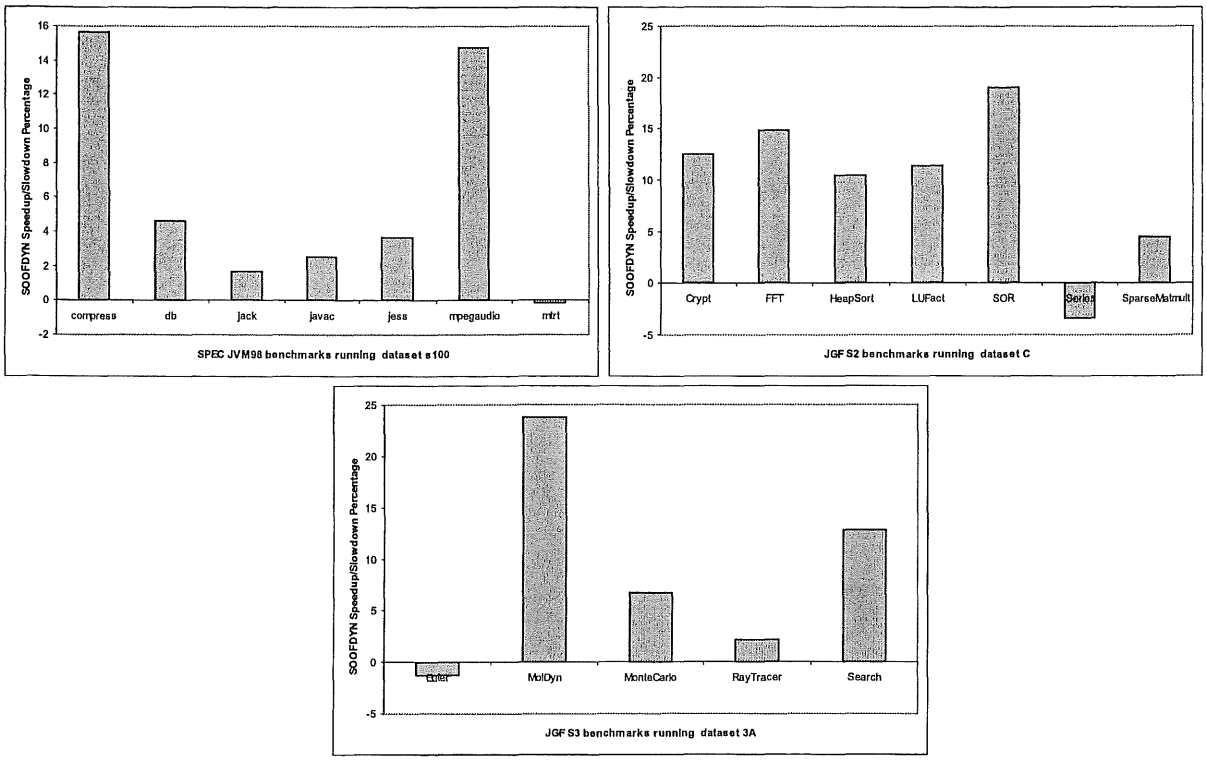

Figure 16: SOOFDYN performance running SPEC JVM98 and JGF suites.

of total translated SOs that are indeed executed is very high: 78% for SPEC JVM98, 100% for JGF S2 and 92% for JGF S3 benchmarks. We measured time overhead of 1% due to the translation of SOs at runtime. With this result we can rule the translation of SOs out as a justification for the slowdown situations and marginal performance improvement we observe in the dynamic case.

The other source of overhead is related with the way SOs are invoked at runtime. In the dynamically customized interpreter there is a double indirection to reach the machine code of a SO. First, the interpreter jumps to the common location where the addresses of all translated SOs are calculated. This action involves 4 instructions and has the cost of a normal bytecode dispatch (discussed when we explained Figure 2). Once at this point, 5 extra operations, shown in Figure 14, which include another load and a jump, are executed to compute the address of the machine code that implements the SO semantics. The cost of executing this group of operations is basically similar to the cost of dispatching a new bytecode. Therefore invoking SOs causes twice the cost of dispatching a new bytecode. For certain types of SOs, deployed by the dynamically customized interpreter, we won't see the benefit of folding their instructions together because any saving the folding produces is outweighed by the cost of invoking the SOs.

In this section we combined the innovation of optimizing interpreters with SOs formed by OF patterns with the benefits of full customization per application offered by the dynamic approach. Unfortunately OF patterns tend to be short and the savings they produce are

overcome by the overhead implicit to the dynamic method. The dynamically customized interpreter will be more useful for longer and more complex SOs, e.g., SOs formed by BBs.

7 Implemented Sets of SOs

7.1 OF Patterns Across Application Suites

Table 2 lists the 21 SOs created from the topmost frequent OF patterns sorted by dynamic frequency. The most frequent pattern is `aload_0 getfield`, as we would expect. Its dynamic frequency is 10 times higher than any other SO listed in Table 2. Java object fields are accessed and updated via `getfield`, `putfield`, `getstatic` and `putstatic` bytecodes. Bytecode `aload_0` refers to the first entry in the local variable array. The very first 4 entries in the local variable array are accessed using fast bytecode implementations. Therefore such local variable slots are oftenly used by efficient Java-to-bytecode compilers.

Many of the patterns listed in the tables throughout Section 7 are semantically equivalent, but with slight variations in the bytecode syntax. Some exmaples are patterns like `aload_0 iload_1 putfield` and `aload_0 aload_1 putfield`, translate into the same machine instructions in our interpreter techniques, and therefore they can be considered as the same pattern instance. Others, like `iconst_0 istore_1` and `iconst_0 istore_3`, lead to different machine code and have to be considered as different SOs. To cope with such peculiarity, after finding patterns, we further *templatize* patterns [10] according to the way our interpreter techniques translate bytecodes. The templatizing phase creates pattern templates that abstract syntactic differences in bytecode opcodes when they are not relevant for our code generation. Templatized patterns are marked with a superscript asterisk in all tables.

7.2 Application Suite-Specific OF Patterns

The types of benchmarks included in SPEC JVM98, JGF S2 and S3 suites differ greatly. While SPEC JVM98 suite consists mostly of integer programs, JGF S2 and S3 applications are all numeric.

Tables 3 and 4 show the chosen SOs for SPEC JVM98 and JGF S2 suites respectively, sorted by their dynamic frequencies. A total of 8 SPEC JVM98 SOs from Table 3 (SOs with i equal to 1, 3, 9, 10, 15, 17, 19 and 20) match with the selected SOs from Table 2, which was built with SOs across all benchmark suites. For JGF S2 benchmarks, 7 of the SOs in Table 4 (SOs with i equal to 2, 6, 10, 12, 13, 15 and 16) match with the chosen SOs from Table 2. A total of 5 JGF S2 SOs (SOs with i equal to 2, 5, 6, 11 and 13) from Table 4 match with the SOs formed specifically for SPEC JVM98 benchmarks. For SPEC JVM98 suite, the highest dynamic frequencies (greater than 1%) are among the top 5 SOs, while for JGF S2 suite, the highest dynamic frequencies are among the top 8 SOs.

In conclusion, for SPEC JVM98 suite, 40% of its SOs with dynamic frequency higher than 1% are included in Table 2. The SOs listed in the latter table were chosen across all benchmark suites. For JGF S2 suite, only 25% of its SOs with dynamic frequency higher than 1% are listed in that same table. These numbers indicate that the selection of SOs

SO i	OF Patterns sorted by $dynamicF_i$	$dynamicF_i$
1	aload_0 getfield	27.90200741
2	iload iload if_icmplt	2.760971763
3	iload iconst_1 iadd	2.339918247
4	aload_0 dup	1.448157631
5	aload_3 iload iaload, aload_3 iload aaload *	1.397158820, 0.182245572
6	iload iconst_1 isub	1.157216182
7	iload istore	0.373250193
	fload fstore *, aload astore *	0.055836714, 0.001813772
8	iconst_0 istore	0.332743505
	aconst_null astore *, fconst_0 fstore *	0.034397696, 0.023069119
9	iload iload_2 if_icmplt	0.307536720
10	iload bipush if_icmplt	0.279811193
11	iload ifne, aload ifnonnull *	0.272936085, 0.169210105
12	aload_0 fload_1 putfield	0.208235797
	aload_0 aload_1 putfield *, aload_0 iload_1 putfield *	0.191103944, 0.169708139
13	iconst_0 istore_3, aconst_null astore_3 *	0.186566640, 0.029063834
14	iload ireturn, aload areturn *	0.178853214, 0.051126559
15	iconst_0 istore_2, aconst_null astore_2 *	0.174500460, 0.000639742
16	iload_3 bipush if_icmplt	0.165644754
17	aload_1 arraylength	0.155113970
18	iload iload_3 if_icmplt	0.142285042
19	aload_0 iconst_0 putfield	0.091160099
	aload_0 fconst_0 putfield *, aload_0 aconst_null putfield *	0.074135478, 0.033382744
20	aload_1 iload aaload	0.089603873
	aload_1 iload iaload *, aload_1 iload faload *	0.045459931, 0.024474994
21	aload_0 iconst_1 putfield	0.056663363

Superscript * indicates templatized patterns

Table 2: Top OF patterns across all benchmark suites sorted by dynamic frequency.

from Table 2 is more suited for the efficient execution of SPEC JVM98 suite than JGF S2 kernel applications.

7.3 Application-Specific OF Patterns

We chose two JGF S2 benchmarks, `SOR` and `Series`, to investigate SOs formed by OF patterns that are application-specific. Table 5 lists the selected SOs for `SOR`, while Table 6 prints the top 21 most executed SOs in `Series`. There is no match among the patterns in Tables 5 and 6. Execution is dominated by the top 4 SOs in `SOR` and the top 5 SOs in `Series`.

A total of 7 SOs in `SOR`, which include the 3 SOs with highest dynamic frequencies, match with JGF S2 SOs listed in Table 4. SOs in `Series` match with 4 of the SOs selected for the whole JGF S2 suite, listed in Table 4. The matches include 2 of the SOs most executed in `Series`.

Summarizing, 75% of the most executed SOs in `SOR` and 40% of the most executed SOs in `Series` are listed in Table 4, which was built with SOs across all benchmarks in JGF S2 suite. A comparison with the SOs selected across all benchmark suites, listed in Table 2, reveals 7 matches with `SOR`-specific SOs and 3 matches with `Series`-specific SOs. This result indicates that the selection of SOs in Tables 2 and 4 are more suited for the efficient execution of `SOR` than that of `Series`.

SO i	JVM98 OF Patterns sorted by $dynamicF_i$	$dynamicF_i$
1	aload_0 getfield	41.76075343
2	aload getfield	2.442815981
3	aload_0 dup	2.344630619
4	aload_1 getfield	1.398526947
5	aload_2 getfield	1.110679955
6	iconst_0 ireturn, aconst_null areturn *	0.884937435, 0.157574936
7	iconst_1 ireturn	0.797745458
8	iload_1 ireturn, aload_1 areturn *	0.589914774, 0.248066929
9	iconst_0 istore	0.533820683
	aconst_null astore *, fconst_0 fstore *	0.066653803, 0.045283826
10	iload bipush if_icmplt	0.522853458
11	iload iload if_icmplt	0.516936826
12	aload_3 getfield	0.493484034
13	dup istore	0.394418272
	dup astore *, dup fstore *	0.010087341, 0.036620439
14	dup istore_2, dup astore_2 *	0.381705418, 0.005161115
15	aload_0 fload_1 putfield	0.408759158
	aload_0 aload_1 putfield *, aload_0 iload_1 putfield *	0.373800682, 0.331903814
16	iconst_0 istore_3, aconst_null astore_3 *	0.366151564, 0.057051231
17	iload iconst_1 iadd	0.361902973
18	aload_2 iload_3 aaload	0.360841184
19	iload ireturn, aload areturn *	0.336541321, 0.096930398
20	iconst_0 istore_2	0.333840232
21	aload ifnonnull	0.332153169
	iload ifne *	0.113867424

Superscript * indicates templatized patterns

Table 3: Top OF patterns across all SPEC JVM98 benchmarks sorted by dynamic frequency.

SO i	JGF S2 OF Patterns sorted by $dynamicF_i$	$dynamicF_i$
1	iload iload iadd	8.780847162
2	aload_0 getfield	7.655223119
3	aload iload daload	7.128188147
4	iload iload_1 if_icmplt	6.838776512
5	iload iload if_icmplt	5.987345440
6	iload iconst_1 iadd	5.056918127
7	dadd dstore	3.435724084
8	iload ifgt	2.909574139
9	iadd istore	0.644465695
10	iload istore	0.517273260
11	iload ifne	0.517127932
12	iload_1 ifeq	0.316979018
13	iconst_0 istore	0.131113291
14	dload dstore	0.114162137
15	iload iload_3 if_icmplt	0.087056854
16	aload_1 arraylength	0.064641093
17	iload_3 iload_0 if_icmplt	0.060543507
18	iload i2d	0.041630205
19	iload dload dastore	0.037658401
20	iload iconst_1 if_icmpeq	0.012127224
21	iload iconst_1 if_icmpne	0.009226863

Superscript * indicates templatized patterns

Table 4: Top OF patterns across all JGF S2 benchmarks sorted by dynamic frequency.

SO i	SOR OF Patterns sorted by $dynamicF_i$	$dynamicF_i$
1	aload iload daload	49.6323632
2	iload iload if_icmplt	16.7337441
3	iload iconst_1 iadd	16.5560875
4	iload iconst_1 isub	16.5560875
5	aload_2 iload aaload	0.17740773
6	iload iload_1 if_icmplt	0.16616004
7	aload_3 iload aaload	0.16592024
8	iconst_1 istore	0.01209565
9	iconst_0 istore	0.00012004
10	iload iload_3 if_icmplt	9.47E-06
11	aload_0 getfield	1.5E-06
12	aload_0 dup	2.81E-07
13	aload_0 iconst_0 putfield	1.87E-07
	aload_0 dconst_0 putfield *, aload_0 lconst_0 putfield *	1.87E-07, 9.37E-08
14	iload iconst_1 isub istore	1.87E-07
15	aload_0 aload_2 putfield	9.37E-08
16	aload_0 iconst_1 putfield	9.37E-08
17	aload_0 iload_1 putfield, aload_0 aload_1 putfield *	9.37E-08, 9.37E-08
18	ldc astore_2	9.37E-08
19	ldc astore_3	9.37E-08
20	aload_2 arraylength	9.37E-08
21	iconst_2 ldc2_w dastore	9.37E-08

Superscript * indicates templatized patterns

Table 5: Top OF patterns in SOR benchmark sorted by dynamic frequency.

8 Conclusions

This piece of work thoroughly studies patterns of Java bytecode operations. Using several benchmarks we were able to show that the top 20 SOs formed by basic blocks are enough to cover more than 50% of the total bytecodes executed. Therefore optimizations that target only the top SOs can substantially improve the interpreted execution performance.

We have shown the high efficiency of an interpreter customized with the top SOs formed by basic blocks which are translated optimizing both stack operations and the number of interpreter trips. Such an interpreter yields speedups comparable to and up to fourfold higher (observed in a particular benchmark) than those of earlier techniques.

Existing Java bytecode interpreters can be further optimized by statically adding to the interpreter a limited number of SOs formed by OF patterns that are valuable across a wide range of applications. Our results showed that it is possible to find such patterns and that they produce average performance improvement of 7%. Much higher speedup of up to 39% was measured when customizing SOs at the level of individual applications. This last result motivated the design of a dynamically customized interpreter. However, the type of the SOs we implemented and the overhead associated with the new scheme led to average performance improvement only slightly higher than that of the statically customized interpreter. Throughout the experiments we suggested that the most valuable SOs be identified ahead of time via profiling and information on SOs be conveyed to the runtime system via annotations.

Finally, we make the case for an interpretation technique that is SO annotation-aware, dynamically customizable and that spends effort in generating optimized code for only the

SO i	Series OF Patterns sorted by $dynamicF_i$	$dynamicF_i$
1	dload_1 dconst_1 dadd	19.97809141
2	dload_3 dload_1 dmul	19.97772179
3	iload ifgt	19.95811331
4	dadd dstore	19.93813522
5	dload dload dadd dstore	19.93813522
6	ddiv dstore	0.039956183
7	aload_0 getfield	0.029975453
8	iload iconst_1 if_icmpeq	0.019978091
9	iload i2d	0.019978091
10	dmul dstore	0.019978091
11	dload dreturn	0.019978091
12	dload_1 dstore	0.019978091
13	dload_3 dload_1 dsub	0.019978091
14	iload_3 i2d	0.019977722
15	iload_3 iconst_2 if_icmplt	4.44E-06
16	aload_1 iload_2 aaload	2.96E-06
17	dload ldc2_w dcmpl	2.96E-06
18	iload_2 iconst_4 if_icmplt	1.85E-06
19	iconst_0 istore_3	1.48E-06
20	iconst_0 ldc2_w dastore	1.48E-06
21	aload_0 dup	1.11E-06

Superscript * indicates templatized patterns

Table 6: Top OF patterns in Series benchmark sorted by dynamic frequency.

top most valuable program basic blocks. Implementation of such engine is our future work.

References

[1] Ana Azevedo. *Annotation-aware Dynamic Compilation and Interpretation.* PhD thesis, University of California, Irvine, 2002.

[2] James R. Bell. Threaded Code. *Communications of the ACM (CACM)*, 16(6):370–372, June 1973.

[3] M. G. Burke, J. Choi, S. Fink, D. Grove, and M. Hind. The Jalapeno Dynamic Optimizing Compiler for Java. *ACM Java Grande Conference*, pages 129–141, June 1999.

[4] L. C. Chang, L. R. Ton, M. F. Kao, and C. P. Chung. Stack operations folding in Java processors. *IEEE Computers and Digital Techniques*, 145(5):333–340, September 1998.

[5] Watheq EL-Kharashi, Fayez ElGuibaly, and Kin F. Li. An operand extraction-based stack folding algorithm for java processors. *2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, September 2000.

[6] EPCC. Java Grande Forum Benchmark Suite. See http://www.epcc.ed.ac.uk/javagrande/javag.html.

[7] M. Anton Ertl. A Portable Forth Engine. *The European Forth Conference*, 1993.

[8] M. Anton Ertl. Threaded Code Variations. *The European Forth Conference*, pages 49–55, 2001.

[9] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *Software - Practice and Experience*, 32(3):265–294, 2002.

[10] David A. Greene, Charles R. Lefurgy, and Trevor N. Mudge. Compiler-Directed Instruction Stream Compression. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan, May 1998.

[11] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an Efficient Java Interpreter. *Lecture Notes in Computer Science*, 2110:613–620, 2001.

[12] D. Griswold. The Java HotSpot Virtual Machine Architecture, March 1998. See whitepaper at http://www.javasoft.com/products/hotspot.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[14] Soo-Mook Moon and Kemal Ebcioglu. LaTTe: An Open-Source Java Virtual Machine and Just-In-Time Compiler. See http://latte.snu.ac.kr.

[15] Ian Piumarta and Fabio Riccardi. Optimizing Direct-threaded Code by Selective Inlining. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[16] Todd A. Proebsting. Optimizing an ANSI C Interpreter with SuperOperators. *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332, 1995.

[17] Austin Kim Yang Qian and Morris Chang. Designing a Memory System Using a Static Loader for Embedded Java Architectures. *2nd International Workshop on Compiler and Architecture Support for Embedded Systems*, pages 565–566, October 1999.

[18] Ramesh Radhakrishnan, Juan Rubio, and Lizy K. John. Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels. *International Conference on Computer Design*, pages 281–284, October 1999.

[19] SPEC Standard Performance Evaluation Corporation (SPEC). SPEC JVM98 Benchmarks, 1998. See http://www.specbench.org/osg/jvm98/.

[20] Sun Microsystems Inc. Picojava I microprocessor core architecture, March 1999. See http://solutions.sun.com/embedded/databook/pdf/whitepapers/WPR-0014-01.pdf.

[21] Sun Microsystems Inc. Picojava-II programmer's reference manual, March 1999.

[22] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Mul ler. Static and Dynamic Program Compilation by Interpreter Specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[23] L. R. Ton, L. C. Chang, M. F. Kao, H. Tsenga, S. Shang, R. Ma, D. Wang, and C. P. Chung. Instruction Folding in Java Processor. *International Conference on Parallel and Distributed Systems*, pages 138–143, December 1997.