Identifying and evaluating a generic set of superinstructions for embedded Java programs

Diarmuid O'Donoghue
Department of Computer Science,
National University of Ireland,
Maynooth, Co. Kildare, Ireland.
dod@cs.may.ie

James F. Power
Department of Computer Science,
National University of Ireland,
Maynooth, Co. Kildare, Ireland.
jpower@cs.may.ie

***Abstract*— In this paper we present an approach to the optimisation of interpreted Java programs using superinstructions. Unlike existing techniques, we examine the feasibility of identifying a generic set of superinstructions across a suite of programs, and implementing them statically on a JVM. We formally present the sequence analysis algorithm and we describe the resulting sets of superinstructions for programs from the embedded CaffeineMark benchmark suite. We have implemented the approach on the Jam VM, a lightweight JVM, and we present results showing the level of speedup possible from this approach.**

***Index Terms*— Java Virtual Machine, interpreted code, superinstructions**

I. INTRODUCTION

The Java programming language, and its associated Java Virtual Machine (JVM) has led to a renaissance in the study of stack-based machines. Much of the research dealing with the JVM has concentrated on heavyweight high-end optimisations such as advanced garbage collection techniques, just-in-time compilation, hotspot analysis and adaptive compilation techniques. However, as highlighted in a number of recent studies [1], [2], [3], Java programs running in low-end or embedded systems often cannot afford the overhead associated with these optimisations. Designers of JVMs for such systems must concentrate their efforts on directly improving interpreted Java code.

One optimisation technique is the use of *superinstructions*, where a commonly occurring sequence of instructions is converted into a single instruction, thus saving fetch and/or dispatch operations for the second and subsequent instructions. This technique was originally applied to C [4], [5] and Forth programs [6], but has lately been extended to cover Java programs as well [7], [3]. Both of these published approaches to implementing superinstructions in Java give some details of the technique used and the speedup achieved. However, they do not present any details of the actual superinstructions used, nor do they investigate fully all of the choices involved in their selection, at least for shorter bytecode sequences.

For example, a straightforward dynamic analysis of interpreted Java programs shows that load instructions can account for about 40% of bytecodes executed, with field accesses accounting for between 10% and 20% [8]. Similarly, our studies have shown that the instruction pair `aload_0 getfield` occurs quite frequently in Java programs - averaging to 9% of the instructions executed in one set of benchmark suites [9]. Most approaches to implementing superinstructions specialise the virtual machine for the program under consideration. However, given the clustering in the distribution of bytecodes used, it seems reasonable to ask if it is possible to engineer a generic set of superinstructions usable across different programs. Such an approach would have the advantage of eliminating the run-time profiling overhead, as well as exposing the selected superinstructions to compile-time optimisation. The trade-off, however, is that a generic set of instructions will naturally not produce the same level of speedup as superinstructions that are tailored for a given application, or even for a particular phase in the execution of a given application.

In this paper we examine the possible gains from attempting to select a generic set of superinstructions to be used across different programs. We study the selection strategy for choosing these instructions and we present some possible selections of superinstructions. We examine their implementation on a lightweight JVM, the JAM Virtual Machine, and present results showing the level of speedup possible from this approach.

II. BACKGROUND AND RELATED WORK

The concept of *superoperators* was introduced by Proebsting for C programs [4], noting that superoperators consistently increase the speed of interpreted code by a factor of 2 or 3. Proebsting suggests that a maximum of 20 superoperators to get full benefit from the technique, and notes that the choice of superoperators is likely to vary between applications. Both these themes are investigated further for Java programs below. Piumarta and Riccardi develop this work by presenting a technique for selecting and implementing superoperators for C and Caml programs dynamically, and approach they term *direct threaded code* [5]. They note the drawbacks from attempting to base this on a static analysis, and present results indicating a speedup factor of between 2 and 3.

Ertl et al. present an interpreter-generator that supports *superinstructions* which correspond to sequences of simpler

Contact author: James F. Power <jpower@cs.may.ie>
Phone: +353-1-7083447, Fax:+353-1-7083848.

Accepted for the International Conference on Embedded Systems and Applications (ESA '04)
Las Vegas, Nevada, USA, June 21-24, 2004, pp. 192-198

instructions [10]. Examples are presented using both a Forth and a Java interpreter. Ertl et al. selected superinstructions for Java by profiling the *javac* and *db* programs from the SPEC suite, up to a maximum length of 4 instructions. The results presented show a speedup factor of less than 2, and even a slow-down on some architectures due to cache misses. They report that the most frequent sequence of instructions in their JVM was `iload iload`. However, the bytecode used in their study was significantly rewritten from the original, and thus our analysis below presents a different picture. In further work, Ertl and Gregg have examined the effect of superinstructions on branch (mis)prediction [11].

More recently, Gagnon and Hendren have examined the speedup possible from using dynamically-calculated superinstructions in Java [7]. As well as noting a speedup factor of between 1.20 to 2.41 over a switch-based interpreter for such a technique, the paper also examines some of the issues resulting from lazy class loading, where an instruction such as `getstatic` may have the side-effect of triggering class initialisation. Their approach parallels that of Piumarta and Riccardi since the instruction sequences are selected and rewritten dynamically, based on eliminating dispatches within basic blocks. As such, they do not need to consider selection strategies, or comment on the type of instruction sequences found in the programs.

Recent work by Casey et al. [3] also examines the use of superinstructions in Java programs. They use between 8 and 1024 superinstructions, and compare selection strategies based on static and dynamic analyses. They note the contrast between the simpler approach of selecting sequences based on static frequencies against the more effective dynamic approach, which they tailor on a per-program basis. Indeed, our approach of selecting sequences based on a dynamic analysis, but averaged across programs, might be seen as a compromise between the strategies presented by Casey et al. One current drawback to their approach is that it does not currently allow "quickable" instructions (such as `getfield`), which would eliminate many of the instruction sequences we have selected below.

Repetition among sequences of bytecodes occurring *statically* in the program source has been studied for the purposes of code or class file compression [1]. Antonioli and Pilz note that the range of instructions used varies between 25 and 113 different instructions, with considerable variance in frequency of usage [12].

An extensive study of the possibilities from Java bytecode compression for embedded systems is presented by Clausen et al. [2]. Here, a static analysis identifies basic blocks that are repeated in the source code, and these are replaced by *macro instructions*. Apart from its basis on static analysis, and it motivation for compression rather than speed, the approach of Clausen et al. is similar to the approach presented here.

Surveys of dynamic instruction usage in Java programs have been conducted for both the SPEC and Java Grande benchmark suites [8], [13]. A comparison of these suites noted a wide discrepancy in class library utilisation by these programs [14]. Preliminary work on the frequencies of instruction pairs has also been carried out [9], and the present work is a natural extension of that paper. A related issue is *instruction reuse* [15], [16], where a given instruction is executed dynamically many times with the same set of operands. While this does have implications for superoperators, it has not yet been studied in the context of Java bytecodes, and is beyond the scope of this paper.

III. SELECTING THE MOST FREQUENTLY OCCURRING SEQUENCES

Our approach involves forming a set of generic superinstructions based on studying instruction sequence usage in a suite of Java programs. We run each program in the suite, collect a trace of the bytecode instructions executed, and this then forms the input data for our analysis. Thus, in this section we examine some of the issues involved in selecting the most frequently occurring bytecode sequences, since these will be replaced by superinstructions in our implementation.

The strategy used in selecting these sequences naturally has an important bearing on our results, and we present this section formally in order to unambiguously describe the selection strategy.

A. Notation

Let us denote a sequence of bytecode instructions as $\hat{b} = [b_1, \ldots, b_n]$ where each b_i is a single bytecode instruction. Let us denote the length of a bytecode sequence as $|\hat{b}|$; clearly $|b_1, \ldots, b_n| = n$.

For any program run P, assume that we have collected a dynamic trace of all the instructions executed when P is run, and let us denote the sequence of bytecode instructions in this trace as T_P. Then the maximum number of (non-unique) sequence occurrences of length n in T_P is always $|T_P| - (n - 1)$.

Let us denote the number of actual occurrences of $\hat{b}$ in the trace of program P as $\Sigma_P(\hat{b})$; then we define the occurrence frequency for an sequence, expressed as a percentage, by:

$$f_P(\hat{b}) = \frac{\Sigma_P(\hat{b})}{|T_P| - (n-1)} * \frac{100}{1}$$

Relativising sequence occurrences by the length of the program trace allows us to compare sequences from different traces, since program size is no longer a factor. Since in practice the size of the program trace is much longer than the size of the sequences under consideration, we can approximate $|T_P| - (n-1)$ as $|T_P|$, thus allowing us to compare sequences of different lengths.

We note two straightforward properties of such bytecode sequences that will be useful in our calculations later:

- **Sequence Inclusion Property**
  A sequence $\hat{s}$ is included in some sequence $\hat{t}$ precisely when there exist integers i, j and n such that $1 \leq i < j \leq n$, and $\hat{t} = [b_1, \ldots, b_n]$ and $\hat{s} = [b_i, \ldots, b_j]$.
  We note that for any program P we have:

  $$f_P[b_1, \ldots, b_n] \leq f_P[b_i, \ldots, b_j]$$

  That is, the sequence $[b_i, \ldots, b_j]$ may occur in contexts other than $[b_1, \ldots, b_n]$; we note that it may also occur multiple times in $[b_1, \ldots, b_n]$, and that these occurrences may overlap.

- **Sequence Overlap Property**
  A sequence $\hat{s}$ overlaps some sequence $\hat{t}$ on the left precisely when there integers i, j and n such that $1 \leq i < j \leq n$, with $\hat{s} = [b_1, \ldots, b_j]$ and $\hat{t} = [b_i, \ldots, b_n]$. The definition of overlapping on the right is defined analogously.
  We note that the frequency with which this overlapping occurs is given by the frequency of composite sequence $f_P[b_1, \ldots, b_n]$. From the sequence inclusion property above we note that this is less than either $f_P(\hat{s})$ or $f_P(\hat{t})$, and the frequency of occurrence of the sequence $\hat{s}$ that do not involve an overlap with $\hat{t}$ is $f_P(\hat{s}) - f_P[b_1, \ldots, b_n]$

These properties have the side-effect of providing a consistency check on the frequency results.

A superinstruction is a new instruction that will denote some sequence of bytecode instructions. We will use lower case Greek letters to denote superinstructions and we write $\beta \equiv [b_1, \ldots, b_n]$ to mean that the superinstruction β corresponds to the sequence of bytecodes $[b_1, \ldots, b_n]$. Once a superinstruction has been defined it effectively becomes a new bytecode, and thus may occur in bytecode sequences and (non-recursively) in other superinstruction definitions.

B. Choosing the superinstructions

Suppose we have calculated the function f_P, giving the frequency of all bytecodes sequences for some program P. Let us assume that this function is total, so that $f_P(\hat{s}) = 0$ whenever $\hat{s}$ does not occur in T_P, the trace of P.

For our approach we wish to calculate the top k superinstructions, but we cannot simply choose the k sequences with the highest frequency, since we must allow for overlaps between sequences. Choosing some sequence $\hat{s}$ as a superinstruction has an impact on the frequencies of any remaining sequences whose bytecodes overlap with $\hat{s}$.

Thus we apply an iterative algorithm, where we choose the most frequently occurring sequence, and then propagate this choice through the remaining sequence, reducing the frequency of any sequence that it overlaps with. Each iteration produces a new set of frequencies, and we can then choose the next topmost superinstruction from these, and propagate this choice.

We note that this consideration of possible overlaps between sequences imposes an extra overhead on the information collected. If the maximum length of any instruction sequence under consideration is l, then we must gather data for all instruction sequences up to length $2l - 1$ in order to allow for the case of two sequences of length l overlapping by just a single instruction.

Propagation algorithm: Suppose we have chosen some superinstruction β.

Then, for each other bytecode sequence $\hat{s}$, either β and $\hat{s}$ do not overlap (in which case do nothing), or there are two cases, as illustrated in Figure 1

- **Case 1**: β is contained entirely within $\hat{s}$
  In this case the sequence is of the form $\hat{s} = [b_1, \ldots, b_n]$, and $\beta \equiv [b_i, \ldots, b_j]$ for $1 \leq i < j \leq n$

$$\textbf{Case 1}: \quad \underbrace{[b_1, \ldots, \overbrace{b_i, \ldots, b_j}^{s.inst.\ \beta}, \ldots, b_n]}_{sequence\ \hat{s}}$$

$$\textbf{Case 2}: \quad [\overbrace{b_1, \ldots, b_i, \ldots, b_j}^{s.inst.\ \beta}, \ldots, b_n] \qquad [b_1, \ldots, \underbrace{b_i, \ldots, b_j, \ldots, b_n}_{sequence\ \hat{s}}]$$

Fig. 1. The two cases where a chosen superinstruction β is either included in, or overlaps with some existing bytecode sequence $\hat{s}$

  Then replace this sequence with the sequence $[b_1, \ldots, b_{i-1}, \beta, b_{j+1}, \ldots, b_n]$, with the same frequency.

$$f_P([b_1, \ldots, b_{i-1}, \beta, b_{j+1}, \ldots, b_n]) = f_P(\hat{s});$$
$$f_P(\hat{s}) = 0;$$

- **Case 2**: β overlaps partially with $\hat{s}$
  Say, for the sake of definiteness, β overlaps bytecodes on the left of the sequence $\hat{s}$.
  In this case, let $\beta \equiv [b_1, \ldots, b_j]$, then the sequence has the form $[b_i, \ldots, b_n]$, where $1 \leq i < j \leq n$. The overlap is the sequence $[b_i, \ldots, b_j]$.
  The frequency of $[\beta, b_{j+1}, \ldots, b_n]$ must now be increased by the frequency of $[b_1, \ldots, b_i, \ldots, b_j, \ldots, b_n]$, and the frequency of the sequence $[b_i, \ldots, b_n]$ should be decreased by this amount.

$$f_P([\beta, b_{j+1}, \ldots, b_n]) \mathrel{+}= f_P([b_1, \ldots, b_n]);$$
$$f_P([b_i, \ldots, b_n]) \mathrel{-}= f_P([b_1, \ldots, b_n]);$$

The above process creates new sequences of bytecodes and superinstructions, and assigns them frequencies. Note that the same sequence of bytecodes and superinstructions may be created at different parts of the algorithm, and thus its corresponding newly-created frequency should be *added* to its existing total.

This process also deals with the case where a superinstruction may overlap some bytecode sequences multiple times. However, in the case where an superinstruction may overlap a sequence in two non-disjoint sections, a choice must be made between the superinstruction occurrences. We always choose to compress the leftmost occurrence to a superinstruction, since the bytecodes are being executed from left-to-right in the sequence.

C. Weighted Case

In this case we have a weighted frequency wf, where the frequency as calculated above is adjusted by some weighting factor w.

$$wf_P(\hat{b}) = f_P(\hat{b}) * w(\hat{b})$$

The weighting factor is meant to represent the potential gain from replacing this sequence of bytecodes with a superinstruction. In the simplest case the gain is equal to the number of fetch-cycles saved; that is:

$$w(\hat{b}) = |\hat{b}| - 1$$

Since the weighting factor is a function only of the bytecode sequence, it is easily woven into the algorithm from the last section. Each time a frequency is adjusted (corresponding to case 1 or 2 above), the weighted frequency is recalculated, counting each superinstruction as a single bytecode instruction.

IV. EXPERIMENTAL SETTING

The experiments in this section were conducted using Robert Lougher's Jam Virtual Machine [17]. The JamVM was specifically designed to have a very small footprint, but yet to support the full JVM specification [18]. The JamVM runs in interpreted mode only, but can be built to implement either switch-based or token threaded approaches (given support for first-class labels). It should be noted that JamVM uses the GNU *classpath* Java class library which is not 100% compliant with SUN's JDK, and may, of course, differ from other Java class libraries.

The platform used was a Dell Dimension 2350 PC, containing a 2.4 GHz Intel Pentium IV processor with a 512K level-1 cache, 1 GB of 266MHz DDR RAM, running the RedHat 9.0 distribution of GNU/Linux. The JamVM interpreter, version 1.0, was compiled using the GNU C compiler from *gcc* version 3.3. In what follows we use the programs from Pendragon Software's Embedded CaffeineMark version 3.0 [19] which is designed to benchmark embedded applications and Java-powered consumer electronics systems.[1]

A. Selecting the Superinstructions

In order to select the instruction sequences that will correspond to the new superinstructions, the CaffeineMark applications were run using a version of the JamVM that had been instrumented to record the instructions executed. Since our superinstructions are selected from within a basic block, the traces were reduced to frequency counts for basic blocks, and a sequence of Perl scripts was then used to collect frequency data on instruction sequences.

B. Superinstruction Length

Since at least 10 unused bytecode instructions are available in the JVM for implementing new superinstructions, the potential effectiveness of using superinstructions can be estimated by measuring the dynamic frequency of the top 10 sequences of each length.

Tables I through V give the frequencies for the top 10 sequences, where the sequence length was bounded by 2, 4 8, 16 and 32 instructions respectively. The top 10 sequences of size up to 32 instructions, shown in Table V, were exactly the same as those of length up to 64. Hence, in what follows, we have limited our study to sequences of up to 32 instructions.

In each of Tables I through V we list the top 10 instruction sequences. For each table, the first column lists the bytecode instructions in the sequence. The second column lists the frequency of the sequence, expressed as a percentage of the

Max. size = 02

Sequence	Frequency Original	Weighted
aload_0 getfield	8.91	8.91
aaload iload_1	2.55	2.55
istore iload	2.49	2.49
iconst_1 isub	2.31	2.31
iinc iload_3	1.36	1.36
iconst_0 goto	1.24	1.24
iload_1 ifne	1.16	1.16
aload_0 iload_1	1.16	1.16
iconst_3 if_icmplt	1.02	1.02
iload_3 iaload	0.87	0.87
Total (top 10)	23.05	23.05

TABLE I

TOP 10 MOST FREQUENT SEQUENCES OF SIZE UPTO 02, BASED ON WEIGHTED FREQUENCY.

Max. size = 04

Sequence	Frequency Original	Weighted
aload_0 getfield	8.91	8.91
aload_0 getfield iload aaload	2.29	4.58
istore iload ifeq	1.79	3.58
aload_0 iload_1 iconst_1 isub	1.14	3.41
aload_0 getfield iload_3	3.10	3.10
dadd dastore iinc iload_3	0.76	2.29
iconst_1 isub iaload	0.88	1.75
iadd putfield iload_1 ifne	0.58	1.74
aload_0 dup getfield iconst_1	0.58	1.74
istore iload iload iadd	0.58	1.73
Total (top 10)	20.60	32.83

TABLE II

TOP 10 MOST FREQUENT SEQUENCES OF SIZE UPTO 04, BASED ON WEIGHTED FREQUENCY.

Max. size = 08

Sequence	Frequency Original	Weighted
aload_0 getfield iload aaload iload_1	2.29	9.16
aload_0 getfield iload_3	3.10	6.21
daload dmul dadd dastore iinc iload_3 iconst_3 if_icmplt	0.76	5.34
aload_0 iload_1 iconst_1 isub invokevirtual	1.14	4.54
istore_2 aload_0 dup getfield iconst_1 iadd putfield iload_1	0.58	4.05
getfield aload_0 getfield iadd istore iload iload iadd	0.58	4.04
istore iload ifeq	1.79	3.58
daload aload_0 getfield iload_3 aaload iload daload	0.76	3.05
iconst_1 isub iaload aload_0 getfield iload_3 iaload if_icmpge	0.58	2.89
aload_0 getfield iload iaload	0.81	2.44
Total (top 10)	12.39	45.31

TABLE III

TOP 10 MOST FREQUENT SEQUENCES OF SIZE UPTO 08, BASED ON WEIGHTED FREQUENCY.

---

[1] The test was performed without independent verification by Pendragon Software and that Pendragon Software makes no representations or warranties as to the result of the test.

Max. size = 16

	Frequency	
Sequence	Original	Weighted
getfield iload aaload iload_1 daload aload_0 getfield iload_3 aaload iload daload aload_0 getfield iload aaload iload_1	0.76	11.45
istore iload iload iadd istore iinc aload_0 getfield iload_3 iconst_1 isub iaload aload_0 getfield iload_3 iaload	0.58	8.67
aload_0 getfield	8.91	5.46
daload dmul dadd dastore iinc iload_3 iconst_3 if_icmplt	0.76	5.34
iload_1 istore_2 aload_0 dup getfield iconst_1 iadd putfield iload_1 ifne	0.58	5.21
aload_0 iload_1 iconst_1 isub invokevirtual	1.14	4.54
istore iload ifeq	1.79	3.58
isub aload_0 getfield iload_3 iaload iastore aload_0 getfield iload_3 iload iastore iinc iload_3 aload_0 getfield if_icmplt	0.29	3.47
aload_0 getfield iload_3 iconst_1 isub iaload istore aload_0 getfield iload_3 iconst_1	0.29	2.31
aload_0 getfield iload iaload iload_1 iconst_2 idiv if_icmpgt	0.31	1.85
Total (top 10)	15.40	51.88

TABLE IV

TOP 10 MOST FREQUENT SEQUENCES OF SIZE UPTO 16, BASED ON WEIGHTED FREQUENCY.

Max. size = 32

	Frequency	
Sequence	Original	Weighted
aload_0 getfield iload aaload iload_1 aload_0 getfield iload aaload iload_1 daload aload_0 getfield iload_3 aaload iload daload aload_0 getfield iload aaload iload_1 daload dmul dadd dastore iinc iload_3 iconst_3 if_icmplt	0.76	22.14
aload_0 getfield aload_0 getfield iadd istore iload iload iadd istore iinc aload_0 getfield iload_3 iconst_1 isub iaload aload_0 getfield iload_3 iaload if_icmpge	0.58	12.13
aload_0 getfield iload_3 iconst_1 isub iaload istore aload_0 getfield iload_3 iconst_1 isub aload_0 getfield iload_3 iaload iastore aload_0 getfield iload_3 iload iastore iinc iload_3 aload_0 getfield if_icmplt	0.29	7.51
iload_1 istore_2 aload_0 dup getfield iconst_1 iadd putfield iload_1 ifne	0.58	5.21
aload_0 iload_1 iconst_1 isub invokevirtual	1.14	4.54
istore iload ifeq	1.79	3.58
aload_0 getfield iload iaload	0.81	2.44
iload_2 iconst_1 iand ifeq	0.57	1.70
aload_0 getfield iload_2 iinc caload istore aload_3 getfield iload iinc caload istore iload iload if_icmpeq	0.12	1.69
iconst_0 goto	1.24	1.24
Total (top 10)	7.87	62.19

TABLE V

TOP 10 MOST FREQUENT SEQUENCES OF SIZE UPTO 32, BASED ON WEIGHTED FREQUENCY.

total number of bytecodes executed. The final column lists the adjusted, weighted frequency, which allows for overlaps between the selected sequences, and uses a weighting factor of one less than the number of instructions in the sequence. It should be noted that there will be a higher overhead in recognising such sequences dynamically in the instruction stream, and that the actual (unweighted) frequency of longer sequences tends to be less than the frequency of shorter sequences. Both of these factors will tend to offset the possible benefits to be gained from using longer sequences.

From Tables I through V, we note the prevalence of the `aload_0 getfield` pair, which is the top sequence in Table I and II, and occurs frequently as part of the top sequences in Tables III throughV. It is also notable that the adjusted frequencies decrease rapidly as we move down the table, indicating diminishing possible returns for greater numbers of superinstructions, as predicted by Proebsting [4].

C. *Implementing the Superinstructions*

Once the sequences corresponding to superinstructions have been selected, it is then necessary to change the virtual machine to provide an implementation. This involves augmenting the main interpreter loop with cases for the extra instructions, and concatenating in the code corresponding to each original instruction as appropriate for each new superinstruction. Since little new code is involved, it is possible to make such modifications at run-time (as described by Piumarta and Riccardi [5]). However, since our goal was to measure the possible savings from superinstruction implementation, we generated the new code off-line, and recompiled versions of the JamVM for each of the four possible selections of superinstructions described in the previous subsection. One side-effect of implementing the superinstructions statically is that the new instruction sequences can be subjected to optimisations by *gcc*, a feature not available to dynamically-generated code.

It is also necessary to change the instruction stream for each application to include these new superinstructions. While this could be done statically, such an approach is cumbersome as it would also involve changing the code in the Java class libraries. Instead we implemented a “just-in-time” style of translation, where the instruction stream was modified dynamically the first time a sequence corresponding to a superinstruction was encountered at run-time.

When an instruction that could correspond to the first instruction of one of the superinstruction sequences was encountered at run-time, the instruction stream was checked to see if the following instructions matched the sequence. If so, the first instruction (only) was modified to become the corresponding superinstruction. If not, the instruction was modified to a “tagged” version of itself. This “tagged” version is coded to execute with the same semantics as the original, without the check for superinstruction sequence occurrence. Thus, the overhead of checking for a matching sequence only occurs the first time the initial bytecode of the sequence is encountered; if the instruction stream does not match a sequence, no overhead is incurred on subsequent iterations.

There are a number of other issues that need to be addressed when modifying the instruction stream in this way. First, with multi-threaded programs the possibility exists that two threads would attempt to modify the same instruction stream simultaneously; this issue is not addressed in this paper, but has been dealt with extensively by Gagnon and Hendren [7]. Second, most virtual machines implement "quick" versions of instructions, where, for example, indirect references to field names are replaced by direct references after the first execution. The JamVM implements 17 such instructions, and some of these are present in our instruction sequences (e.g. `getfield`). This does not present a problem for our approach; on the first pass through a sequence the instructions are changed to their "quick" versions, as usual. The second time through, those sequences of instructions corresponding to superinstructions are picked up by our modifications.

A final issue that must be considered is that of basic blocks, since, in general, control may be transferred in to or out of an instruction sequence. As noted earlier, we did not include instructions that could terminate a basic block internally in our sequences, so control cannot be transferred *out* of them. Since we have modified only the first instruction in the sequence, control transfers *in* to the sequence are not a problem, since the original bytecodes, other than the first, remain there unchanged. We note that one disadvantage of this approach is that we do not achieve any code size compression from implementing superinstructions.

V. RESULTS

In order to measure the effect of superinstruction implementation, three new versions of the JamVM were prepared, implementing the instructions sequences in Tables I through V. The JamVM as shipped actually implements the `aload_0 getfield` superinstruction, so a further version was prepared without this, in order to fully judge the effect of superinstruction implementation.

Thus, size different versions of the JamVM were used:

- **none** This is the basic JamVM with no superinstructions implemented
- **orig** This is a version of the JamVM as it is distributed, where only the `aload_0 getfield` superinstruction has been implemented
- **upto***n* A version of the JamVM with 10 superinstructions implemented; these are the superinstruction sequences of length upto *n*, as listed in Tables I through V.

In addition, each of these six versions of JamVM was built in both threaded and switch-based mode to give an estimation of the possible savings under each system. The data in Table VI records the results for running the six JamVMs over the CaffeineMark suite in a switch-based mode, whereas the data in Table VII shows the same information when the JamVMs are built using threaded dispatch. In each of Table VI and Table VII we report the result for each individual program in the suite, as well as the overall result. The numbers in the tables represent the CaffeineMark score, where a higher score indicates a greater number of operations performed per unit time.

	none	orig.	upto02	upto04	upto08	upto16	upto32
Sieve	1379	1591	1472	1464	1777	1768	1805
Loop	1433	1652	1617	1865	2130	2448	2599
Logic	2034	2065	1757	2041	2026	2074	1998
String	527	559	540	544	539	518	621
Float	1270	1363	1558	1782	535	439	517
Method	1527	1554	1686	2029	2510	2654	2836
Overall	1262	1363	1345	1490	1330	1324	1429

TABLE VI

RESULTS OF RUNNING EACH VERSION OF JAMVM, EACH BASED ON A *switched* INTERPRETER, OVER THE PROGRAMS FROM THE CAFFEINEMARK SUITE.

	none	orig.	upto02	upto04	upto08	upto16	upto32
Sieve	2146	2207	1822	2032	2335	2569	2353
Loop	1806	1940	1928	2300	2725	3504	3176
Logic	2465	2422	2348	2410	2368	2381	2373
String	633	644	591	596	597	592	704
Float	1913	1813	2122	2461	514	472	524
Method	1994	2016	2824	3063	3161	3258	3609
Overall	1687	1702	1752	1921	1563	1640	1693

TABLE VII

RESULTS OF RUNNING EACH VERSION OF JAMVM, EACH BASED ON A *threaded* INTERPRETER, OVER THE PROGRAMS FROM THE CAFFEINEMARK SUITE.

Looking at the overall results, we can see that, as expected, the threaded interpreter outperforms the switch-based interpreter.[2] Conversely, the speedup resulting from using superinstructions in the switch-based interpreter are greater than those for the threaded interpreter. This is to be expected, since the threaded interpreter has a reduced overhead for instruction dispatch, and so there is less to be gained from implementing superinstructions. The overall performance is summarised in Figure 2 for ease of comparison.

The best performing machine in each case is *upto04*, which shows an overall speedup of 18% in the switch-based interpreter, and 14% in the threaded interpreter. For the programs in this benchmark suite, superinstructions of length 4 would seem to represent the best compromise, maximising the frequency of occurrence, while minimising the overhead of implementation.

We note that there is significant variance between the performance of individual programs in the suite. The speedup achieved for both *Loop* and *Method* is quite dramatic, almost doubling their performance in the best case. The improvement for *String* and *Sieve* is relatively modest, and *String* actually exhibits a slight decrease in performance for all but the last machine. Clearly, in a real-world situation, it would be necessary to gauge the relative importance of the individual programs before selecting a particular optimisation level.

It is interesting to note the marked fall-off in performance of the *Float* program once the superinstruction length exceeds 4. This is attributable to the low frequency of occurrence of instructions relevant to *Float* in these longer sequences. A

---

[2] Interestingly, this was not the case when JamVM was compiled using *gcc* 3.2.2, where a compiler bug prevented the disabling of global common subexpression elimination (gcse), and the instruction dispatch sequence was hoisted.

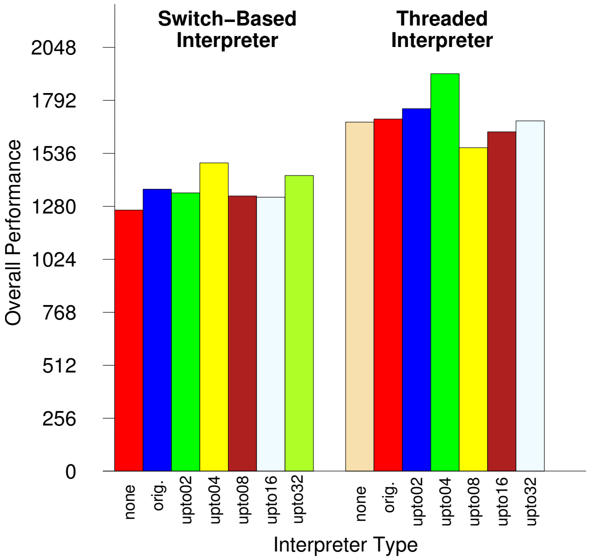

Fig. 2. Overall performance of the switch-based and threaded interpreters

clear implication of this is that our technique may not work for suites of programs with a highly heterogeneous mix of programs, and may actually inhibit performance in these cases.

VI. CONCLUSIONS

In this paper we have presented an approach to selecting and implementing superinstructions for Java programs based on an off-line analysis of a suite of programs. While not providing the same performance improvement as a per-program analysis, this approach has the advantage of eliminating the need for run-time profiling, as well as exposing superinstruction implementations to compiler optimisations.

As well as dealing explicitly with the possibilities of constructing a generic superinstruction set, this paper makes three other contributions not found in existing work:

- We formally present the instruction sequence selection procedure, based on a static analysis of dynamic program traces
- We list five possible selections of superinstruction sets, along with the corresponding distributions based on profiling programs in the CaffeineMark benchmark suite
- We have implemented the approach, and present results for small, generic superinstruction sets (as opposed to large basic-block results presented in previous work)

A number of further enhancements of this work are possible. At the moment we use a weighting factor based on the number of dispatch instructions saved. However, a weighting factor based on the possible optimisation of the resulting sequences might give better results. Also, it is possible that sets of superinstructions might be tailored for different types of applications (e.g. batch applications, GUI-based applications, scientific applications).

Our present analysis is based on individual instructions. However, merging similar instructions might lead to higher frequencies and thus better results. This might include equating specialised instructions with their generic counterparts, such as `iload_1` and `iload`, or even merging functionally similar bytecodes (e.g. `iload`, `aload` and `fload` all load a 32-bit value onto the stack).

REFERENCES

[1] Derek Rayside, Evan Mamas, and Erik Hons, “Compact Java binaries for embedded systems,” in *9th NRC/IBM Centre for Advanced Studies Conference*, Toronto, Canada, November 8-11 1999, pp. 1–14.

[2] Lars Rder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller, “Java bytecode compression for low-end embedded systems,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 3, pp. 471–489, May 2000.

[3] Kevin Casey, David Gregg, and Anton Ertl, “Towards superinstructions for Java interpreters,” in *7th International Workshop on Software and Compilers for Embedded Systems*, Vienna, Austria, September 24-26 2003.

[4] Todd A. Proebsting, “Optimizing an ANSI C interpreter with superoperators,” in *Symposium on Principles of Programming Languages*, San Francisco, California, January 23-25 1995, pp. 322–332.

[5] Ian Piumarta and Fabio Riccardi, “Optimizing direct-threaded code by selective inlining,” in *Conference on Programming Language Design and Implementation*, Montreal, Canada, June 17-19 1998, pp. 291–300.

[6] M. Anton Ertl, “Threaded code variations and optimizations,” in *EuroForth*, Saarland, Germany, November 23-26 2001, pp. 49–55.

[7] Etienne Gagnon and Laurie Hendren, “Effective inline-threaded interpretation of Java bytecode using preparation sequences,” in *Compiler Construction*, Warsaw, Poland, April 5-13 2003, pp. 170–184.

[8] David Gregg, James F. Power, and John Waldron, “Benchmarking the Java virtual architecture - the SPEC JVM98 benchmark suite,” in *Java Microarchitectures*, N. Vijaykrishnan and M. Wolczko, Eds., chapter 1, pp. 1–18. Kluwer Academic, 2002.

[9] Diarmuid O’Donoghue, Áine Leddy, James F. Power, and John Waldron, “Bi-gram analysis of Java bytecode sequences,” in *Second Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, Dublin, Ireland, June 13-14 2002, pp. 187–192.

[10] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan, “vmgen – a generator of efficient virtual machine interpreters,” *Software–Practice and Experience*, vol. 32, no. 3, pp. 265–294, 2002.

[11] M. Anton Ertl and David Gregg, “Optimizing indirect branch prediction accuracy in virtual machine interpreters,” in *Conference on Programming Language Design and Implementation*, San Diego, California, June 9-11 2003, pp. 278–288.

[12] D. Antonioli and M. Pilz, “Analysis of the Java class file format,” Technical Report 98.4, Dept. of Computer Science, University of Zurich, Switzerland, April 1988.

[13] David Gregg, James Power, and John Waldron, “Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 459–484, March 2003.

[14] Siobhán Byrne, James F. Power, and John Waldron, “A dynamic comparison of the SPEC98 and Java Grande benchmark suites,” in *First Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, Orlando, Florida, July 22-25 2001, pp. 95–98.

[15] Avinash Sodani and Gurindar S. Sohi, “Dynamic instruction reuse,” in *24th International Symposium on Computer Architecture*, Denver, Colorado, June 2-4 1997, pp. 194–205.

[16] Avinash Sodani and Gurindar S. Sohi, “An empirical analysis of instruction repetition,” in *8th International Symposium on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Oct 3-7 1998, pp. 35–45.

[17] Robert Lougher, “JamVM v. 1.0.0,” Available at the URL: http://jamvm.sourceforge.net/, March 10 2003.

[18] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison Wesley, 1996.

[19] Pendragon Software Corporation, “Caffeinemark 3.0,” 1997, http://www.benchmarkhq.ru/cm30/info.html.