High Performance Annotation-aware JVM for Java Cards

Ana Azevedo Arun Kejariwal Alex Veidenbaum Alexandru Nicolau

Center for Embedded Computer Systems
School of Information and Computer Science
University of California at Irvine
apazos@qualcomm.com, arun_kejariwal@computer.org, {alexv,nicolau}@cecs.uci.edu

ABSTRACT

Early applications of smart cards have focused in the area of personal security. Recently, there has been an increasing demand for networked, multi-application cards. In this new scenario, enhanced application-specific on-card Java applets and complex cryptographic services are executed through the smart card Java Virtual Machine (JVM). In order to support such computation-intensive applications, contemporary smart cards are designed with built-in microprocessors and memory. As smart cards are highly area-constrained environments with memory, CPU and peripherals competing for a very small die space, the VM execution engine of choice is often a small, slow interpreter. In addition, support for multiple applications and cryptographic services demands high performance VM execution engine. The above necessitates the optimization of the JVM for Java Cards.

In this paper we present the concept of an annotation-aware interpreter that optimizes the interpreted execution of Java code using Java bytecode SuperOperators (SOs). SOs are groups of bytecode operations that are executed as a specialized VM instruction. Simultaneous translation of all the bytecode operations in an SO reduces the bytecode dispatch cost and the number of stack accesses (data transfer to/from the Java operand stack) and stack pointer updates. Furthermore, SOs help improve native code quality without hindering class file portability. Annotation attributes in the class files mark the occurrences of valuable SOs, thereby dispensing the expensive task of searching and selecting SOs at runtime. Besides, our annotation-based approach incurs minimal memory overhead as opposed to just-in-time (JIT) compilers.

We obtain an average speedup of 18% using an interpreter customized with the top SOs formed from operation folding patterns. Further, we show that greater speedups could be achieved by statically adding to the interpreter application-specific SOs formed by top basic blocks. The effectiveness of our approach is evidenced by performance improvements of (upto) 131% obtained using SOs formed from optimized basic blocks.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*optimization; annotations*

General Terms

Performance, Algorithms

Keywords

Java Card, Virtual Machine, Superoperators, High Performance

1. INTRODUCTION

Smart cards have become ubiquitous platforms for personalized services [1]. For example, a smart card is used as a phone card, a health insurance card, identity cards, in GSM phones or an electronic wallet [2, 3]. A typical smart card consists of a microprocessor, ROM (Read Only Memory), EEPROM (Electrically Erasable Programmable Read Only Memory) and RAM (Random Access memory). The ROM contains the card operating system and other applications.

Although a smart card has benefits over magnetic-strip cards, it has certain limitations such as non-portability of applications, lack of flexibility to download applications into a smart card. Recently, Java Card has been proposed as a solution to these limitations. Java Card is based on a subset of the Java API plus some special-purpose card commands [4, 5]. The Java Card technology enables portability of applications, and provides a secure platform to store and execute multiple applications, called *applets*,[1] on a smart card [6]. Applets are small enough so that several of them can fit into a single card with small amount of memory. Further, it facilitates installation of applications after the card has been issued which provides card issuers with the ability to dynamically respond to their customer's changing needs. For example, if a customer decides to change the frequent flyer program associated with the card, the card issuer can make this change, without having to issue a new card. Java provides a security model that lets applets from multiple sources reside safely on the same card. This is important because an applet on a smart card may attempt to access data intended to be private only to some other applet. The Java Card platform can be implemented in either of the following two ways:

a) It can be implemented as an external Java Card tool. The tool loads, verifies and prepares Java classes for on-card execution.

[1]Applets are small code objects that are designed to be downloaded onto a client machine from a remote host.

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 1: Traces of the translations of an `ILOAD` followed by an `ISTORE` operation

b) It can be implemented as an on-card Java Card VM based on Java bytecode Interpreter.

With the development of Java Cards, new application languages are being designed and put into use, for example, Java Card 2.x [7, 8]. The above facilitates code reuse with the potability across different chip architectures. In addition, the object-oriented approach of Java Card 2.x provides flexibility in programming smart cards.

The widespread use of networked devices has rendered the program and application data on such cards susceptible to external attacks. Thus it has increased the need for securing the data against such attacks[9, 10]. Securing the digital data mandates personal identification and non-repudiation. For the same, most cards have builtin support for cryptography. The need for higher security levels has led to the use of complex crypto keys (usually DES or RSA private keys). However, use of complex, computation-intensive security algorithms adversely affects the performance of the smart card due to its limited computation resources.

Most of the research in the context of the JVM has concentrated in high-end optimizations such as advanced garbage collection techniques, just-in-time (JIT) compilation, hotspot analysis and adaptive compilation techniques. However, Java programs running on low-end smart cards cannot afford the overhead (such as increase in class file size and additional compilation time) associated with such optimizations. The latter stems from the limited on-card memory which restricts the size of the VM code itself. In addition, support for multiple applications and cryptographic services demands high performance VM execution engine. The above necessitates the optimization of the JVM for Java cards.

In this paper we propose new methods for optimizing the performance of Java interpreters for Java cards with patterns of Java bytecode operations or SuperOperators.[2] For example, consider the two traces shown in Figure 1 which illustrate how combining bytecode operations into patterns can lead to optimized interpreted execution. The upper side of Figure 1 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 portion of Figure 1 shows the situation in 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 (instructions 5 and 9 in the lower trace). The combined translation of the two bytecodes executes 12 instructions, a 40% improvement over the separate translation.

[2]Techniques based on SOs are orthogonal to implementing threading and should further boost the efficiency of a threaded code interpreter (shown in lower portion of Figure 2).

The rest of the paper is organized as follows: In Section 2 we present a brief overview of program interpretation schemes. Next, in Section 3 we discuss our approach for generation of SOs. Static and dynamic customization of the interpreter is discussed in Sections 4 and 5 respectively. The SO annotation format is discussed in Section 6. The experimental framework and results are presented in Section 7. In Section 8 we discuss previous work. Finally, we conclude with directions for future research in Section 9.

2. BACKGROUND

Program interpretation is the process of emulating in software the basic tasks of fetching, decoding and executing instructions of a normal program execution, which are usually done by a microprocessor hardware. Therefore interpretation has 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 part of Figure 2, 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 3, which shows part of LaTTe's interpreter engine directly written in SPARC assembly code. LaTTe is an open-source, high performance JVM interpreter implementation [11]. We use LaTTe for our experiments as it is non-proprietary and for ease of reproducibility of results. However, the ideas presented in this paper are applicable and beneficial for optimizing any JVM.

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, the top of the stack (`TOP`) and the logical program counter (`PC`), are kept in SPARC machine registers for improved performance. Figure 3 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.

```
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 2: Bytecoded interpreter and threaded code interpreter schemes

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 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 [12]). A dispatch optimization technique has been proposed in [13].

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 (like threaded code interpreter), 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. In this paper we focus on minimizing the stack machine overhead.

JIT compilers [11, 14, 15] eliminate the above issues altogether at the cost of more space to store the compiled methods and the compilation framework itself and associated runtime cost. A JIT compiler is not a viable solution in domains where space constraints limit the available memory such as smart card.

3. GENERATION OF SOS

Operation folding (OF) has been proposed as a mechanism to enhance the PicoJava's performance by turning many cycles of stack oriented instructions into a one-cycle register-based instruction, which can be implemented with a few registers [16]. OF groups or folds contiguous operations that have true data dependency [17]. For example, the following bytecode sequence can be transformed using operation folding in the Java processor.

```
iload_1 iload_2 iadd istore_3 → add R1, R2, R3
```

In PicoJava instructions are combined into folding patterns ac-

```
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 3: Bytecoded interpreter loop example

cording to a definite set of language-based grouping rules. The algorithm for folding instructions is very simple. It scans the bytecode stream looking for sequences of instructions that match the patterns and folds instructions corresponding to the longest pattern amongst the choices. Instruction folding in PicoJava is implemented directly in hardware. It is limited by the maximum number of instructions it can fold at a time and in the number of grouping rules, which together define the size and complexity of the instruction decoder. Also, it only folds bytecode operations for which there is a microcode instruction that executes the bytecode semantics.

In order to alleviate the above limitation, we proposed to generalize the PicoJava instruction folding mechanism in software [18]. As a natural consequence, we can process more complex bytecodes and longer sequences as long as we have enough registers to hold the values from the stack accesses which are eliminated during the folding process. For example, our technique can handle both long and double values in local variables or in the constant pool. The proposed technique can be easily extended to include method invocation bytecodes to facilitate exploitation of registers for parameter passing when folding instructions.

One way to form SOs is from stack operation folding (OF) patterns. The patterns are groups of Java bytecode operations that repeat across methods pertaining to a single program or whole benchmark suites. Language-based grouping rules as in PicoJava I and PicoJava II can be employed to find patterns. Another way to find patterns is to search for them in the compiler's internal expression trees. In [19], Kim proposed a Java bytecode optimization wherein *non-contiguous* operations that may be far apart on the stack can be folded together. El-Kharashi proposed an *operand extraction-based* stack folding technique for finding *nested* folded patterns [20].

A dictionary-based search can be employed for searching the longest pattern [18]. The OF patterns are prioritized by different heuristics such as length of the pattern, static/dynamic frequency of occurrence and are selected for forming SOs according to the level of customization of the interpreter, as discussed below:

i) **Application-specific OF Patterns**: The SOs are formed by OF patterns which are application-specific.

ii) **Benchmark Suite-specific OF Patterns**: The SOs are formed by OF patterns common to all the applications within a benchmark suite.

iii) **OF Patterns Across Benchmark Suites**: The stack operation folding technique is applied across all applications of the different application suites to obtain SOs that are valuable across benchmark suites.

Another 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. For further details, the reader is referred to [18]. In this study, we do not consider 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.

Although an interpreter optimized with SOs formed with OF patterns can improve the performance of a wider range of applications, the speedups it will produce will probably be lower compared to techniques that deploy sets of SOs formed by basic blocks. Also, it should be noted that the basic block approach for forming SOs is applicable on a per-application basis as basic blocks, unlike OF patterns, do not repeat across applications or benchmark suites.

The BB and OF candidate patterns for folding are prioritized either statically or dynamically, as discussed in Sections 4 and 5 respectively. A higher priority value of a SO is indicative of larger performance gain achievable by folding the corresponding pattern. We are interested in patterns that repeat across all applications of our Java Card benchmark suite. From these patterns we select the topmost ones (most frequently occurring) to form SOs.

4. STATIC CUSTOMIZATION

When statically customizing the interpreter, the original interpreter loop is extended with new bytecodes that define the SOs. The number of new bytecodes is limited by the number of free opcodes available, i.e., the number of opcodes the interpreter does not use. Figure 4 illustrates an example of LaTTe's interpreter loop with a SO defined as a new bytecode in the interpreter engine, with opcode 203.

A single optimal set of common patterns is statically chosen from the set of all possible OF patterns. One of the ways to prioritize the candidate SOs is based on their average static frequency (SF). SF of a SO i is defined as follows:

```
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
…
.macro DECLARE opcode
\(\(.org)) _interpret_start + \opcode * DISP
.endm
…
! void interpret (Method *m, void *args, void* bcode)
interpret:
            …
            ! Initialize registers, e.g., %i1
            sethi %hi(_interpreter_start), ORIGIN
            or ORIGIN, %lo(_interpret_start), ORIGIN
            …
_interpreter_start:

DECLARE 0       !NOP
…
DECLARE 1       ! ACONST_NULL
…
DECLARE 201      ! JSR_W
…
DECLARE 203: !  SO iload_3 iload   iadd
! Read next bytecode to be executed
 ldub [ PC + 4], TARGET
! Execute the SO semantics
ld [%i4 -12], %l3
ldub [%i2 +2], %o2
sll %o2, 2, %o2
neg %o2
ld [%i4 + %o2], %l4
add %l3, %l4, %o0
st %o0, [%i3 -4]
add TOP,  -4, TOP
!Transfer control to the next  bytecode or SO
sll TARGET, SDISP, TARGET
jmp ORIGIN + TARGET
add %i2,  4, %i2
…
_interpreter_end:
…
```

Figure 4: Statically customizing an interpreter loop

$$SF_i = \frac{\sum_1^n \frac{100 \times SF_{ik}}{\sum_1^t SF_{ik}}}{n} \quad (1)$$

where SF_{ik} is the number of times SO i repeats in the static code of program k, t is the total number of SOs in the static code of program k and n is the number of programs averaged over ($n = 20$). In [18], Azevedo showed that it is better to prioritize the candidate SOs based on their dynamic frequencies instead of their static frequencies. Therefore, we use dynamic frequency as the basis for SO selection. Each SO is "marked" in a class file with an annotation. The interpreter uses the annotations at run-time to switch to execute an SO.

The advantage of the static approach is that executing a SO has the same dispatch cost of a single bytecode execution. As a SO comprises of more than one bytecode operation, the overall dispatch cost is reduced. When generating machine code for each SO we also optimize stack accesses and stack pointer updates, placing values in registers whenever possible.

5. DYNAMIC CUSTOMIZATION

One of the ways to achieve best speedup is to optimize an interpreter with SOs customized for a particular application. However, the above has very limited applicability. A better solution would be to adapt the interpreter code based on the application under execution. A JIT compiler achieves this by compiling functions at the method level and recompiling, in case of re-optimizing compilers. However, JIT leads to large JVM code which makes the above approach unsuitable for Java cards.

```
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)
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
sub PC, BCBASE, %l3
sll %l3, 2, %l4
add SONCODE, %l4, %l4
ld [%l4], %l4
jmp %l4
nop

! displacement from bytecode begin %l3
! calculating index into native code table %l4
! address of the translated code %l4
! jump to the SO translated code
                …
_interpreter_end:
                …
```

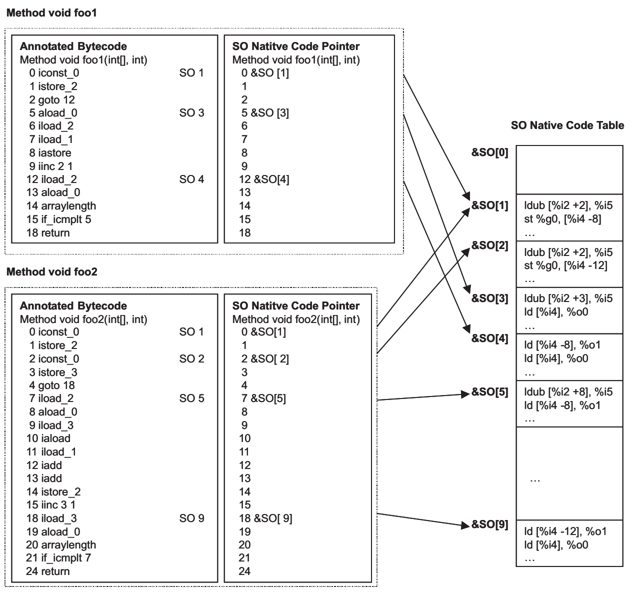
Figure 5: Dynamically customizing an interpreter loop

In this section we propose a dynamic scheme that is more flexible in customizing the interpreter for a particular applet or service application. In this new approach, we propose to add SOs to the interpreter "on-the-fly". For efficient selection of the candidate SOs, the same are prioritized dynamically based on their average dynamic frequency (DF). DF of a SO i is defined as follows:

$$DF_i = \frac{\sum_1^n \frac{100 \times DF_{ik}}{\sum_1^t DF_{ik}}}{n} \quad (2)$$

where DF_{ik} is the number of times SO i executed during the execution of program k.

The interpreter loop is essentially the same as in LaTTe's original interpreter code (except for some extra operations to control the dynamic scheme). However, at runtime when a SO is identified (via annotations) 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 the native code table, indexed by the SO identification number. This table is shared by all the methods in the program. Each bytecode in a method that corresponds to a SO start is annotated and 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. Figure 5 (on page) shows how the interpreter loop of a dynamically customized interpreter invokes a SO. The data structures used in the dynamic case are shown in Figure 6.



```
Annotated Bytecode
Method void foo1(int[], int)
  0 iconst_0            SO 1
  1 istore_2
  2 goto 12
  5 aload_0             SO 3
  6 iload_2
  7 iload_1
  8 iastore
  9 iinc 2 1
 12 iload_2             SO 4
 13 aload_0
 14 arraylength
 15 if_icmplt 5
 18 return
```

```
SO Natitve Code Pointer
Method void foo1(int[], int)
  0 &SO [1]
  1
  2
  5 &SO [3]
  6
  7
  8
  9
 12 &SO[4]
 13
 14
 15
 18
```

```
Annotated Bytecode
Method void foo2(int[], int)
  0 iconst_0            SO 1
  1 istore_2
  2 iconst_0            SO 2
  3 istore_3
  4 goto 18
  7 iload_2             SO 5
  8 aload_0
  9 iload_3
 10 iaload
 11 iload_1
 12 iadd
 13 iadd
 14 istore_2
 15 iinc 3 1
 18 iload_3             SO 9
 19 aload_0
 20 arraylength
 21 if_icmplt 7
 24 return
```

```
SO Natitve Code Pointer
Method void foo2(int[], int)
  0 &SO[1]
  1
  2 &SO[ 2]
  3
  4
  7 &SO[5]
  8
  9
 10
 11
 12
 13
 14
 15
 18 &SO[ 9]
 19
 20
 21
 24
```

```
&SO[0]
&SO[1]  ldub [%i2 +2], %i5
        st %g0, [%i4 -8]
        …
&SO[2]  ldub [%i2 +2], %i5
        st %g0, [%i4 -12]
        …
&SO[3]  ldub [%i2 +3], %i5
        ld [%i4], %o0
        …
&SO[4]  ld [%i4 -8], %o1
        ld [%i4], %o0
        …
&SO[5]  ldub [%i2 +8], %i5
        ld [%i4 -8], %o1
        …
          …
&SO[9]  ld [%i4 -12], %o1
        ld [%i4], %o0
        …
```

Figure 6: Data structures used in a dynamically customized interpreter

The interpreter size in the dynamic approach differs from the static approach in the space required by the SO translation functions for each pattern class, the array of pointers to the shared native code table created for each method and the shared native code table. The latter can grow as large as the number of SOs executed. Besides the SO annotation overhead, dynamic customization also incurs the overhead of invoking the translated SOs.

6. SO ANNOTATION FORMAT

Approaches such as JIT compilation rely on bytecode analysis (of varying sophistication) to extract information about the program, which is then used to optimize the native code during the translation process. However, extracting information from a low-level representation such as the bytecodes is very expensive. In order to alleviate the same, Hummel et al. proposed an approach based on annotations, wherein the bytecodes are annotated during the source code to bytecode translation, thus facilitating aggressive optimization by an annotation-aware bytecode compiler [21, 18]. We employ the same (annotation-based) approach for optimizing the performance of Java interpreters with SOs.

SO annotations identify where SOs occur in the bytecode stream. Thus, annotations help eliminate the expensive pattern search and the selection phases at runtime. SO annotations are encoded in Java

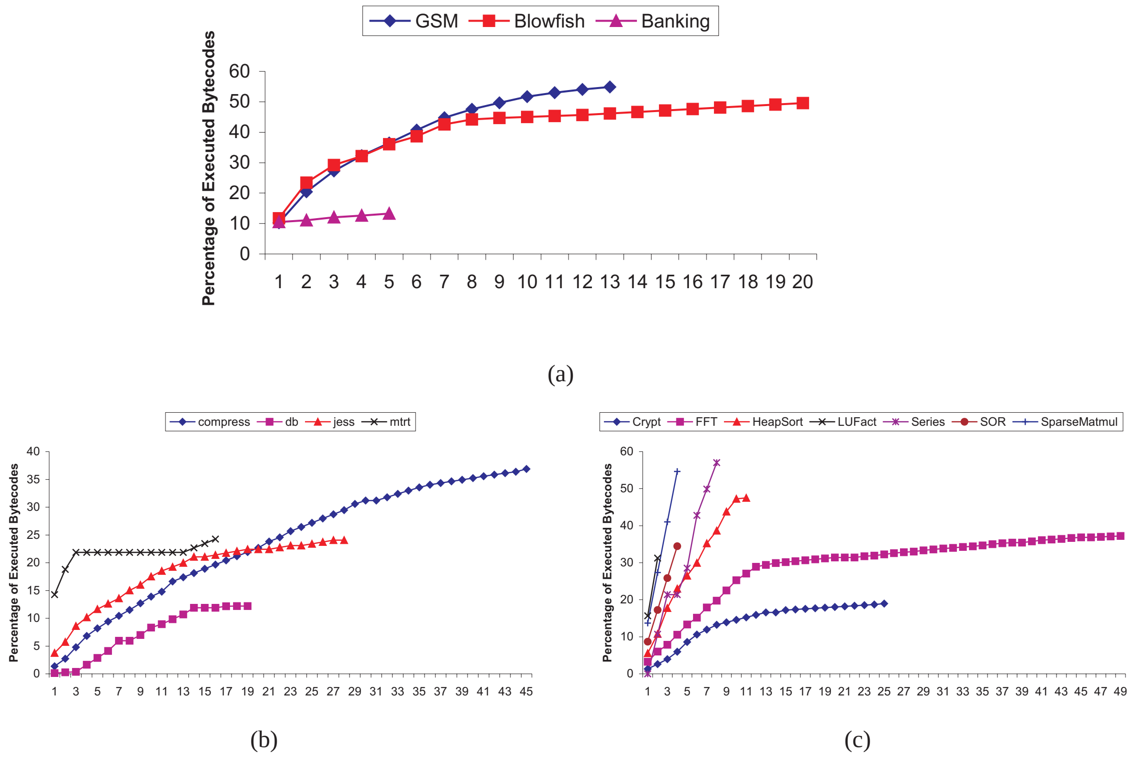


Figure 7: SuperOperators formed by OF Pattern with DF $> 1\%$

```
SOAnnotationTableAttribute{
  u2 name;
  u4 length;
  u2 tableLength;
  {
    u2 pc;
    u1 id;
    u1 type;  // used in the dynamic case only
  }SOAnnotationTable[tableLength];
}
```

Figure 8: `SOAnnotationTable` **attribute format**

class files as a new user-defined attribute to the Code attribute [22] of a Java method, referred to as `SOAnnotationTable`, as shown in Figure 8 (on page). Field `name` is a valid index into the constant pool table representing the string "`SOAnnotationTable`"; `length` indicates the total size of the attribute; and `tableLength` indicates the number of entries in the `SOAnnotationTable` array. Each entry in the `SOAnnotationTable` array contains an index `pc` into the code array of the corresponding Code attribute; a SO `id` identification number; and the `type` of the SO.

7. EXPERIMENTAL EVALUATION

To evaluate the proposed techniques we use three Java card applications, viz., Blowfish, GSM and Banking, and applications from the SPEC JVM98, JGF S2 benchmark suites. In Section 7.1, we present a quantitative measure of the percentage of total bytecode execution corresponding to the SOs generated using our annotation-based approach. Speedup results of the static and dynamic schemes discussed earlier are presented in Sections 7.2 and 7.3 respectively. Results of the code size overhead associated with both the static and dynamic schemes are discussed in Section 7.4. Lastly, we compare the performance of the different SO-based techniques.

7.1 SO Selection

Figure 7 shows the percentage of the total bytecode execution of the top SOs formed by OF patterns with dynamic frequency $> 1\%$ for different benchmarks. The top SOs for Java Card applications are shown in Figure 7(a). Observe that the SOs of GSM and Blowfish account for more than 50% of the total bytecode execution. In order to illustrate the generic nature of our approach, the percentage of the total bytecode execution of the top SOs for SPEC JVM98 and JGF S2 benchmark suites is shown in Figures 7(b) and 7(c) respectively. Note that the SOs of LUFact and SparseMatMul constitute over 50% of the total bytecode execution. For complete results on SpecJVM98, Java Grande Forum S2 see [18]. From hereon, we only present the results for the applications of the Java Card benchmark suite.

7.2 Statically Customized Interpreter

In order to estimate an upper bound of the performance of SO-based interpreters, we statically customized the interpreter, forming the SOs statically on a per-application basis. We chose to implement the top 21 SOs formed from OF patterns, for this is the number of free opcodes available in LaTTe's interpreter. Figure 9 shows the corresponding application-specific speedup for the different applications of the Java Card suite. Speedups of 28% and 23% were obtained for Blowfish and GSM respectively. The low speedup for Banking can be attributed to the fact that the OF patterns contribute only 10% of the total bytecodes executed.

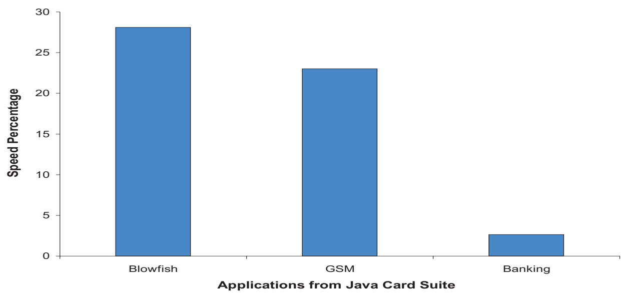


Figure 9: Results of application-specific static customization

7.3 Dynamically Customized Interpreter

The results in Figure 10 correspond to dynamically customizing interpreters with SOs formed by OF patterns that are application-specific. We obtain an average speedup 15% for Blowfish, 2.1% for GSM and 2.6% for the Banking application. We note that the OF patterns are too small to overcome the overhead associated with dynamic customization. In order to mitigate the affect of the overhead, we experiment with SOs formed by top BBs in Section 7.5. Greater speedups (compared to static customization) can be achieved by considering larger number of BBs.

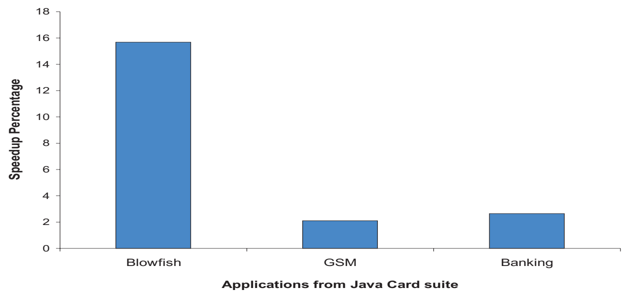


Figure 10: Results of Dynamic Customization

7.4 Code Size Overhead

Static and dynamic customization of the interpreter leads to increase in (Java) code size. Table 1 shows the % increase in code increase for both static and dynamic schemes, discussed in Sections 4 and 5 respectively. From Table 1 we note that there is a modest increase in code size of 6.6% and 14.7% (on an average) for static and dynamic schemes respectively. The slightly higher code size increase in the dynamic case can be attributed to the added byte to represent each type of SO.

7.5 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

Benchmark	% Code size increase	
	Statically Customized	Dynamically Customized
Blowfish	7	18
GSM	4	15
Banking	9	11

Table 1: % increase in code size

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 (for details refer to [18]).

- **BBOpt: Optimizing top SOs formed by basic blocks**
  In this interpreter version, the most valuable SOs are fully optimized, i.e., the bytecode translations within a BB are concatenated and optimized, eliminating the dispatch and stack model costs.
- **BB: Unoptimized SOs formed by basic blocks**
  The interpreter simply concatenates the translation of the individual bytecodes that compose the most important SOs.
- **OF: Optimizing top SOs formed by Operation Folding patterns**
  The interpreter fully optimizes SOs as in BBOpt, but SOs are formed by operation folding sub-patterns that appear in the most executed basic blocks.

From Figure 11 we observe that the BBOpt scheme yields significantly higher speedup than BB and OF for Blowfish (131.6%) and GSM (84.4%). It can be attributed to the fact that the two benchmarks have large, frequently executed basic blocks. In contrast, there is very insignificant performance gain for the Banking application owing to small, infrequently executed basic blocks. The results shown in Figure 11 correspond to the top 7-21 basic blocks in each application. Optimization of other basic blocks and deploying the corresponding SOs would yield further speedups.

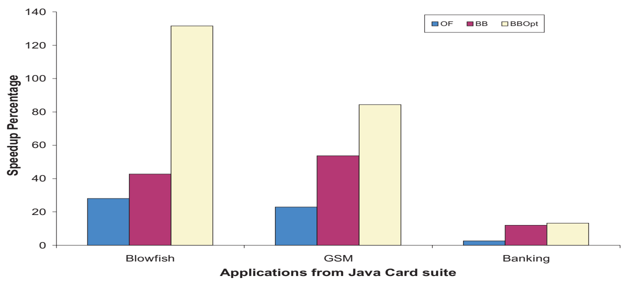


Figure 11: Performance comparison of different SO types

8. RELATED WORK

The breakthrough in the efficient implementation of virtual machine interpreters is the threaded code technique [23, 24, 25, 26]. 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 [27] introduces SOs as specialized instructions automatically inferred from repeated patterns in the tree-like intermediate representation produced by *lcc* compiler [28]. His bytecoded interpreter extended with SOs runs 2 to 3 times faster with the tested benchmarks. Ertl, Gregg et al. [29, 30] 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* [31]. 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. [32] 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 [33] and PicoJava II [34] 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 [35, 36, 20]. Kim [37] 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.

Static program analysis for identifying repetitive sequences of bytecodes for compression has been studied in [38, 39]. Casey et al. [40] proposed a Java instruction, called *superinstruction*, to reduce the number of instruction dispatches. However, their approach for selecting bytecode sequences is restricted to per-program basis. Donoghue and Power proposed an approach to (statically) select a generic set of superinstructions to be used across different programs [41]. In [42], Ertl and Gregg proposed a method to exploit dynamic superinstructions in conjunction with stack caching [25] to minimize the interpreter dispatch overhead; in addition, they propose a shortest path search algorithm to determine an "optimal" sequence of state transitions and VM instructions. In [13], Ogata et al. proposed a prefetching-based technique to minimize redundant memory accesses and the overhead of indirect branches. Gaganon and Hendren proposed a technique called *preparation sequences* employed for inlined-threading of Java interpreters in presence of dynamic class loading, lazy class initialization and multithreading [43].

9. CONCLUSIONS

In this paper we presented an annotation-aware JVM for Java cards that optimizes interpreted execution of Java code using Java bytecode SOs. The most valuable SOs are identified ahead of time via profiling and information about SOs is conveyed to the runtime system via annotations. Annotation attributes in the class files mark the occurrences of valuable SOs, dispensing the expensive task of selecting SOs at runtime, without hindering class file portability. The low run-time overhead coupled with minimal memory requirements makes our annotation-based approach ideal for optimization of JVM for low-end embedded systems.

In order to achieve high performance, the SOs can be formed by OF patterns which cover more than 50% of the total bytecodes executed. Therefore, optimizations that target only the top SOs can substantially improve the interpreted execution performance. An average speedup of 18% is obtained using an interpreter customized with the top SOs formed by OF patterns. Greater speedups can be achieved by statically adding to the interpreter application-specific SOs formed by top BBs. Our results show performance improvement of (upto) 131% obtained using SOs formed from optimized BBs. As future work, we would like to extend our annotation-based approach for compression of Java bytecodes for Java cards.

10. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under Grant No. 0311738.

11. REFERENCES

[1] O. Potonniée. Ubiquitous personalization: A smart card based approach. In *Proceedings of the 4th Gemplus Developer Conference*, November 2002.

[2] Smart card: A primer. `http://www.javaworld.com/javaworld/jw-12-1997/jw-12-javadev.html`.

[3] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley and Sons, 1997.

[4] Java card technology. `http://java.sun.com/products/javacard/`.

[5] A Java card primer. `http://www.developer.com/java/other/article.php/910261`.

[6] U. Hansmann, S. Nicklous, T. Schaeck, A. Schneider, and F. Seliger. *Smart Card Application Development Using Java*. Springer-Verlag, second edition, 1999.

[7] Z. Chen. *Java Card^{TM} Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 1991.

[8] G. Grimaud and J.-J. Vandewalle. Introducing research issues for next generation Java-based smart card platforms. In *Proceedings of the Smart Objects Conference*, pages 138–141, Grenoble, France, May 2003.

[9] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, 1996.

[10] H. Pietiläinen. Elliptic curve cryptography on smart cards. Master's thesis, Helsinki University of Technology, October 2000.

[11] LaTTe. `http://latte.snu.ac.kr/`.

[12] R. Radhakrishnan, J. Rubio, and L. K. John. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. *Proceedings of the 17th International Conference on Computer Design*, pages 281–284, October 1999.

[13] K. Ogata, H. Komatsu, and T. Nakatani. Bytecode fetch optimization for a Java interpreter. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 58–67, San Jose, CA, 2002.

[14] 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.

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

[16] H. McGhan and M. O'Connor. PicoJava: A direct execution engine for java bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[17] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

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

[19] A. Kim and M. Chang. Java bytecode optimization with advanced instruction folding mechanism. pages 268–275, October 2000.

[20] W. El-Kharashi, F. ElGuibaly, and K. F. Li. An operand extraction-based stack folding algorithm for Java processors. *Proceedings of the 2nd Annual Workshop on Hardware Support for Objects and Microarchitectures for Java*, 2000.

[21] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the java bytecodes in support of optimization. *Journal Concurrency:Practice and Experience*, 9(11):1003–1016, November 1997.

[22] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, MA, 1999.

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

[24] M. A. Ertl. A Portable Forth Engine. In *EuroForth '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[25] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[26] M. Anton Ertl. Threaded Code Variations. In *EuroForth '93 conference proceedings*, pages 49–55, 1993.

[27] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, San Francisco, CA, 1995.

[28] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Boston, MA, 1995.

[29] D. Gregg, M. A. Ertl, and A. Krall. Implementing an efficient java interpreter. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 613–620, 2001.

[30] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen — A generator of efficient virtual machine interpreters. *Software - Practice and Experience*, 32(3):265–294, 2002.

[31] I. Piumarta and F. Riccardi. Optimizing Direct-threaded Code by Selective Inlining. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[32] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

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

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

[35] 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.

[36] 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.

[37] A. Kim, Y. Qian, and Morris Chang. Designing a memory system using a static loader for embedded Java architectures. *Proceedings of the 2nd International Workshop on Compiler and Architecture Support for Embedded Systems*, pages 565–566, October 1999.

[38] D. Rayside, E. Mamas, and E. Hons. Compact Java binaries for embedded systems. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 9, Mississauga, Ontario, Canada, 1999.

[39] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[40] K. Casey, D. Gregg, M. A. Ertl, and A. Nisbet. Towards superinstructions for java interpreters. In *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems*, pages 329–343, Vienna, Austria, 2003.

[41] D. O'Donoghue and J. F. Power. Identifying and evaluating a generic set of superinstructions for embedded java programs. In *Proceedings of the International Conference on Embedded Systems and Applications*, pages 192–198, 2004.

[42] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. In *Proceedings of the 2004 workshop on Interpreters, Virtual Machines and Emulators*, pages 7–14, Washington, D.C., 2004.

[43] E. Gagnon and L. J. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 170–184, Warsaw, Poland, 2003.