CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2006; **18**:1465–1484
Published online 12 January 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1016

# Optimizing code-copying JIT compilers for virtual stack machines

David Gregg[1,*,†] and M. Anton Ertl[2]

[1]*Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland*
[2]*Institut für Computersprachen, Technische Universität Wien, Argentinierstrasse 8, 1220 Wien, Austria*

**SUMMARY**

**Just-in-time (JIT) compilers are widely used to implement stack-based virtual machines, such as the Java and .NET virtual machines. One disadvantage of most JIT compilers is that they are unportable; much of the back-end is specific to the target machine. An alternative to machine-specific code generation methods is to define a routine in a high-level language for each virtual machine instruction. These can be compiled to native code using a normal C compiler. The native code for these routines can then be strung together, allowing very simple, unoptimized code to be produced just in time. In this paper we present such a system based on an existing implementation of the Forth language. We present a novel system of optimizations for the system based on exploiting common sequences of virtual machine instructions. We use a small domain specific language and tool to generate stack-optimized code for sequences of virtual machine instructions, and for choosing the most useful sequences for a code-copying compiler. By measuring the length of the resulting executable code, we allow machine-specific sequences to be chosen without any machine-dependent code in our system. Experimental results show that best (average) speedups of 47.2% (15.75%) are possible on a Pentium 4 machine, and even higher an a PowerPC based machine. Furthermore, our optimizations allow the size of the generated code to be reduced by an average of 17.9% on the Pentium 4, and 20.5% on the PowerPC over a wide range of programs. Copyright © 2006 John Wiley & Sons, Ltd.**

KEY WORDS: compiler; interpreter; virtual machine

## 1. INTRODUCTION

Virtual stack machines such as the Java, Forth and .NET virtual machines are a popular choice as a portable target for compiling high-level languages. Virtual stack machines are usually implemented

*Correspondence to: David Gregg, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland.
†E-mail: david.gregg@cs.tcd.ie

*Received 16 May 2003*
*Revised 15 April 2004*
*Accepted 15 October 2004*

by an interpreter or just-in-time (JIT) compiler. JIT compilers provide the best performance, but interpreters have a number of advantages that make them attractive. Interpreters can be constructed to be trivially portable (retargetable) to new architectures. The same source code for the interpreter can run on many different target architectures, reducing the cost of developing and maintaining the virtual machine. They are a great deal simpler to construct than JIT compilers, making them more reliable, quicker to construct and easier to maintain. Finally, interpreters can require far less memory than a JIT compiler, both for the interpreter itself and the interpreted virtual machine code.

The main drawback of interpreters is that they run code much more slowly than native code produced by a compiler. A highly efficient interpreter is typically a factor of ten slower for general-purpose programs over native code produced by an optimizing compiler [1]. For this reason, interpreters are used primarily where retargetability, development time, maintenance costs or memory requirements are more important than execution speed.

An alternative solution is the code-copying JIT compiler. These are written in a high-level language, and contain almost no architecture-specific code. Instead, a C routine is written to implement each virtual machine instruction. This routine is compiled to machine code at the time that the JIT compiler is built. At run time, machine code for arbitrary virtual machine code can be generated by stringing together copies of the executable machine code for each virtual machine instruction. In this way simple, unoptimized native machine code can be generated almost without machine-specific source code.

Code-copying JIT compilers are only slightly more complicated to implement than an efficient interpreter. They are easier (cheaper) to maintain than non-portable compilers because only a single version of the source code is used for every target architecture. The main disadvantage is that, like standard JIT compilers, the generated native code is usually much larger than virtual machine code. In fact, code-copying compilers usually produce much larger code than regular JIT compilers, because no optimizations are performed to reduce code size.

In this paper we present novel techniques for improving the quality of code generated by code-copying JIT compilers, without reducing their portability. Our techniques are based on stack optimizations for superinstructions [2]. A superinstruction is a common sequence of virtual machine instructions, which we combine into a single unit. We use a small domain-specific language to describe the stack effects of virtual machine instructions. Using a source code generator tool, we automatically optimize the stack operations in virtual machine instructions, allowing us to produce smaller, faster code. We provide heuristics for choosing those superinstructions which give the best reduction in code size and operations performed, and present experimental results.

The remainder of this paper is organized as follows. The next section introduces our system of code fragments in C. Section 3 describes how we use GNU C to copy executable machine code. In Section 4 we explain the main differences between code produced by a code-copying compiler and a simple machine-dependent JIT compiler. Section 5 describes our language and tool for defining the behaviour of virtual machine instructions. Optimizations to reduce the cost of accessing the virtual machine stack are presented in Section 6. Section 7 introduces a scheme for reducing register pressure by shortening the live ranges of local variables in the C code to implement virtual machine instructions. Section 8 considers strategies for choosing which sequences of virtual machine instructions to optimize. Finally, we present experimental results in Section 9.

## 2. VIRTUAL MACHINES AND CODE FRAGMENTS

Programs for the Java Virtual Machine (VM) and similar machines use a flat, sequential layout of the operations (in contrast to, e.g., tree-based intermediate representations), similar to real machine code. Such intermediate representations are called virtual machine code, and each operation is known as a virtual machine instruction. VM code is portable, and is usually designed to be both easy to interpret and compile.

The basis of our scheme is a set of fragments of C code, each of which implements a VM instruction. When compiled, the corresponding native machine code can be copied and strung together to generate native code for arbitrary computations. For this scheme, we need two things. First, we must ensure that when the code is copied that the diverse segments of native code will work together. This means that values that are live from one segment to another must be stored in the same register or memory location in both routines. The compiler should also avoid optimizing the code in any way that moves functionality from one segment of code to another. The other feature we need for this scheme to work is some way to copy executable code.

We solve both problems using GNU C's labels-as-values extension. GNU C (the language accepted by the GCC compiler) allows the programmer to take the address of a label, and store that address in a variable. The programmer can use a goto statement that jumps to the address stored in a variable.

In the source code for our system, a label is placed at the start and end of each of the C code fragments that implements a VM instruction. The address of each of these labels is taken and stored in an array of labels. In addition, to ensure that all fragments can safely follow one another, a small section of code containing a goto through a pointer variable is added after each fragment.

The resulting source code is almost identical to that of a threaded code interpreter [3]. A threaded code interpreter has a fragment of code to implement each instruction in the VM's instruction set. Each such fragment is followed by a goto through the VM's instruction pointer. Owing to the similarity, it is actually easier to build a code-copying JIT compiler by first constructing a threaded code interpreter, since it allows us to build a working system more quickly, which can later be optimized to make it more efficient. Figure 1 shows an outline of the source code. In the next section, we show the process of copying executable code.

## 3. CODE COPYING

We use GNU C's labels as values extensions to copy executable native code. By taking the addresses of the `start` and `end` of the fragment, we can copy the code in between using the `memcpy` function. Native code for an entire basic block of VM code can be generated by placing the copies sequentially in memory, so that the code for each instruction falls into the following one.

It is important to note that the original VM code must be kept, despite the translation to native code. The reason is that in addition to VM instructions, VM code normally contains inline immediate operands. For example, the JVM instruction `BIPUSH` ('byte immediate push') consists of the `BIPUSH` opcode, followed by a 1-byte immediate operand representing the immediate value to be pushed onto the stack. To ensure that the executable code can access these operands, we must keep the original code, and update the VM's instruction pointer for each instruction, to keep track of the point of execution in the original code.

```
typedef void *Inst;

void engine()
{
  static Inst start_labels[] =
              { &&iadd_start /* ... */ };
  static Inst ends_labels[] =
              { &&iadd_end /* ... */ };
  Inst *ip;
  int *sp;

  goto *ip++;

 iadd_start:
  sp[1]=sp[0]+sp[1];
  sp++;
 iadd_end:
  goto *ip++;
}
```

Figure 1. Instruction fragments as a threaded interpreter, using GNU C.

Generating executable code that extends across basic block boundaries is a little more complicated. Falling through from one basic block to another can be achieved by simply placing the executable code for the two blocks sequentially in memory. To implement VM branches, we use a GCC goto to jump to the address contained in a variable. Each VM branch takes an immediate argument, which is written into the original VM code. The immediate contains the address of the code which the branch should jump to. To implement this correctly, we must map branch offsets in the VM code to addresses in the copied executable code. The resulting native code is not as efficient as would be produced by a normal JIT compiler, but the corresponding source code is completely portable.

There is one part of a JIT compiler that cannot be implemented portably, however. Many processors place limitations on executable code being generated or modified at run time. Processors usually assume that code in the instruction cache will not be modified. Therefore, after generating executable code, the instruction cache must be flushed before the generated code runs. Otherwise, program behaviour can be unpredictable. There is no standard function for flushing the instruction cache, so each time the compiler is ported to a new architecture, an assembly language routine must be written to flush the i-cache. This typically consists of less than five lines of assembly language.

One problem with copying executable code is that it can only copy code that is relocatable; i.e. it cannot copy code if the code fragment contains a PC-relative reference to something outside the code fragment, or if it contains an absolute reference to something inside the code fragment. Whether the code for a VM instruction is relocatable or not depends on the architecture and on the compiler; so, a general no-copying list [4] is not sufficient.

Our approach to this problem is to have two versions of the VM interpreter function, one with some gratuitous padding between the VM instructions. We compare the code fragments for each VM instruction of these two functions; if they are the same, the code fragment is relocatable, if they are different, it is not. This solution is not perfect. In particular, it depends on the compiler producing the same executable code for both copies of the source code. We know of no good general solution to this problem of detecting which code can be safely copied. But our existing scheme works well in practice.

## 4. WHY IS THIS CODE SLOW?

Copying executable fragments allows us to generate very simple, unoptimized code just in time. There are a number of important differences between our copied code and that generated by a normal simple JIT compiler, however. These differences limit the performance of our copied code.

- *Stack accesses*. A VM instruction for a stack-based VM typically reads and/or writes a number of items from the stack. The stack is usually represented as an array in memory, so each stack access involves an expensive memory access. In contrast, even simple JIT compilers usually keep the more important values in registers.
- *Non-local optimizations*. Each fragment of code to implement a VM instruction is generated in isolation. There is no attempt to take advantage of inter-VM instruction optimization opportunities. For example, instruction scheduling, register allocation and copy propagation might all benefit from the compiler being able to see a larger 'window' of code to optimize. Simple JIT compilers usually apply limited optimizations across VM instruction boundaries.
- *Immediate arguments*. Immediate arguments are stored in VM code at an offset from the VM instruction pointer. Thus, accessing immediate arguments is expensive, since it requires an address calculation and a memory access. We must also maintain the value of the original VM instruction pointer by updating it for every VM instruction executed. It is important to note that all branches and calls involve accessing an immediate argument, so the cost is considerable. A regular JIT compiler can avoid most of this work. Most native instruction sets allow small constants to be expressed as immediate arguments in machine code, and the offsets for native code branches are always included in the code.
- *Indirect branches*. Our code-copying scheme uses a goto through a pointer variable to implement all branch, call and return VM instructions. When this C code is compiled, GCC normally generates an indirect branch to implement this goto. Indirect branches are more expensive than direct branches on most architectures. One of the main reasons for this is that while most processors provide a sophisticated two-level predictor for direct (conditional) branches, the predictor for indirect branches is usually much simpler (typically a one-level branch target buffer [5]).

In this paper, we deal with the first two of these problems. Our approach is based on replicating C source code to expose larger regions of source to optimization by the C compiler. We also automatically apply source-level optimizations to move parts of the stack into registers, and reduce the cost of communication between VM instructions. In the following section we explain our system for applying these optimizations.

```
IADD ( iValue1 iValue2 -- iResult ) 0x60
{
  iResult = iValue1 + iValue2;
}
```

Figure 2. Definition of IADD VM instruction.

## 5. VM INSTRUCTION DEFINITION

We use a combination of a small domain-specific language and C code to describe the behaviour of VM instructions. This description is then passed into Vmgen, a tool for specifying and generating VM interpreters. The definition of an instruction consists of a specification of the effect on the stack, followed by C code to implement the instruction. Figure 2 shows the definition of the JVM instruction `IADD`. The instruction takes two operands from the stack (`iValue1,iValue2`), and places the result (`iResult`) on the stack.

Figure 3 shows a simplified version of the output from Vmgen for the `IADD` instruction. It begins with the label that allows us to find the start address of the code. Then the stack items used by the instruction are declared. Next, there are two assignments to fetch the values of the stack items from the stack in memory. The `IADD` instruction reduces the height of the stack by one, so the stack pointer is updated by adding $-1$. The next piece of code is a direct copy of the C code in the instruction specification. After that, an assignment to store the resulting value back to the top of the stack appears. Then the code updates the original VM instruction pointer. This is necessary because some VM instructions need to access immediate arguments which are embedded in the original VM code. This is followed by the end label, which allows us to find the address of the end of this block of code. Finally, the macro `NEXT` appears, which we define elsewhere to be a goto using the label stored in a pointer variable as the target.

Another optimization that Vmgen can automatically apply is to keep the topmost stack element in a local variable. This local variable will usually be allocated to a register by the C compiler. Ertl [6] found that caching this topmost stack element in a register reduces memory traffic due to stack accesses by almost 50% over a wide range of programs. Figure 4 shows a simplified version of the output from Vmgen when this optimization is applied, and the topmost item on the stack is kept in the local variable `TOS`. Note that one of the loads from stack and the store have been replaced with register copies, reducing total memory operations by two-thirds.

The generated C code from Vmgen looks long and inefficient (and the complete version is even longer), but GCC optimizes it well and produces good code. In particular, assignments that copy a value from one local variable to another are usually optimized away. Figure 5 shows the x86 assembly code for the code in Figure 4, with VM register names instead of real register names.

The assembly is an efficient implementation of the corresponding C code. However, the quality of the code is still far from that of a simple JIT compiler. In the next section we introduce an optimization based on code replication to reduce this gap in code quality.

```
START_IADD:      /* start label */
{
int iValue1;     /* declaration of.. */
int iValue2;     /*  ...stack items  */
int iResult;
iValue1 = *(sp-1); /* fetch stack items */
iValue2 = *sp;
sp += -1;        /* stack pointer update */
{                /* user provided C code */
  iResult = iValue1 + iValue2;
}
*sp = iResult;   /* store stack result */
ip++;            /* update VM ip */
}
END_IADD:        /* end label */
NEXT;            /* indirect goto */
```

Figure 3. Simplified Vmgen output for IADD VM instruction.

```
START_IADD:      /* start label */
{
int iValue1;     /* declaration of.. */
int iValue2;     /*  ...stack items  */
int iResult;
iValue1 = *(sp-1); /* fetch stack items */
iValue2 = TOS;
sp += -1;        /* stack pointer update */
{                /* user provided C code */
  iResult = iValue1 + iValue2;
}
TOS = iResult;   /* store stack result */
ip += 1;         /* update VM ip */
}
END_IADD:        /* end label */
NEXT;            /* indirect goto */
```

Figure 4. Simplified Vmgen output for IADD VM instruction, with topmost stack item in a local variable.

```
mov -4(ip), tmp      ; tmp = *(sp-1)
add $-4, sp          ; sp += -1
add $4, ip           ; ip += 1
add tmp, TOS         ; TOS = TOS + tmp
```

Figure 5. x86 assembly for `IADD`.

```
SIPUSH ( #iImmediate -- iResult ) 0x11
{
  iResult = iImmediate;
}
```

Figure 6. Definition of `SIPUSH` VM instruction.

```
mov 4(ip), tmp       ; tmp = *(ip+1)
mov TOS, (sp)        ; *sp = TOS
add $-4, sp          ; sp += 1
add $8, ip           ; ip += 2
mov tmp, TOS
```

Figure 7. x86 assembly for code `SIPUSH`.

## 6. SUPERINSTRUCTIONS

A superinstruction is a new VM instruction that consists of several existing VM instructions. By combining several simple VM instructions into one larger instruction, it is possible to perform inter-VM instruction optimizations.

Vmgen allows us to generate superinstructions, using profiling information. For example, we might find that the VM instructions `SIPUSH` and `IADD` occur very frequently in sequence in Java programs. Given their frequency, it may be worthwhile to create a superinstruction `SIPUSH_IADD` which behaves exactly like the original sequence of VM instructions but may be more efficient.

Figure 6 shows the instruction definition for the JVM instruction `SIPUSH` (short immediate push). The # symbol in the definition means that it takes an immediate value from the VM instruction stream. When passed through Vmgen and compiled with GCC, we get the corresponding assembly code, shown in Figure 7. Note that to push a new item onto the stack, we need to store the existing value in the variable `TOS` into the stack in memory. Also, we need to update the instruction pointer by two positions, since the VM instruction consists of the `SIPUSH` opcode followed by an immediate operand containing the value to push.

```
START_SIPUSH_IADD: /* start label */
{
int sp0;  /* synthetic names */
int sp1;
int ip1;  /* synthetic name for item... */
          /* ..in VM instruction stream */
ip1 = *(ip+1); /* fetch immediate */
sp0 = TOS;     /* fetch stack item */
{ /* SIPUSH */
  int iImmediate; /* declare stack item */
  int iResult;
  /* fetch stack item to local variable */
  iImmediate = ip1;
  {               /* user provided C code */
    iResult = iImmediate;
  }
  sp1 = iResult;  /* store stack result */
}
{ /* IADD */
  int iValue1;    /* declare stack items */
  int iValue2;
  int iResult;
  iValue1 = sp1;  /* fetch stack items to */
  iValue2 = sp0;  /* ...local variables */
  {               /* user provided C code */
    iResult = iValue1 + iValue2;
  }
  sp0 = iResult;  /* store stack result */
}
TOS = sp0;
ip += 3;          /* update VM ip */
}
END_SIPUSH_IADD:  /* end label */
NEXT;             /* indirect goto */
```

Figure 8. Simplified Vmgen output for SIPUSH_IADD superinstruction.

By adding SIPUSH_IADD to the list of superinstructions for our code-copying compiler, Vmgen will produce the source code in Figure 8, which is generated automatically from the instruction definitions of SIPUSH and IADD.

There are a number of notable features about this code. First, all used stack items are loaded from memory into local variables at the start of the code. The different VM instructions within the superinstruction communicate by reading from and assigning to these local variables. Presuming that the C compiler is able to allocate these local variables to registers, this will greatly reduce the amount of memory traffic from accessing the VM stack. IADD alone requires one load to access the stack,

```
mov 4(ip), tmp       ; tmp = *(ip+1)
add $12, ip          ; ip += 3
add tmp, TOS         ; TOS += tmp
```

Figure 9. x86 assembly for code SIPUSH_IADD.

and SIPUSH requires one store access to the stack. In contrast, the superinstruction SIPUSH_IADD deals only with the topmost stack item, and thus eliminates all stack-memory traffic entirely.

Another notable feature of the code in Figure 8 is that there is no stack pointer update. SIPUSH increases the size of the stack by one, and IADD reduces its size by one. Vmgen detects that the two-stack pointer updates are redundant, and eliminates them. In addition, there is only one instruction pointer update.

Although the source code for SIPUSH_IADD looks large and complicated, it compiles to very efficient assembly (see Figure 9). Remarkably, the assembly code for the superinstruction is actually shorter than the code for *either* of the component VM instructions. Copying the executable code for this superinstruction rather than the executable code for the component instructions allows our code-copying JIT compiler to generate code that is more than a factor of two times smaller and (probably) faster for this particular sequence of VM instructions. If we can find a frequently used set of instruction sequences that are also suitable for inter-VM instruction optimizations, we should be able to produce smaller, faster code for all programs.

## 7. AN IMPROVEMENT

Loading all used stack items to registers at the start of a superinstruction is a very effective way to limit the memory accesses to the stack. One weakness of this approach, however, is that it creates a large number of local variables containing live values at the start of the superinstruction. When we compile the superinstruction source code, the C compiler attempts to place these local variables in registers. Given the large number of live variables, the register allocator may spill some of the values to memory. If this happens, the generated machine code will be much worse than if we had not copied the stack items to local variables. The stack item will be copied from memory to the local variable, which is another location in memory. Each access to the local variable requires a load or store. We found that this was a particular problem on the Intel x86 architecture, which has a small number of general purpose registers. In many cases, spilling causes the executable code for the superinstruction to be longer than the lengths of the component VM instructions combined.

We observed that this spilling problem arises primarily when some values that are loaded at the start of the superinstruction are often not used until close to the end of the superinstruction. Ideally, the C compiler would move these loads downwards to reduce register pressure. However, this is usually impractical because of possible pointer aliasing problems.

Our solution to this problem is to modify Vmgen to reduce register pressure by moving loads of stack items to later in the superinstruction and placing stores earlier. We compute the set of stack items used by each component VM instruction. A value is loaded from the stack only at the start of the code for the first component instruction that uses that value. A value is stored to the stack as soon as it is used for the last time. This reduces register pressure by shortening the ranges of live values.

## 8. WHICH SEQUENCES?

Given the facility to generate optimized superinstructions, the key question is which sequences should be chosen. Essentially there are three desirable features of a potential superinstruction. First, the sequence of simple VM instructions that it replaces should occur frequently in real programs. Secondly, the code fragment for the optimized superinstruction should be faster than the sequence of fragments it replaces. Finally, the machine code fragment should occupy less memory than the fragments for the sequence it replaces.

Choosing sequences of VM instructions that appear frequently in programs is relatively straightforward, even though the actual program that will be run is unknown at the time that the sequences are chosen. Gregg and Waldron [7] evaluated a wide range of strategies for choosing generally useful sequences, given a set of benchmark programs. They found that the best strategy was simply to choose those sequences that appear statically most frequently in the test programs. Sequences that appear statically many times in one program are also likely to be executed many times in other programs.

The second criterion is more difficult to evaluate. By examining the assembly language for a native code fragment, it might be possible to estimate the efficiency of a machine code fragment. However, our code-copying compiler must be entirely architecture independent. To our compiler, the machine code is an opaque sequence of bytes. We therefore suggest the following proxies.

First, when generating the C code for each superinstruction Vmgen keeps track of three pieces of information—the number of loads from the stack to local variables, the number of stores to the stack from local variables, and the number of times that the stack pointer is updated. As we saw in Section 6, the optimizations applied by Vmgen reduce the number of loads, stores and updates for certain combinations of instructions. This information can be computed by generating the C code for all possible superinstructions. By choosing sequences that reduce the number of stack operations, we should generate faster code.

An alternative strategy is to measure the length of the machine code for all potential superinstructions. We generate C code for all sequences and place it inside a function which measures the number of bytes between the start label and end label for each instruction. Once compiled on the target architecture, we can compare the length of the superinstruction with the total length of all component instructions. If the executable machine code is shorter, it is likely (but not certain) that it also runs faster.

A nice feature of measuring the length of executable machine code is that it allows us to make a machine-dependent comparison entirely portably, with no machine-dependent code. As part of the configuration process, our compiler system can use GCC to generate sample machine code for various sequences, and compare the lengths of those sequences on that particular architecture. Using this information, suitable superinstructions can be chosen for the build of our portable JIT compiler system.

The third desirable property of a superinstruction is that its executable code occupies less space than the executable code for each of its component VM instructions. When compiling a large program,

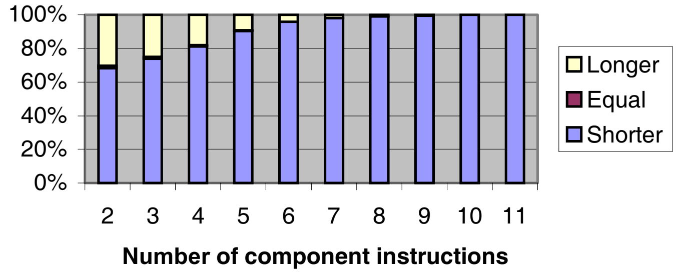

Figure 10. Breakdown of the proportion of superinstructions whose x86 executable code is smaller, equal in size, or larger than the sum of the sizes of the executable code for the component instructions.

each executable fragment is likely to be copied many times. Furthermore, as outlined in Section 4, the general purpose fragments we generate are usually much larger than the context-specific code that a regular JIT compiler can generate. Thus, code size is likely to be much more important for a code-copying JIT compiler than for a regular, machine-specific compiler.

## 9. EXPERIMENTAL RESULTS

To evaluate the potential usefulness of our optimized superinstructions in a code-copying JIT compiler, we first generated a large number of superinstructions and compared them with the code for their component instructions. We used the Gforth system, a product quality implementation of the Forth language which is available under the GNU general public licence. Forth is a stack-based programming language used primarily for low-level programming (such as the OpenBoot system, used to boot all Sun workstations) and embedded systems. We generated x86 machine code for superinstructions for 3706 sequences of VM instructions from several general purpose Forth programs, varying in length from 2 to 11 component instructions.

Figure 10 shows a breakdown of the proportion of superinstructions whose x86 executable code is smaller, equal in size, or larger than the sum of the sizes of the executable code for the component instructions. In the case of short superinstructions, containing just two component instructions, the executable code for the superinstruction is actually larger than the sum of the code size for the component instructions in 30.2% of cases. However, as the number of component instructions increases, this frequency drops off very quickly. There are two effects at work here. First, as the number of component instructions rises, Vmgen has more opportunity to eliminate redundant stack memory accesses and stack pointer updates. Secondly, as the body of source code presented to the C compiler becomes larger, more optimizations become possible. However, it is important to note that short supersinstructions are those most likely to be used frequently, so the 30.2% figure for superinstructions of length 2 is somewhat disappointing.

Figure 11 shows percentage size reductions for those superinstructions which are shorter than the sums of the lengths of the component instructions. Where the length of the executable code of the

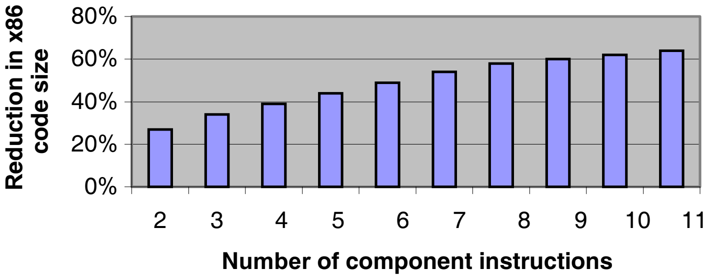

Figure 11. Average percentage reduction in the size of the x86 executable code of superinstructions compared to the sum of the sizes of the executable code for the component instructions.

superinstruction is shorter, the savings are significant. Those two-component superinstructions which are shorter are on average 27% shorter. As the number of components increases the savings become even larger. It is also important to note that these are average figures. Our code-copying system can choose the superinstructions that give above average savings, presuming that they appear in real programs reasonably frequently. As we show in the next paragraphs, this is a valuable strategy for superinstruction selection.

To test their usefulness, we implemented a code-copying compiler and superinstructions in the Gforth system. The system is entirely automated. To add superinstructions, one runs a set of sample benchmark programs to collect profiling information on the frequencies of sequences of Forth VM instructions. These frequencies can be fed into Vmgen using a series of scripts, resulting in a list of superinstructions that we want added to the system. Vmgen automatically generates optimized C source code for each superinstruction based on the instruction definition of the component instructions.

We tested two superinstruction selection strategies. First, we chose sequences to be superinstructions by simply choosing the most statically frequently occurring sequences across a range of benchmark programs. In order to prevent any single program dominating the statistics, we considered only sequences that appeared in at least two of the benchmark programs. Previous work [7] has shown this simple strategy to be highly effective. Our second strategy was more complicated. Superinstructions were ranked on the basis of a combination of static frequency, and the percentage reduction in executable code size of the superinstruction compared with the sum of the sizes of the executable code for the component instructions. The largest reductions in code size are usually for very long superinstructions. To avoid biasing our selection too heavily towards long, perhaps infrequently used superinstructions, we divided the product of the frequency and the percentage reduction in code size by the number of component instructions in the superinstruction.

To find the lengths of the executable code for all potential superinstructions we would have to generate source code for all possible supersinstructions, and compile this code. To reduce the search space, we generated superinstructions only for those sequences of VM instructions that appear in at least two of our benchmark programs—a total of 4761 superinstructions. Generating the machine code for all these superinstructions took some hours, but the process is entirely automated, and must be performed only once when the code-copying compiler is ported to a new type of machine.

Furthermore, examining the length of the executable code allows us to choose superinstructions that work well on a particular type of machine without knowing anything about the architecture of the machine. It allows machine-dependent optimization, implemented entirely machine independently.

At run time, our system must choose when and where to replace sequences of simple VM instructions with a superinstruction. Often, for a given basic block, there are several possible choices of how the simple instructions might be replaced. The process of choosing which replacement is most suitable is known as *parsing* the sequence, a term from dictionary-based data compression [8]. We chose to use an optimal strategy based on dynamic programming. Although the overhead of optimal parsing causes a small execution time penalty, it ensures that the reduction in code size is the very greatest possible.

To test the usefulness of superinstructions in a code-copying compiler for Forth, we tested the performance of the generated code for several benchmark programs. The programs are as follows.

- *brainless*. A chess playing program.
- *gray*. A parser generator which accepts an LL1 grammar and produces a recursive descent parser in Forth.
- *pentomino*. A puzzle-solving problem that tries to fit 12 pentominos (shapes constructed from five squares) together. Pentomino includes a code generator that produce large numbers of very similar new routines at run time.
- *tcsp*. Tom Kerrigan's chess engine (v1.73) ported from C to ANS Forth.
- *bench-gc*. A conservative garbage collector for Forth. It was run with a test program which allocates and collects large amounts of memory.
- *brew*. An evolutionary programming playground, which simulates the interaction between creatures.
- *mach32.l*. A GForth utility program.
- *prims2x*. A VM interpreter generator which forms part of the Gforth system. It accepts a specification of the VM instructions and outputs C source for an interpreter.
- *cd16v11*. A cycle-accurate simulator for a simple processor.
- *fib*. A naive recursive computation of the first 34 Fibonacci numbers.
- *matrix*. Multiplication of 200 by 200 matrices.
- *sieve*. Sieve of Eratosthenes program which computes all primes between 2 and 8190.
- *bubble*. Bubble sorts an array of 6000 elements.

All benchmark programs were run with a modified version of GForth 0.6.2 generating executable code just in time. The most important change we made to this version of GForth was to greatly speed up the optimal parsing algorithm, to reduce the overhead of using superinstructions. GForth was compiled using GNU GCC version 2.95. Experiments were run on two different machines: a Pentium 4 based machine with 1 GB of main memory running Red Hat Linux 7.2, and a PowerMac G4 machine with a 450 MHz PowerPC7400 processor and 256 MB RAM.

Figures 12 and 13 show the speedup of our code-copying GForth compiler due to superinstructions. The baseline is a version with no superinstructions, and speedups are shown for varying numbers of superinstructions across the benchmarks programs. Separate figures are also shown for our two superinstruction selection strategies based on static frequency only (so) and a combination of static frequency and reduction in length of the executable superinstruction code (sl). A number of features are notable.

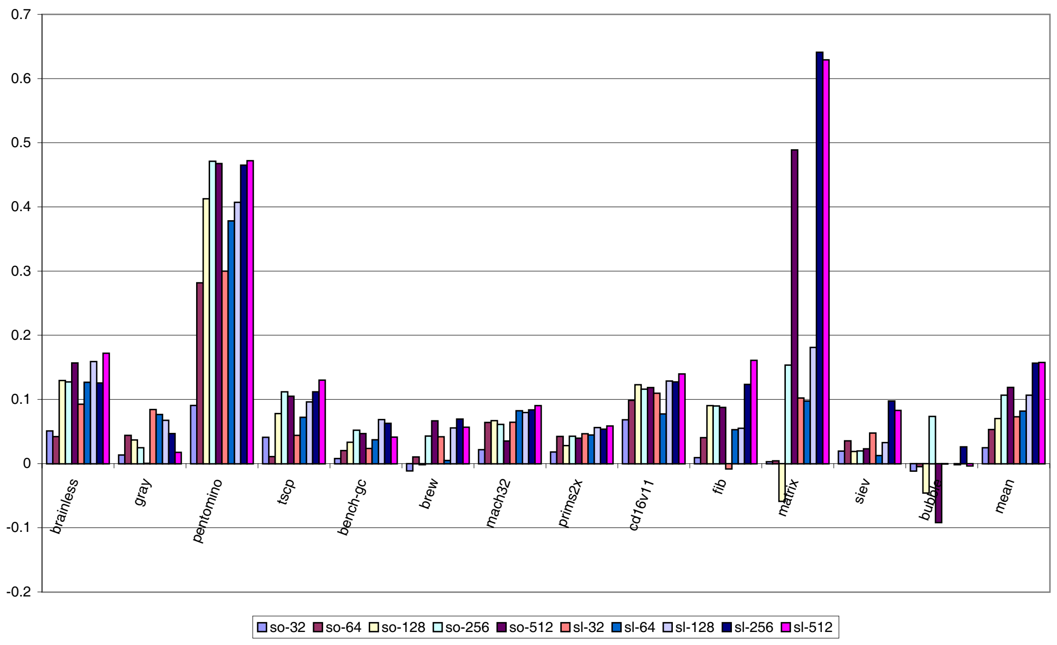

Figure 12. Speedup of the generated x86 code on a Pentium 4 based machine for benchmarks programs using a varying number of superinstructions, selected based on the combination of static frequency and length (sl), or only static frequency alone (so).

First, superinstructions give very significant reductions in some programs, such as pentomino where 512 superinstructions chosen on a mixture of static frequency and executable instruction length allows a speedup of up to 47.2% on the Pentium 4 and 89.5% on the PowerPC. On the other hand, the reduction in running time for other programs such as gray, brew and bubble is very small. In fact, the overhead of optimal parsing combined with the fact that using superinstructions sometimes results in worse code being generated means that brew and bubble run more slowly with some choices of superinstructions.

A second notable feature is that choosing superinstructions based on frequency and reduction in the length of the generated executable code is superior to choosing them simply on frequency alone. Not only are the speedups greater on the programs that benefit significantly from superinstructions; just as important, on the programs where the benefit is small or even negative, adding executable code length to the selection strategy greatly reduces the likelihood that superinstructions will result in worse performance. Overall, with 512 superinstructions, the static only selection strategy gives a speedup of 11.9%, whereas adding executable code length to the selection strategy yields an average speedup of 15.75% on the Pentium 4. On the PowerPC, the difference between using length information is less, but still consistent across different benchmarks.

Figures 14 and 15 show the reduction in the size of the executable code generated by our code-copying JIT compiler for the various benchmark programs, using varying numbers of superinstructions

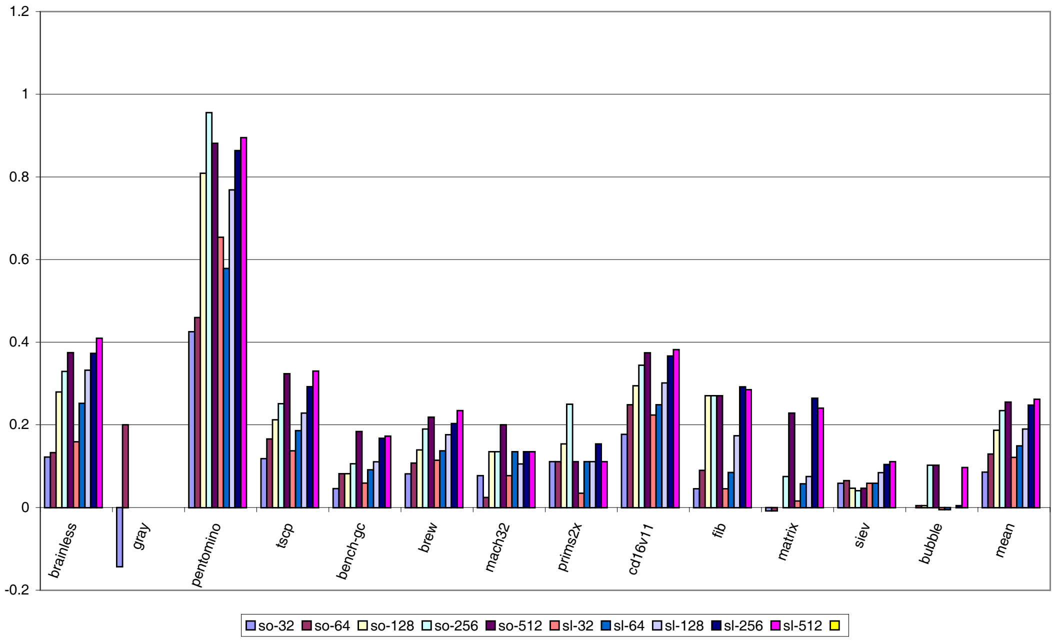

Figure 13. Speedup of the generated PowerPC machine code on a PowerPC7400 based machine for benchmarks programs using a varying number of superinstructions, selected based on the combination of static frequency and length (sl), or only static frequency alone (so).

and the two selection strategies. Unlike running times, superinstructions always reduce the size of the executable code, since our optimal parsing strategy only applied superinstructions where code size reductions will occur.

The code size reduction results essentially fall into three categories. First, the reduction for pentomino is large at 22.9% on the Pentium 4 and 29.0% on the PowerPC for the best strategy. We investigated this and found that Pentomino contains many very large basic blocks containing 15 or more VM instructions. These large basic blocks allow many opportunities to apply superinstructions. In contrast, in prims2x over 40% of basic blocks consist of only one VM instruction. Not surprisingly, superinstructions give less benefit either in running time or in code size reduction.

The code size figures for the remaining benchmark programs are remarkably similar. The main reason for this is that the statically appearing code for these programs is dominated by library and Gforth system code that is common to all programs. The code size reduction is similar, because we are mostly measuring reductions in the same code. Overall superinstructions give a very significant benefit in code size reduction to code generated by a code-copying JIT compiler. The best strategy gives an average reduction in code size of 17.9% on the Pentium 4 and 20.5% on the PowerPC. Even very small numbers of superinstructions can give significant benefits.

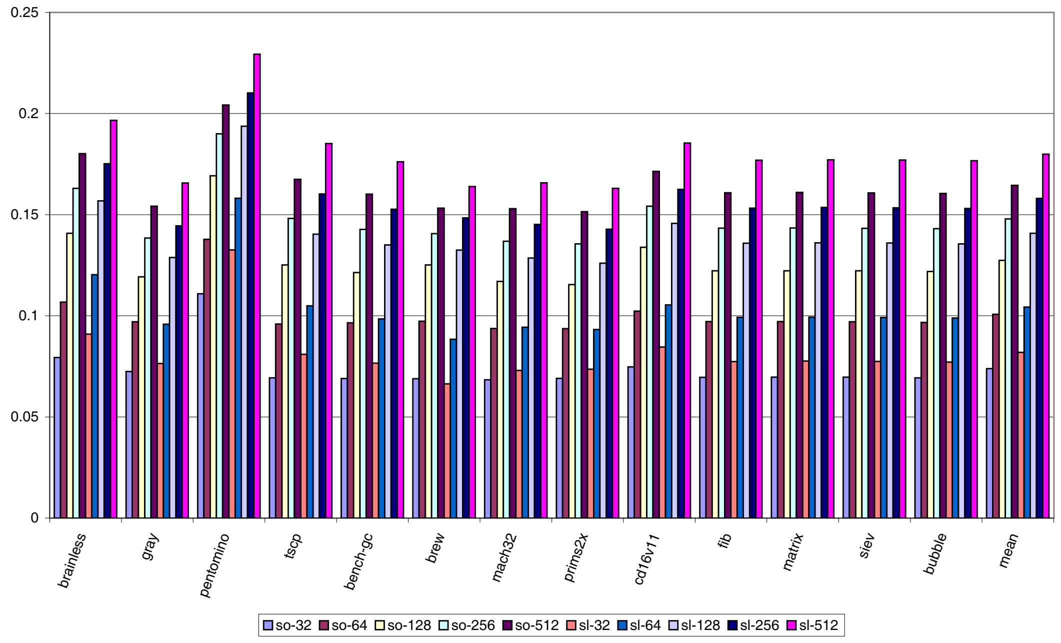

Figure 14. Percentage reduction in the size of the x86 executable code generated for benchmark programs using a varying number of superinstructions, selected based on the combination of static frequency and length (sl), or only static frequency alone (so).

An important question is how the performance of the generated code from a code-copying JIT compiler compares with that of traditional native-code JIT compilers. Although we have shown that we can improve the performance of a simple code-copying JIT compiler by using superinstructions, it is also important that the base version is fast. Otherwise, our optimizations would offer only a relatively small speedup of a very slow system.

To give some indication of relative performance, Figure 16 shows a comparison of running times of our system with two native code Forth JIT compilers running on an AMD Athlon processor. The running times for GForth are for base version without any superinstructions. BigForth is a fast and simple native-code compiler that makes effective use of peephole optimizations. iForth is another simple compiler, with fewer optimizations. Although our base system is significantly slower than either of the two traditional JIT compilers, the difference in performance is not so great as to make our approach unviable.

Another important question is whether we could expect to see similar gains for code-copying JIT compilers for other stack-based VMs. In the case of the Java VM, we believe that even greater speedups and code size reductions are possible, for two reasons. First, Forth basic blocks tend to be very short, limiting the potential of superinstructions. Gregg *et al.* [9] found that 30–40% of basic blocks in

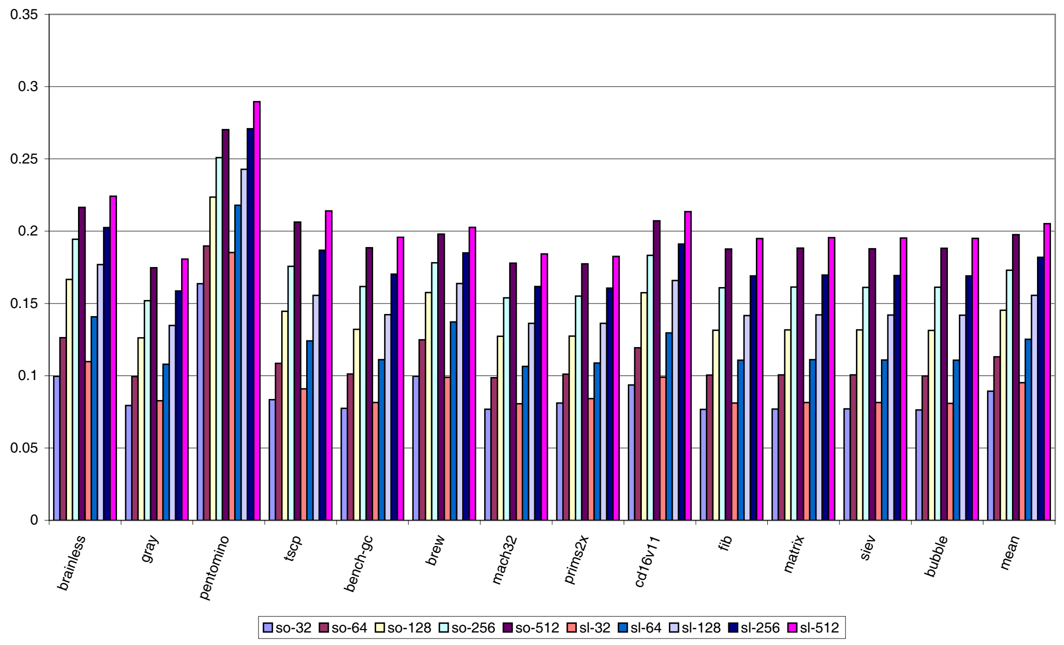

Figure 15. Percentage reduction in the size of the PowerPC executable code generated for benchmark programs using a varying number of superinstructions, selected based on the combination of static frequency and length (sl), or only static frequency alone (so).

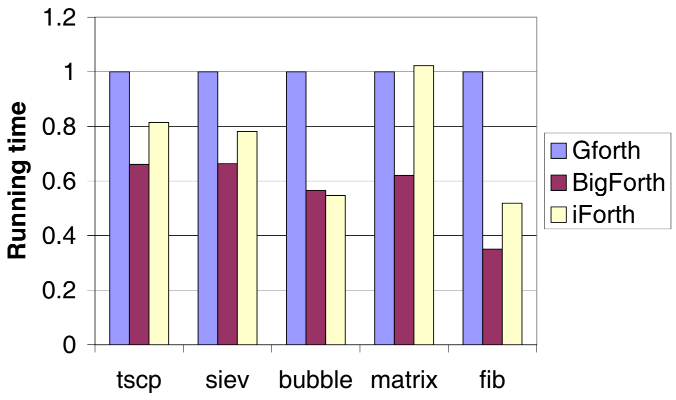

Figure 16. Comparison of running times of our base system (without superinstructions) with two native-code JIT compilers.

Forth programs consist of a single VM instruction and typical basic blocks contain about three VM instructions. In contrast, Java basic blocks usually contain about five or six JVM instructions. Another reason for optimism is that Java compilers typically leave that stack empty at the end of each statement. Thus, basic blocks often consist of well-balanced sequences of VM instructions that have no net effect on the stack. Such sequences are very well optimized by Vmgen.

## 10. RELATED WORK

Piumarta and Riccardi [4] proposed dynamically generating code by copying code fragments from an interpreter. Their work differs from ours in a number of important respects. First, their scheme can be applied only to single basic blocks, whereas we generate code for entire procedures, eliminating many unnecessary indirect branches. Secondly, no attempt is made to choose superinstructions which are better than sequences of their component instructions. Finally, our stack optimizations are entirely new.

Tempo [10] is a run-time specializer that generated C code concatenated sections of C source code. A major difference from our approach is that Tempo specializes particular functions for particular inputs. Furthermore, Tempo uses relocation information from the object file to perform additional optimizations. Although the result is that better machine code is generated, it introduces specific machine dependencies into the process, making the technique more difficult to retarget.

Another approach to creating a compiler from an interpreter is to use partial evaluation to specialize an interpreter for a particular program [11]. However, this research [12,13] used near source-to-source translators. More recently, Beckmann *et al.* [14] have used this approach to specialize a machine emulator, by generating C++ code at run time, and compiling the code dynamically using a traditional C++ compiler.

## 11. CONCLUSIONS

Portable JIT compilers can be created using GNU C extensions to copy executable code. We have presented a novel optimization for such compilers for stack-based machines. Our approach is based on superinstructions, a well known optimization for interpreters. We have developed a small domain-specific language and tool for generating optimized C source code for stack-based VMs. Our generator can automatically create source code for superinstructions which is optimized to reduce the number of memory accesses for stack items, and eliminates redundant stack pointer updates.

We have also presented a novel strategy for choosing superinstructions which are likely to be most useful to a code-copying compiler. Our new approach is based on generating the source code for each potential superinstruction, and compiling this code with the C compiler. Once compiled we measure the length of the executable code for the superinstruction. This gives us an exact measure of the reduction in code size that can be achieved by using that superinstruction, and gives us a reasonable estimate of whether the superinstruction code is likely to execute more quickly.

We have implemented a code-copying JIT compiler and a system of superinstructions in GForth, a product quality implementation of the Forth language. Experimental results show that average speedups of 15.75% are possible on a Pentium 4 and 26.18% on a PowerPC machine. A sufficient number of superinstructions allows code size reductions of an average of 17.9% (Pentium 4) and 20.5% (PowerPC) over a wide range of programs. We believe that even greater gains are possible for other stack-based virtual machines, where basic blocks are larger.

## ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, whose comments greatly improved the quality of this paper.

## REFERENCES

1. Hoogerbrugge J, Augusteijn L, Trum J, van de Wiel R. A code compression system based on pipelined interpreters. *Software—Practice and Experience* 1999; **29**(11):1005–1023.
2. Proebsting TA. Optimizing an ANSI C interpreter with superoperators. *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, San Francisco, CA, 23–25 January 1995. ACM Press: New York, 1995; 322–332.
3. Bell JR. Threaded code. *Communications of the ACM* 1973; **16**(6):370–372.
4. Piumarta I, Riccardi F. Optimizing direct threaded code by selective inlining. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, 17–19 June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):291–300.
5. Driesen K, Hölzle U. Accurate indirect branch prediction. *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, Barcelona, 27 June–1 July 1998. *ACM Computer Architecture News* 1998; **26**(3):167–178.
6. Ertl MA. Stack caching for interpreters. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, CA, 18–21 June 1995. *ACM SIGPLAN Notices* 1995; **30**(6):315–327.
7. Gregg D, Waldron J. Primitive sequences in general purpose Forth programs. *Proceedings of the 18th EuroForth Conference (EuroForth'02)*, Vienna, Austria, September 2002; 24–32.
8. Bell T, Cleary J, Witten I. *Text Compression*. Prentice-Hall: Englewood Cliffs, NJ, 1990.
9. Gregg D, Ertl A, Krall A. Implementation of an efficient Java interpreter. *Proceedings of the 9th High Performance Computing and Networking Conference*, Amsterdam, The Netherlands, June 2001 (*Lecture Notes in Computer Science*, vol. 2110). Springer: Berlin, 2001; 613–620.
10. Noel F, Hornof L, Consel C, Lawall J. Automatic, template-based run-time specialization: Implementation and experimental study. *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98)*, Chicago, IL, 14–16 May 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998. Available at: http://computer.org/proceedings/iccl/8454/8454toc.htm.
11. Jones N, Gomard C, Sestoft P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall: Englewood Cliffs, NJ, 1993.
12. Neumann G. Metaprogrammierung und Prolog. *Internationale Computer-Bibliothek*. Addison-Wesley: Bonn, 1988.
13. Venugopal KS, Manjunath G, Krishnan V. sEc: A portable interpreter optimizing technique for embedded Java virtual machine. *Proceedings of the 2nd USENIX Java Virtual Machine Research and Technology Symposium*, San Francisco, CA, 1–2 August 2002. USENIX Association: Berkeley, CA, 2002.
14. Beckmannn O, Fordham P, Houghton A, Kelly P. A library for explicit dynamic code generation and optimisation in C++. *Proceedings of the 10th International Workshop in Compilers for Parallel Computers*, Amsterdam, The Netherlands, January 2003; 147–156.