vmgen — A Generator of Efficient Virtual Machine Interpreters

M. Anton Ertl[1,*,†], David Gregg[2], Andreas Krall[1], and Bernd Paysan[3]

[1] *Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria*
[2] *Department of Computer Science, Trinity College, Dublin 2, Ireland*
[3] *Stockmannstr. 14, D-81477 München, Germany*

SUMMARY

In a virtual machine interpreter, the code for each virtual machine instruction has similarities to code for other instructions. We present an interpreter generator that takes simple virtual machine instruction descriptions as input and generates C code for processing the instructions in several ways: execution, virtual machine code generation, disassembly, tracing, and profiling. The generator is designed to support efficient interpreters: it supports threaded code, caching the top-of-stack item in a register, combining simple instructions into superinstructions, and other optimizations. We have used the generator to create interpreters for Forth and Java. The resulting interpreters are faster than other interpreters for the same languages and they are typically 2–10 times slower than code produced by native-code compilers. We also present results for the effects of the individual optimizations supported by the generator.
Key Words: interpreter; virtual machine; generator; stack architecture; superinstruction; byte code

1 Introduction

Interpreters are a popular approach for implementing programming languages. Several advantages make interpreters attractive:

- Ease of implementation.
- Portability.
- Fast edit-compile-run cycle.

While other implementation techniques provide one or two of these advantages, only interpreters provide all three.

A popular choice when implementing interpreters is to use a virtual machine (VM), i.e., an intermediate representation with many similarities to real machine

*Correspondence to: M. Anton Ertl, Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria
†E-mail: anton@mips.complang.tuwien.ac.at

code[a]; in particular, VM code consists of a sequence of VM instructions. In such designs the interpretive system is divided into a front end, i.e., a compiler that produces VM code, and a VM interpreter that executes this code. The advantages of this approach are efficiency (the VM is usually designed to be interpreted with minimal interpreter overhead), and a clean interface between modules of the interpretive system. Well-known examples of virtual machines are Java's JVM [LY99], Prolog's WAM [AK91], and Smalltalk's VM [GR83].

Virtual machines are often designed as stack architectures, for two reasons: 1) It is easy to generate stack-based code from most languages; 2) stack-based VM instructions require less decoding overhead than register-based VM instructions and therefore execute faster [Ert95].

1.1 Automation

When creating a VM interpreter, there are many repetitive pieces of code: The code for executing one VM instruction has similarities with code for executing other VM instructions (get arguments, store results, dispatch next instruction); likewise for VM disassembly and VM code generation. Moreover, the code for dealing with one VM instruction is distributed across several places: VM interpreter engine, VM disassembler, and VM code generation support functions. If you want to change or add a VM instruction, typically all of these places have to be updated. These issues suggest generating the code for processing VM instructions from a common VM instruction description.

In this paper we present such a generator: vmgen. From a single, simple VM instruction description file (Section 3) it generates C code for:

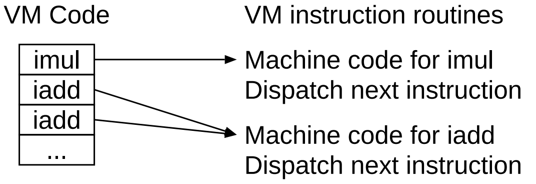
- Executing the instruction (with operand access, debugging support, and instruction dispatch generated automatically; see Section 4.1–4.3).
- Routines for generating VM code (to be used in the front end of the interpretive system; see Section 4.5).
- Disassembling the VM code (useful in debugging the front end; see Section 4.6).
- Profiling VM instruction sequences (Section 4.7).
- Combining instructions into superinstructions (Section 4.8).

Vmgen has special support for stack-based VMs. It has no special support for register-based VMs, but most of its features are also useful when working on such VMs; only stack operand access generation and stack access optimizations are useless for register VMs.

1.2 Efficiency

In addition to reducing the work of writing and maintaining an interpreter, vmgen makes it easy to write fast interpreters by supporting efficient implementation techniques and a number of optimizations.

[a]This is the usual meaning of *virtual machine* in interpreter construction (first item in http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?virtual+machine).




```
GNU C              Alpha assembly
next_inst = *ip;   ldq  s2,0(s1)   ;load next VM instruction
ip++;              addq s1,0x8,s1  ;increment VM instruction pointer
goto *next_inst;   jmp  (s2)       ;jump to next VM instruction
```

Figure 1: Threaded code: VM code representation and instruction dispatch

Some might argue that efficiency is not important in interpreters because they are slow anyway. However, this attitude can lead to an interpreter that is more than a factor of 1000 slower than native code produced by an optimizing compiler [RLV$^+$96], whereas the slowdown for efficient interpreters is only a factor of 10 [HATvdW99]. That is, the difference between a slow and a fast interpreter is larger than the difference between a fast interpreter and native code.

Another argument is that interpreted programs spend much of their time in native-code libraries, so speeding up the interpretive engine will not provide a speedup to these programs. However, the amount of time spent in libraries is not necessarily known at the start of a project, and the prospect of a > 1000 slowdown (if the library does not cover everything and a significant amount of computation has to be performed using simple operations) is daunting. Moreover, having an efficient interpreter increases the number of applications where the library dominates run-time. For example, consider an application that would spend 99% of its time in libraries when compiled with an optimizing native code compiler: with an efficient interpreter it will spend 90% of the time in the library (for an overall slowdown of 1.1 over optimized native code), whereas with an inefficient interpreter it will spend less than 10% of its time in the library (for an overall slowdown of > 10).

Vmgen supports the following techniques and optimizations for writing efficient interpreters:

- Threaded code (Section 1.3).
- Scheduling the dispatch of the next VM instruction (Section 5.1).
- Keeping the top-of-stack in a register (Section 5.2).
- Combining VM instructions into superinstructions (Section 4.3 and 4.8).
- Eliminating stores of unchanged stack items (Section 5.3).
- Improving branch prediction accuracy (Section 5.4).

1.3 Threaded code

Threaded code [Bel73] represents a VM instruction as address of the routine that implements the instruction. In threaded code the code for dispatching the next instruction consists of fetching the VM instruction, jumping to the fetched address, and incrementing the instruction pointer. This technique cannot be implemented in ANSI C, but it can be implemented in GNU C using the labels-as-values extension. Figure 1 shows threaded code and the instruction dispatch sequence.

Another popular technique for implementing VM interpreters represents a VM instruction as an integer (often byte-sized) and uses a C `switch` statement for dispatch.

The advantages of threaded code over switch dispatch are a short, fast instruction dispatch sequence and better branch prediction accuracy on machines with branch target buffers [EG01]. A disadvantage of threaded code compared with bytecode is the larger VM code size.

2 Vmgen overview

Figure 2 shows how the source files for a vmgen-based interpreter are compiled, and the generated files involved. The input to vmgen is a `.vmg` file that contains type and stack definitions and instruction specifications (see Section 3 for details). Vmgen outputs six files containing C code; most of these files must be `#include`d in a wrapper C function[b].

Vmgen comes with an example, and the example's wrapper files can usually be used with few changes; the exception to this rule is `compiler.c`, which contains the front-end of the interpretive system and therefore is quite different for different languages.

If the interpreter is compiled with VM profiling enabled, running the interpreter produces a VM profile in addition to the usual results; this VM profile is useful for selecting which VM instruction sequences should be combined into superinstructions.

3 Input

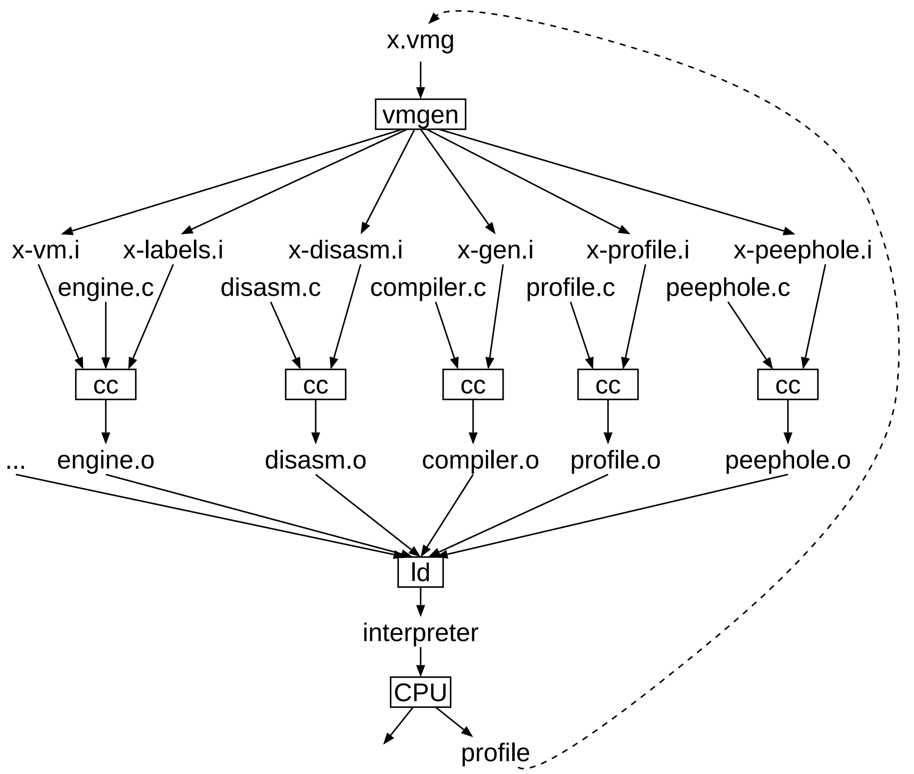
Figure 3 shows a simplified grammar for vmgen's input language. Figure 4 shows a complete (but not very useful) vmgen-style virtual machine specification, explained in the following.

3.1 Simple instruction specifications

A typical example for a simple instruction description is the JVM instruction `iadd`:

```
iadd  ( i1 i2 -- i )
i = i1+i2;
```

[b]We use the extension `.i` (for *include*), because the commonly used extension `.h` (for *header*) indicates declarations and macro definitions.



wrapper	generated file	description
x.vmg		input: VM description (Section 3)
engine.c	x-vm.i	VM instruction execution (Section 4.1–4.3)
engine.c	x-labels.i	VM instruction address array (Section 4.4)
disasm.c	x-disasm.i	VM code disassembler (Section 4.6)
compiler.c	x-gen.i	VM code generation (Section 4.5)
profile.c	x-profile.c	VM instruction sequence profiling (Section 4.7)
peephole.c	x-peephole.c	superinstruction tables (Section 4.8)

Figure 2: Files and processing steps in interpreter generation

Vmgen extracts a lot of information from the first line of the instruction specification, and generates a lot of code from it, whereas the C-code part is just used almost verbatim in one of the generated files.

In particular, the parenthesized part in the first line (the *stack effect*) contains a lot of information: the number of items popped from and pushed on the stacks, their order, their stack, their type, and by what name they are referred to in the C code. In our example, `iadd` pops the two integers `i1` and `i2` from the data stack, executes the C code, and then pushes the integer `i` on the data stack.

The stack effect is responsible for a lot of generated code: for declaring C variables for the stack items, for accessing the arguments and results, for

```
description  -> {simple_inst|super_inst|comment|escape}
simple_inst  -> inst_id ' (' stack_effect ')' newline
                C_code newline newline
stack_effect -> {item_id} '--' {item_id}
super_inst   -> inst_id ' =' {inst_id} newline
comment      -> '\ ' comment_string newline
escape       -> '\E ' (stack_def|stack_prefix|type_prefix) newline
stack_def    -> 'stack' stack_id pointer_id c_type_id
stack_prefix -> stack_id 'stack-prefix' prefix_id
type_prefix  -> 's" ' type_string '"' ('single'|'double') stack_id
                'type-prefix' prefix_id
```

Figure 3: Simplified EBNF grammar for vmgen input

```
\ stack definitions:
\E stack data-stack sp Cell

\ stack prefix definitions:
\E inst-stream  stack-prefix #

\ type prefix definitions:
\E s" int" single data-stack type-prefix i

\ simple instruction definitions:
iadd  ( i1 i2 -- i )
i = i1+i2;

ipush ( #i -- i )

\ superinstruction definitions:
ipush_iadd = ipush iadd
```

Figure 4: A simple virtual machine specification

outputting the arguments and results in traces, and for dealing with immediate arguments in VM code generation and disassembly.

3.2 Special macros

The user-supplied C code can contain a few macros with special meaning to vmgen.

SET_IP This macro sets the VM instruction pointer. It is used for implementing VM branches. It also indicates the end of a VM-code basic block for profiling.

SUPER_END Use this for ending a basic block in an instruction without SET_IP (for profiling, see Section 4.7). For example, our JVM implementation uses SUPER_END in VM instructions that return from the engine.

`TAIL` This macro indicates that the execution of the current VM instruction ends and the next one should be invoked. Using this macro is only necessary if you want to do this in the middle of the user-supplied C code; vmgen automatically appends this code at the end without needing `TAIL` there. Vmgen itself expands `TAIL`.

As an example of the use of these macros, consider a conditional branch:

```
ifeq ( #aTarget i -- )
if ( i == 0 ) {
  SET_IP(aTarget); TAIL;
}
```

The `#` prefix indicates an immediate argument (see Section 3.4). To improve branch prediction accuracy, we use `TAIL` here instead of falling through to the end (see Section 5.4).

3.3 Types

The type of a stack item is specified through its prefix, like in Hungarian notation or Fortran. In our example, all stack items have the prefix `i` that indicates a 32-bit integer. The types and their prefixes are specified at the start of the `.vmg` file, like this:

```
\E s" int" single data-stack type-prefix i
```

The `s" int"` indicates the C type of the prefix (`int`), `single` indicates that this type takes only one slot on the stack (currently vmgen supports types taking one or two slots), `data-stack` is the default stack for stack items of that type (see Section 3.4), and `i` is the name of the prefix. If there are several matching prefixes, the longest one is used.

3.4 Stacks

Vmgen supports virtual machines with several stacks. It also treats the instruction stream like a stack in some respects; this approach is a convenient way to automate the treatment of immediate arguments. Accesses to a non-default stack are specified by a prefix that is not part of the name of the argument. For example, consider the code for the VM instruction `ipush`[c] that pushes a constant on the stack:

```
ipush ( #i -- i )
```

Here `#` is the prefix for the instruction stream[d]. `ipush` just takes the value `i` from the instruction stream and pushes it on the stack.

You define a stack like this:

```
\E stack data-stack sp Cell
```

[c] A generalization of the JVM instructions `bipush` and `sipush`.

[d] inspired by the 6800/6502/68000 immediate addressing mode syntax

`data-stack` is the name of the stack (prefix definitions refer to that name), `sp` is the name of the stack pointer, `Cell` is the C type of a generic stack item (i.e., the type that the stack pointer points to). The instruction stream works somewhat differently from the other stacks and is therefore predefined (stack pointer `IP`, type `Cell`).

You define a stack prefix like this:

```
\E inst-stream  stack-prefix #
```

`inst-stream` is the name of the stack (this stack is predefined), and `#` is its prefix.

The conversion between the generic stack item type and the type of a particular stack item is performed by conversion macros when accessing the stack item (see Section 4.1).

3.5 Superinstructions

A superinstruction performs the work of a sequence of simple instructions. It is defined like this:

```
ipush_iadd = ipush iadd
```

`ipush_iadd` is the name of the superinstruction, `ipush` and `iadd` are the names of the component instructions; there can be arbitrarily many components.

3.6 Virtual register machines

There are two ways to use `vmgen` for defining a register VM:

The direct way is to define instructions that take register numbers as immediate arguments and do not use stacks at all:

```
add ( #isrc1 #isrc2 #idest -- )
reg[idest] = reg[isrc1] + reg[isrc2];
```

The indirect way is to define a hybrid stack/register VM where the only register accesses are transfers between the registers and the stacks; full register instructions are defined as superinstructions:

```
load ( #isrc -- i )
i = reg[isrc];

store ( i #idest -- )
reg[idest] = i;

add = load load iadd store
```

The advantages of the indirect approach are that `vmgen`'s tracing feature produces more useful output (it reports the values loaded from and stored to registers), and that it is easier to deal with instructions specialized for hard-coded registers (e.g., `load_r1`).

Of course, once you are using a hybrid stack/register VM, you might choose to use the stack instead of registers for short-lived intermediate results, and use profiling to decide which sequences should become superinstructions, resulting in a best-of-both-worlds VM.

```
I_iadd: {              /* label */
int i1;                /* declarations of stack items */
int i2;
int i;
NEXT_P0;               /* dispatch next instruction (part 0) */
i1 = vm_Cell2i(sp[1]); /* fetch argument stack items */
i2 = vm_Cell2i(spTOS);
sp += 1;               /* stack pointer updates */
{                      /* user-provided C code */
#line 6 "mini.vmg"
i = i1+i2;
}
NEXT_P1;               /* dispatch next instruction (part 1) */
spTOS = vm_i2Cell(i);  /* store result stack item(s) */
NEXT_P2;               /* dispatch next instruction (part 2) */
}
```

Figure 5: Simplified version of the code generated for the `iadd` VM instruction in the `*-vm.i` file (see text for a description)

4 Output

4.1 Interpreter engine

Figure 5 shows a simplified and commented version of the `*-vm.i` output that `vmgen` generates for the `iadd` VM instruction. It starts with the label of the VM instruction. Then the stack items used by the instruction are declared. `NEXT_P0`, `NEXT_P1`, and `NEXT_P2` are macros for the instruction dispatch sequence. The assignments following `NEXT_P0` are the stack accesses for the arguments of the VM instruction. Then the stack pointer is updated (the stacks grow towards lower addresses). The next piece of code is the C code from the instruction specification. After that, apart from the dispatch code there is only the stack access for the result of the instruction.

The stack accesses to the top-of-stack use `spTOS` instead of `sp[0]` to enable top-of-stack caching (see Section 5.2).

`vm_Cell2i` and `vm_i2Cell` are macros for changing the type of the stack item from the generic type to the type of the actual stack item; they have to be defined in the C program that `#include`s this `*-vm.i` file; we have defined `Cell` to be an integer type and have used casts for most of these macros (an alternative would be to define `Cell` as a union).

This C code looks long and inefficient (and the complete version is even longer), but GCC optimizes it quite well and produces the optimal code shown in Fig. 6 on the Alpha architecture. It also produces optimal code for `iadd` for most architectures we looked at.

4.2 Debugging

A typical C debugger is not very well suited for debugging an interpreter because the C debugger works at a too-low level and does not know anything about the

```
ldl     t0,8(s3)  ;i1 = vm_Cell2i(sp[1]);
ldq     s2,0(s1)  ;load next VM instruction
addq    s3,0x8,s3 ;sp += 1;
addq    s1,0x8,s1 ;increment VM instruction pointer
addl    t0,s4,s4  ;i = i1+i2;
jmp     (s2)      ;jump to next VM instruction
```

Figure 6: Alpha code produced for `iadd`

```
I_iadd:
NAME("iadd")     /* print VM inst. name and some VM registers */
...              /* fetch stack items */
#ifdef VM_DEBUG
if (vm_debug) {
  fputs(" i1=", vm_out); printarg_i(i1);   /* print arguments */
  fputs(" i2=", vm_out); printarg_i(i2);
}
#endif
...              /* user-provided C code */
#ifdef VM_DEBUG
if (vm_debug) {
  fputs(" -- ", vm_out);                   /* print result(s) */
  fputs(" i=", vm_out); printarg_i(i);
  fputc('\n', vm_out);
}
#endif
...              /* store stack items; dispatch */
```

Figure 7: Tracing code generated for the `iadd` VM instruction in the `*-vm.i` file (see text for a description)

interpreted program; e.g., stepping through the interpreter is tedious, and the C debugger does not offer support for stepping through the interpreted program.

Vmgen supports debugging at the VM level in two ways: through a VM disassembler (see Section 4.6) and a trace mechanism (described in this section).

Note that debugging at the VM level is useful for debugging the interpretive system, but not usually for debugging the end-user's program, so the interpreter writer may want to provide additional facilities for debugging end-user programs.

Figure 7 shows the tracing code that we left out of Fig. 5. `NAME` is a macro to output the instruction name and the contents of interesting VM registers (e.g., the instruction pointer and the stack pointers). The trace is written to the file `vm_out` and thus can be directed anywhere without upsetting `stdout` or `stderr`. The user defines the `printarg_` functions and can thus control how the arguments and results are displayed (useful for types like functions that are better displayed symbolically).

The tracing outputs are surrounded by `#ifdef`s to avoid influencing the code quality for the non-debugging version of the interpreter; they are also

```
I_ipush_iadd:
{
Cell _IP0;              /* synthetic names for stack item vars */
Cell _sp0;
Cell _sp1;
NEXT_P0;
_IP0 = vm_Cell2Cell(IPTOS); /* fetch superinstruction arguments */
_sp0 = vm_Cell2Cell(spTOS);
INC_IP(1);                  /* stack pointer update(s) */
/* ipush ( #i -- i ) */      /* component instruction ipush */
{
  int i;                    /* declare ipush stack items */
  i = vm_Cell2i(_IP0);      /* fetch ipush argument */
  {                         /* ipush user-supplied C code */
  }
  _sp1 = vm_i2Cell(i);      /* store ipush result */
}
/* iadd ( i1 i2 -- i ) */    /* component instruction iadd */
{
  int i1;                   /* declare iadd stack items */
  int i2;
  int i;
  i1 = vm_Cell2i(_sp0);     /* fetch iadd arguments */
  i2 = vm_Cell2i(_sp1);
  {                         /* iadd user-supplied C code */
  i = i1+i2;
  }
  _sp0 = vm_i2Cell(i);      /* store iadd result */
}
NEXT_P1;
spTOS = vm_Cell2Cell(_sp0);  /* store superinstruction result */
NEXT_P2;
}
```

Figure 8: Simplified version of the code generated for the `ipush_iadd` superinstruction in the `*-vm.i` file (see text for a description)

surrounded by `if (vm_debug)` to allow switching tracing on and off at run-time (e.g., from the C debugger).

4.3 Superinstructions

Figure 8 shows the code that `vmgen` produces for the superinstruction `ipush_iadd`. It introduces additional variables for all the stack items involved in the superinstruction, loads the arguments into these variables at the start, stores the results from these arguments at the end, and performs combined stack pointer updates at the start (in our example, there is no overall change of `sp`, so only `IP` is updated, using the `INC_IP` macro).

The code generated for each component instruction of a superinstruction is

```
ldq     t0,0(s1)    ;_IP0 = vm_Cell2Cell(IPTOS)
ldq     s2,8(s1)    ;load next VM instruction
addq    s1,0x10,s1 ;increment VM instruction pointer
addl    t0,s4,s4    ;i = i1+i2;
jmp     (s2)        ;jump to next VM instruction
```

Figure 9: Alpha code produced for `ipush_iadd`

similar to the code generated for the same instruction as a simple instruction, but the arguments are fetched from and the results are stored to the stack item variables of the superinstruction, there are no stack pointer updates and no instruction dispatch code.

This approach relies heavily on the optimization capabilities of the C compiler, in particular copy propagation. Fortunately, GCC is quite good at that, as shown in Fig. 9. The code is even shorter than the code for `iadd`, because `sp` is not updated. As you can see, combining instructions into a superinstruction not only optimizes dispatches away, but also stack accesses and stack pointer updates.

The debugging code (not shown) is inserted into the component instructions, not the superinstruction; the advantages of this approach are that the arguments and results are reported with meaningful names and that the output is the same (and, e.g., comparable with `diff`) whether or not instructions are combined into superinstructions; the disadvantage is that the superinstructions are not visible in the debugging output.

4.4 Labels

We implement threaded code using GNU C's labels-as-values feature. One problem with this approach is that the labels are only visible in the function containing the interpreter engine. However, we also need to know these labels in the functions that generate the VM code.

Our solution is to store all the labels in an array, and return the array from a special call to the engine function, so that they can be globally accessed, through the variable `vm_inst`.[e]

Vmgen supports this approach by generating the contents of the array initializer in the file `*-labels.i`. For example, for `iadd` it generates

```
(Label)&&I_iadd,
```

This is the GNU C syntax for the address of the label `I_iadd`.

4.5 VM code generation support

Vmgen generates functions for generating VM code in the file `*-gen.i`. For example, here is the function to generate an `ipush` instruction (`ipush` pushes a constant on the stack; it takes an immediate argument and is therefore a bit more interesting than `iadd`):

[e] Of course, any `goto` to any of these labels must still occur from within the engine function. They are passed outside in order to avoid having to put all the VM code generation code into the engine function.

```
void gen_ipush(Inst **ctp, int i)
{
  gen_inst(ctp, vm_inst[1]);
  genarg_i(ctp, i);
}
```

There is a pointer to the end of the generated code (pointed to by `ctp`), which is incremented by `gen_inst` and `genarg_i`. `vm_inst[1]` contains the label `I_ipush`.

The advantages of providing these functions are: The front end can use a symbolic name instead of having to refer to `vm_inst[1]` etc., increasing readability and maintainability; passing an immediate argument to such a function increases readability; and the type checker of the C compiler can check the number and types of the arguments.

The following example shows how these functions might be used in a front-end written in yacc:

```
expr: num           { gen_ipush(&p, $1); }
    | expr '+' expr { gen_iadd(&p); }
```

4.6 Disassembler

Having a VM disassembler is useful for debugging the front end of the interpretive system. All the information necessary for VM disassembly is present in the instruction descriptions, so vmgen generates the instruction-specific parts automatically:

```
if (ip[0] == vm_inst[1]) {
  fputs("ipush", vm_out);
  fputc(' ', vm_out); printarg_i((int)ip[1]);
  ip += 2;
  goto _endif_;
}
```

This example shows the code generated for disassembling the VM instruction `ipush`. The `if` condition tests whether the current instruction (`ip[0]`) is `ipush` (`vm_inst[1]`). If so, it prints the name of the instruction and its arguments, and sets `ip` to point to the next instruction. Printing the instruction's address can be done in the enclosing loop.

The sequence of `if`s results in a linear search of the existing VM instructions; we chose this approach for its simplicity and because the disassembler is not time-critical.[f]

4.7 Profiling

Vmgen supports profiling at the VM level. The goal is to provide information to the interpreter writer about frequently-occurring (both statically and dynamically) sequences of VM instructions. The interpreter writer can then use this information to select sequences for combining them into superinstructions.

[f]The `if` ends with a `goto _endif_` instead of an `else` because a long `else if` cascade overruns the parser stack of GCC (after about 2000 `else ifs`). We cannot use `switch` instead, because `vm_inst[1]` is not a valid case label.

The approach to profiling taken here is to record how often the VM control flow branches to each VM branch target. For conditional branches, the fall-through path is also counted as a branch target. At the end of the run, the counts of VM basic blocks entered by fall-through (without a conditional branch) are corrected, the basic blocks are disassembled, and their subsequences are output with attached execution frequencies. There are scripts for aggregating this output into totals for static occurences and dynamic execution frequencies, and to process them into superinstruction rules for the `.vmg` file. This branch-target counting approach is quite fast (slowdown factor ≈ 2), so you can profile long-running applications.

The support for profiling comes in several parts:

- The file `profile.c` contains most of the routines used by profiling.
- The code produced for a VM branch (a VM instruction with `SET_IP`) contains a call to `SUPER_END` near the end; so the interpreter writer just needs to define `SUPER_END` as `vm_count_block(IP)` for the profiling version of the interpreter.
- Vmgen generates the file `*-profile.i` which performs the disassembling for `profile.c`.

The `*-profile.i` code for an instruction is similar to the disassembler code:

```
if (ip[0] == vm_inst[1]) {
  add_inst(b, "ipush");
  ip += 2;
  goto _endif_;
}
```

Disassembling a basic block ends if `ip` points to the start of another basic block or if the C code of the instruction contains `SET_IP` or `SUPER_END`.

This profiling approach does not notice basic block boundaries arising from targets of branches that are not taken in the profiling run. However, the front end can register additional basic block boundaries explicitly if that additional precision is desired.

4.8 Peephole Optimization

The current approach to combining instructions into superinstructions is a very simple peephole-optimizing approach: Every invocation of `gen_inst` (see Section 4.5) checks if the new instruction can be combined with the last instruction into a superinstruction; of course, the last instruction can already be a superinstruction.

Instructions must not be combined across VM branch targets (because you cannot branch into the middle of a superinstruction), so accurate reporting of VM branch targets is essential (but usually easy, because the front-end knows branch targets during VM code generation).

Vmgen supports this approach with routines in `peephole.c` and by generating tables in `*-peephole.i` for mapping two instructions to a superinstruction:

```
{    1,    0,    2}, /* ipush_iadd */
```

This is the rule for combining ipush (index 1) and iadd (index 0) into ipush_iadd (index 2).

Peephole optimization is transparent to the human-written code generation code: For example, if there is a call to gen_ipush, followed by a call to gen_iadd, then the code generation support code automatically generates ipush_iadd.

The only additional requirement is to report basic block boundaries arising from branch targets. Once you have done that, you can add or delete superinstructions at will, rebuild the interpreter, and the new superinstructions are used automatically.

5 Optimizations

Vmgen implements a number of optimizations and supports a few others. The optimizations performed when combining instructions into superinstructions have already been described in Section 4.3 and 4.8.

5.1 Scheduling and prefetching

If vmgen were to generate a single NEXT (dispatch next VM instruction) macro invocation at the end of the code for each instruction, GCC might not be able to schedule the NEXT code to be processed in parallel with the rest of the code for the instruction, e.g., if the rest of the code ends with a basic block boundary or with a store (potential aliasing with the load at the start of the dispatch code[g]).

Therefore, vmgen generates three macro invocations for dispatch (NEXT_P0, NEXT_P1, NEXT_P2) and distributes them through the code for an instruction. There are various ways to define these macros (and some related ones, e.g., ADD_IP and SET_IP) to optimize for the specific properties of the architectures and microarchitectures, e.g.: number of registers, the latency between the VM instruction load and the dispatch jump, and autoincrement addressing mode. This scheme even allows prefetching the next-but-one VM instruction; Gforth (an interpreter built with vmgen) uses this on the PowerPC architecture to good advantage (about 20% speedup, see Section 7.3).

Figure 6 shows an example of well-scheduled code: the code for dispatching the next instruction is interleaved with the other code.

Vmgen also supports scheduling in other ways: e.g., the stack pointer updates are between the loads from the stack and the user-provided C code to fill the latency of the loads; this means that the C code also fills the update-to-store latency (address-generation interlock) that some processors have.

5.2 Top-of-stack caching

Vmgen supports keeping the top-of-stack item (TOS) of each stack in a register (i.e., at the C level, in a local variable). This optimization has the following benefits:

- It reduces the number of loads from and stores to a stack (by one each) of every VM instruction that takes one or more arguments and produces

[g] The restrict feature of ISO C'99 could be used to avoid aliasing, but was not available when we originally encountered this problem in 1992.

one or more results on that stack. This halves the number of data-stack memory accesses in Gforth [Ert95].

- Many of the remaining loads and stores are placed further away from the instructions that use or produce the result, supporting more instruction-level parallelism; this is especially important for floating-point operations on in-order execution CPUs.

The downside of this optimization is that it requires an additional register, possibly spilling a different VM register into memory. Still, we see an overall speedup for Gforth even on the register-starved IA32 (Pentium, Athlon) architecture.

`Vmgen` performs this optimization by replacing `[0]` with `TOS` when referencing stack items. In addition, there are the following cases to consider:

- If a VM instruction takes no arguments from a stack but produces results, it has to flush (store) the TOS for that stack to memory at the start of the VM instruction; that TOS will be refilled from the results at the end.
- If a VM instruction produces no results on a stack but takes arguments, it has to refill (load) the TOS for that stack from memory at the end of the VM instruction.
- However, if the VM instruction does not access the stack at all, the TOS need not be spilled and refilled.

You can see the benefits of these optimizations in Fig. 6 (only one memory access for three stack accesses) and Fig. 9 (no stack memory accesses remaining).

As long as all stack accesses are managed by `vmgen`, this optimization is transparent to the interpreter writer (apart from the need to define a few macros, e.g., `spTOS`).

However, sometimes an interpreter needs to access stack items in memory (e.g., JVM locals), or set the stack pointer (e.g., JVM return instructions).

If the VM contains such instructions, the user has two options:

- Disable top-of-stack caching.
- Flush the top-of-stack cache explicitly before the stack access or stack pointer change, and reload it afterwards.

5.3 Eliminating stack stores

For VM instructions like

```
dup ( i -- i i )
```

naively generated code would store `i` twice, one store (for the bottom element) into the memory location it was loaded from. This store is redundant, but the C compiler does not reliably optimize it away.

Therefore, `vmgen` optimizes it away at the C code level. It uses the name of the stack item as an indication: if a result stack item has the same name as the argument stack item coming from that location, `vmgen` suppresses the store.

Vmgen requires that the user-supplied C code does not change argument stack items, otherwise this optimization would be incorrect. Currently this requirement is not checked, but it could be enforced by declaring the argument stack item variables `const` and using an initialization instead of the assignment to the variable.

There are two complications, however: If top-of-stack caching is enabled, the bottom-element store in the `dup` example is no longer redundant, because `i` comes from the TOS register, and the store is to memory (conversely, the top store is to the TOS register and GCC optimizes it away); so, in the example above vmgen generates code for conditionally compiling the store.

The other complication is that store optimization is hard to do for superinstructions, because it would require tracking the stack items through the components. Therefore, vmgen currently does not perform the store optimization for superinstructions.

5.4 Branch prediction

Mispredictions of indirect branches are a major component of the run-time of efficient interpreters [EG01]. Many current processors use a branch target buffer (BTB) to predict indirect branches, i.e., they predict that the target address of a particular indirect branch will be the same as on the last execution of the branch.

So, ideally, we should use a different indirect branch if the next VM instruction is different from the last time an indirect branch is executed. On a general scope, this means that each VM instruction should have a separate indirect branch (our threaded-code dispatch macros ensure this), and that each VM instruction (or superinstruction) should occur at most once in an inner loop (frequently-used VM instructions are a problem here, but superinstructions can mitigate that).

For conditional branch VM instructions it is likely that the two possible next VM instructions are different, so it is a good idea to use different indirect branches for them. Vmgen supports this optimization with the macro `TAIL` (see Section 3.2). Vmgen expands this macro into the whole end part of the VM instruction (or superinstruction), including the code for storing the stack items into the appropriate memory locations (and top-of-stack register(s)).

6 Experience

We have used vmgen to implement two interpreters:

Gforth is a portable product-quality interpretive implementation of Forth [Ert93]. Forth is a stack-based language, and Gforth has three programmer-visible stacks (data stack, return-stack, and floating-point stack); most of the VM instructions are directly used as Forth words. The Gforth-0.5.0 VM specification has 2340 lines, contains 319 VM instructions, and vmgen generates 15466 lines of C from this (with no superinstructions). The Gforth project started in 1992 and Gforth has been distributed as a GNU package since 1996. Vmgen has been used from the start and proved to be very useful, not just because it allowed adding new VM instructions with little effort, but also because it made it easy to generate additional output formats that were not envisioned originally (e.g., a TAGS file supporting fast navigation of the VM description from Emacs).

The other interpreter we implemented is a threaded-code variant of the *Cacao* JVM JIT compiler [GEK01]. The goals of this project are to see how useful vmgen is for other interpreters than Gforth, to add any missing functionality, and to build a high-performance interpreter for the JVM. In order to achieve the last goal, we decided to translate the original bytecode into threaded code for a JVM-like VM in JIT-compiler fashion. This avoids the performance penalties of bytecode interpretation (in particular for immediate operand access), and allows having more than 256 VM instructions (including superinstructions). The manually written VM specification has 1089 lines and contains 156 VM instructions (we did not do a 1:1 mapping of the JVM instructions to our VM); in addition, there are 113 VM instructions for JNI functions that are generated automatically; vmgen generates 14623 lines of C code from this (with no superinstructions).

Vmgen works well for threaded-code Cacao; we added a few new features for this project that were not necessary for Gforth (e.g., generating a VM disassembler and VM code generation support), but nothing unexpected. Actually the use of vmgen in Cacao is more straightforward than in Gforth.[h] Using vmgen for a byte-code interpreter should be possible; however, because vmgen currently supports only types with up to two stack slots, dealing with immediate operands with more than two bytes (coming from the instruction stream "stack") would currently require manual construction of the operands out of smaller fragments.

The presence of superinstructions had an interesting effect when we wrote threaded-code Cacao: In several cases where we considered introducing a new VM instruction, we just used a sequence of already existing VM instructions, because the new VM instruction would be introduced automatically if the sequence was executed frequently enough. So, superinstructions are not just a run-time optimization, but they also help optimizing programming time.

Vmgen needs a few seconds to run. However, compiling the resulting files can take several minutes even on fast machines, because some of the functions to be compiled are huge after including the generated code (e.g., 135000 lines for the engine function of Cacao with 1069 superinstructions). The main practical limit, however, is memory during compilation: Compiling Gforth with 1300 superinstructions requires about 200MB on a 32-bit machine and about 350MB on a 64-bit machine. Compiling Cacao with 2400 superinstructions requires about 800MB on a 64-bit machine (and severely thrashes on a 512MB machine).

Historically, the root of vmgen is the experience gained in writing a Prolog interpreter based on the WAM (Anton Ertl, Thomas Graf, Andreas Krall 1990) and a 4GL interpreter based on a stack-based VM (Anton Ertl 1991); both projects were done at DMS Decision Management Systems GmbH, Vienna. So when the Gforth project started in 1992, we knew the repetitive and automatable parts of writing an interpreter, and wrote a generator for it, first in Emacs Lisp, with a rewrite in Forth in 1994 (the Elisp version was Emacs-version-dependent, slow, and hard to modify); the current version consists mainly of a 1246-line Forth program. The early experiences also influenced the modifications that we performed when we generalized the generator tailored for Gforth's

[h]Gforth requires a few additional twists because its basic implementation model is indirect threaded code [Dew75]; the code that deals with that additional complication is small enough that we did not create automatic support for it.

needs into a general VM interpreter generator.

7 Performance

In this section, we present basic performance data and evaluate the effect of the various optimizations. Many of the optimizations have been evaluated in earlier papers; the contribution of this section is in a more detailed evaluation and in measurements on modern hardware (branch prediction hardware makes a big difference). Also, the earlier measurements were performed in environments that are different from the environment we are using (i.e., different VMs, different dispatch method, different hardware), so we performed these measurements to validate the earlier results, and to quantify the effects of the optimizations on interpreters generated with `vmgen` on modern hardware.

7.1 Benchmarks and environment

The Forth benchmarks are:

sieve, bubble, matrix, fib Small integer benchmarks.

brainless (0.0.0; on Alphas 0.0.2) A chess program written by David Kühling (3000 lines).

brew (0.03z9) Evolutionary programming simulation, written by Robert Epprecht (18000 lines); we did not run this benchmark on the Alphas (it is not 64-bit clean).

pentomino A puzzle solver, written by Bruce Hoyt (500 lines).

The JVM benchmarks are:

javac The JDK 1.0 Java compiler compiling JavaLex.

db The SPEC JVM98 benchmark _209_db.[i]

sieve A small integer benchmark.

suml A tiny benchmark that does nothing but increment long variables. It is designed to maximize the speed difference between native-code Cacao and interpreters; e.g., it uses longs because the JVM requires more VM instructions for incrementing a long than an int (there is no long version of `iinc`). The value of this benchmark is in providing worst-case performance numbers for interpreters rather than realistic numbers.

Table 1 shows the environments we used for most benchmark runs (exceptions are noted where they occur). On all machines we used GCC 2.95 for compiling C code.

The baseline interpreters we use in the following sections have all optimizations applied except peephole optimization for using superinstructions; we did not apply superinstructions in the baseline because we are still experimenting

[i]Bugs in Cacao (already present in the original native-code version) prevented running more of SPEC JVM98.

CPU	Arch.	Clock	System	OS
Athlon	IA32	800MHz	Abit KT7	Linux-2.4.0, glibc-2.1
Pentium III	IA32	750MHz	Asus CUBX	Linux-2.2.14, glibc-2.1
PPC 7400	PowerPC	450MHz	PowerMac G4	Linux-2.2.9, glibc-2.1
PPC 604e	PowerPC	200MHz	PowerMac 7500	Linux-2.2.9, glibc-2.1
21064a	Alpha	300MHz	AlphaPC64	Linux-2.2.13, glibc-2.0
21164a	Alpha	600MHz	PC164LX	Linux-2.2.13, glibc-2.0
21264	Alpha	500MHz	XP 1000	Linux-2.2.14, glibc-2.0

Table 1: Hardware and OS used for benchmarking

Athlon	sieve	bubble	matrix	fib	brainless	brew	pent.
BigForth 2.0.0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Gforth	2.89	3.11	8.68	5.59	3.97	1.81	6.81
Win32Forth 4.2	4.63	4.68	11.23	8.09	4.47		13.83
PFE 0.30.90	14.97	14.94	23.95	23.59	8.20	3.74	24.51

21064a	javac	db	siev	suml
Cacao native	1.00	1.00	1.00	1.00
Cacao int	2.12	2.29	10.16	39.91
DEC JVM 1.1.4 native	13.14	23.65	2.04	3.85
DEC JVM 1.1.4 int	15.80	27.50	22.51	104.88
OSF JVM 1.0.1 int			29.06	156.42

21164a	javac	db	siev	suml
Cacao native	1.00	1.00	1.00	1.00
Cacao int	2.27	2.10	17.54	25.08
DEC JVM 1.3.1 native		5.01	2.52	3.23
Kaffe 1.0.5 int		19.57	63.47	93.21

Table 2: Relative user times of running benchmarks with Gforth, Cacao int and their competitors (smaller is better)

with the selection of the right superinstructions, and because superinstructions increase the chances of I-cache conflicts, which can cause large performance variations that may hide other performance effects.

On IA32 we compiled Gforth (all versions we compare) with explicit register allocation of the most important VM registers, because otherwise GCC spills some of them to memory, resulting in significant slowdown (factor 1.5–2 on the Athlon)).

7.2 Basic performance

This section compares the performance of interpreters generated with vmgen with the performance of other systems.

Table 2 compares the performance of the interpreters generated with vmgen (Gforth and Cacao int) with other language implementations on the same hardware. The results are execution times relative to the fastest implementation (i.e., smaller is better).

BigForth is a relatively simple Forth native code compiler (it uses macro expansion and peephole optimization for code generation) [Pay91]; Gforth is 1.81–8.68 times slower on the Athlon; on the Pentium III the factors are similar (2.36–7.96). The small slowdown on brew is probably caused by library functions that are implemented more efficiently in Gforth (with the right superinstructions Gforth beats BigForth on brew).

Win32Forth is a Forth interpreter written in assembly language. We ran it on Windows 95 and measured elapsed time.[j] Why is Win32Forth slower (factor 1.13–1.60 excluding pentomino) than Gforth? The two main reasons are: Win32Forth uses a position-independent image format, and relocates at run-time; this requires one additional computation on every memory access into the image; Gforth avoids this overhead by relocating at load time. And Win32Forth uses indirect threaded code, i.e., one indirection more per VM instruction dispatch (in Gforth the slowdown caused by indirect threaded code is 1.13–1.4). Gforth's speedup over Win32Forth shows that it is possible to create fast interpreters with `vmgen`.

PFE is a Forth interpreter written in C; execution speed is only a secondary goal in its development. Still, it is one of the faster C-based Forth interpreters (at least when using GNU C's global register variables (configure option `--with-regs`), as we have done). Gforth's speedup over PFE and other C-based interpreters shows that the speedup between different interpreters can be larger than the speedup from a fast interpreter to native code, so you cannot generalize from the performance of one interpreter to interpreters in general.

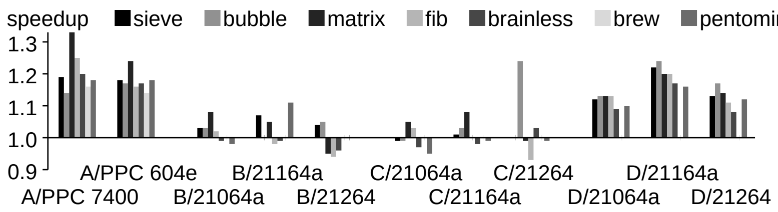
Cacao native is a JVM JIT compiler that allocates locals and stack items to registers and also performs loop optimizations (affects only the sieve results). It shares code like the garbage collector and synchronization with Cacao int. On the small benchmarks, the slowdown of Cacao int is more than a factor of 10 (with a worst case slow-down of 40), but on the larger ones the slowdown is a factor of about 2. One important reason for this is that (according to gprof) for these benchmarks Cacao int spends about 30% of its time in routines like synchronization or garbage collection that are the same speed on both implementations; so even with an infinite speedup on the rest Cacao native could get only a speedup of 3.33.

We ran the DEC JVM 1.1.4 and OSF JVM under Digital Unix 4.0F. The DEC JVM native (JIT) versions beat Cacao int by a lot on the small benchmarks, but are slower than Cacao int on the large benchmarks. For the DEC JVM 1.1.4 we know that this is due to a slow implementation of synchronization [KP98]. The moral of these results is that speeding up run-time system routines like synchronization can have more impact (and probably costs less effort) than implementing a JIT compiler to native-code.

The other interpretive JVM implementations are slower than Cacao int by more than a factor of two on all benchmarks; we believe that this speed advantage is due to the threaded code implementation and the other performance features that Cacao int derives from `vmgen`.

In conclusion, even without superinstructions the performance of interpreters generated with `vmgen` is very competitive relative to other interpreters, and the slowdown over native-code compilers is limited.

[j] For Gforth under Linux the elapsed time was close to the user time, except for the pentomino benchmark, which writes a lot to the terminal; so we expect that these timings (except pentomino) are comparable to the other times.



A	speedup of prefetching (PowerPC default)	over no scheduling support
B	speedup of early fetch (Alpha default)	over prefetching
C	speedup of early fetch (Alpha default)	over no scheduling support
D	speedup of early fetch (no TOS caching)	over no scheduling support (no TOS caching)

Figure 10: Comparing several dispatch code scheduling variants

7.3 Scheduling and prefetching

Gforth defines the macros `NEXT_P0` etc. differently for different architectures, mainly to support the C compiler in scheduling and code selection (e.g., autoincrement addressing modes).

In particular: on the Alpha architecture we use an early fetch of the next VM instruction (in `NEXT_P0`) to support filling the delay slots of that load instruction; on Power/PowerPC Gforth pre-fetches the next-but-one instruction in `NEXT_P1`, because these CPUs usually have a particularly long load-to-jump latency (7 cycles on the PPC 604). We do not schedule the instruction fetch early on IA32, because that would increase register pressure and cause more spilling.

Figure 10 shows the results of our evaluations of different scheduling strategies on Alphas and PowerPCs.

On the PowerPCs prefetching gives a good speedup (≈ 1.2) over no scheduling, because of the long load-to-jump latencies. We did not compare early fetching with no scheduling on the PowerPC, but we expect the results would be similar to the results on the Alphas.

On the Alphas, there is little difference between the three scheduling strategies.

The small difference between early fetching and prefetching (case B in Fig. 10) is easily explained by the fact that the Alphas have a relatively short load-to-jump latency, so prefetching does not help often or much; and they have enough resources that the additional move necessary with prefetching hurts little or not at all.

But we were surprised by the small difference between early fetch and no scheduling on the Alpha (case C in Fig. 10). Looking at the machine code, we found that GCC managed to schedule the load well ahead of the jump by itself in many cases, because in many VM instructions there is no store or basic block boundary that would impede scheduling.

However, when we disabled top-of-stack caching in both variants (case D in Fig. 10), using our scheduling support provided significant speedups over no scheduling support, because then many VM instructions perform a store near the end. The significance of this experiment is that in other VMs stores or basic block boundaries near the end of the VM instruction might be more frequent than in our baseline Gforth, so explicit early fetching may have more value in

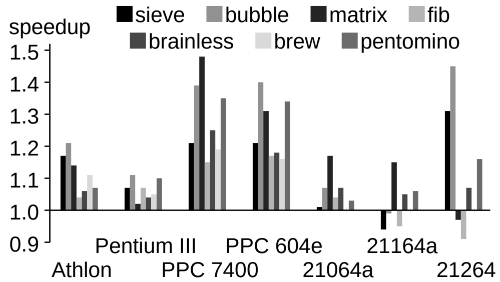


Figure 11: Speedup of the base system over a version without top-of-stack caching

general than is apparent from the results for case C.

In conclusion, for best performance the dispatch macros have to be defined in a processor-specific way; fortunately, the interpreter writer can use Gforth's macros as a starting point (e.g., we did so for Cacao int).

7.4 Top-of-stack caching

To evaluate the effect of top-of-stack caching, we compiled Gforth on all machines without top-of-stack caching (this requires just defining the preprocessor symbol `USE_NO_TOS`) and compared the result to the base system. You find the results in Fig. 11.

On the IA32 processors (in particular the Athlon) top-of-stack caching provides a surprisingly large speedup, given that this architecture is register-starved and that the optimization increases register pressure; looking at the machine code of some frequently-executed VM instructions, we did not see additional spill code.

On the PowerPCs the speedup is even larger, as we expected on a machine with lots of registers.

To our surprise, the speedup of top-of-stack caching on the Alphas (in particular, the 21164a) is relatively small, and for some benchmarks there is even a slowdown. Apparently these processors have enough resources to process the additional loads and stores incurred without top-of-stack caching; and apparently the additional latencies are not a problem or masked by parallel processing with other data (e.g., VM instruction fetch). We explain the slowdowns with non-monotonic effects in instruction issuing (i.e., inserting an instruction can cause a speedup); this explanation is supported by the fact that the first three benchmarks show a slowdown on the 21164a and a large speedup on the 21264 or vice versa.

In conclusion, top-of-stack caching provides a performance benefit on most processors, but the size of the benefit is hard to predict.

7.5 Eliminating stack stores

We evaluated the store elimination optimization by disabling it in `vmgen`, building a version of Gforth with these redundant stores, and measuring it on the Athlon.

Among our benchmarks only sieve and bubble profit significantly ($\approx 10\%$) from store elimination, because they use VM instructions in their inner loops that profit from this optimization. We see little difference for the other benchmarks.

Using the Athlon's performance-monitoring counters, we see the following: for the three larger benchmarks this optimization reduces the number of instructions executed on the Athlon by 0.6%–0.8%, but on brew and brainless there is a small slowdown, caused by a higher number of branch mispredictions (probably because different code placement leads to different conflicts in the BTB).

In conclusion, this optimization provides only minor benefits.

7.6 Branch prediction

We evaluated the effect of using different indirect jumps for the different outcomes of VM conditional branches by comparing (on the Athlon) the base Gforth system with a version that was built from a variant of the `.vmg` file where we eliminated all uses of the `TAIL` macro.

For the small benchmarks we see speedups of 2%–9% from using `TAIL`. For the large benchmarks we see only 0%–2% speedup. Looking at the performance counter results for the large benchmarks, the branch prediction results are mixed; apparently the variation in prediction accuracy ($\approx 1\%$) resulting from code placement differences is larger than the the prediction accuracy benefits we get from using `TAIL`.

However, we also see a reduction in the number of executed instructions (0.6%–1.7% for the large benchmarks); looking at the machine code, we see that GCC performs some additional optimizations if we use `TAIL`.

In conclusion, using `TAIL` gives mixed results for branch prediction, but it produces minor additional benefits.

7.7 Superinstructions

For Cacao int we selected superinstructions by profiling javac and db, and then making superinstructions for all VM instruction sequences up to a given length that occured in the run; with two training programs and three maximum lengths (2, 3, 4) this provided six versions of Cacao with superinstructions. Similarly, for Gforth we used brainless and brew for training runs; however, we had less memory for compiling the result and the training runs produced more unique sequences, so we reduced the number of resulting superinstructions by throwing out all sequences with a dynamic execution count of less then 10000 in the training run. Figure 12 shows the number of superinstructions generated for each variant.

While we see nice speedups on small benchmarks (up to a factor of 5.46 for matrix on Athlon), we focus on the larger benchmarks in this section, because the small benchmarks are too sensitive to the presence or absence of a few specific superinstructions to be general indicators of performance.

Figure 13 shows how the use of superinstructions affects execution time on different processors; we measured brainless and used the superinstructions derived from brew, but the results for other benchmarks are mostly similar (although the magnitude of the speedups varies).

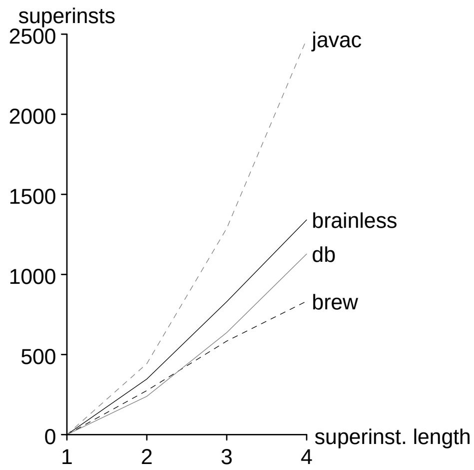


Figure 12: Number of superinstructions generated with different training programs and different maximum sequence lengths

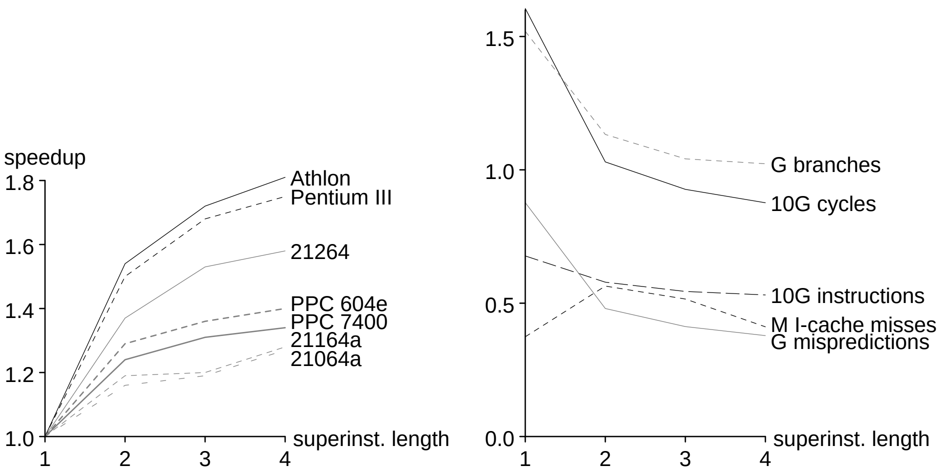


Figure 13: Results for *brainless*, with superinstructions from a *brew* training run. Left: speedup on various CPUs. Right: various performance metrics on the Athlon; 10G, G, and M indicate different scaling factors.

Figure 13 also displays some performance counter results on the Athlon for the same benchmark; the superinstructions reduce the number of native instructions a bit (factor up to 1.28), the number of native branches some more (1.49), and the number of mispredicted branches even more (2.32). The number of cycles on the Athlon (1.83) is strongly influenced by the number of mispredicted branches, because every misprediction costs about 12 cycles, and there are only 7.72–14.03 instructions per mispredicted branch; spending the majority of the cycles on mispredicted branches also explains the relatively low number of instructions per cycle (0.42–0.61) on this triple-issue machine. Most of the branches in Gforth are indirect branches for dispatching the next VM instruc-

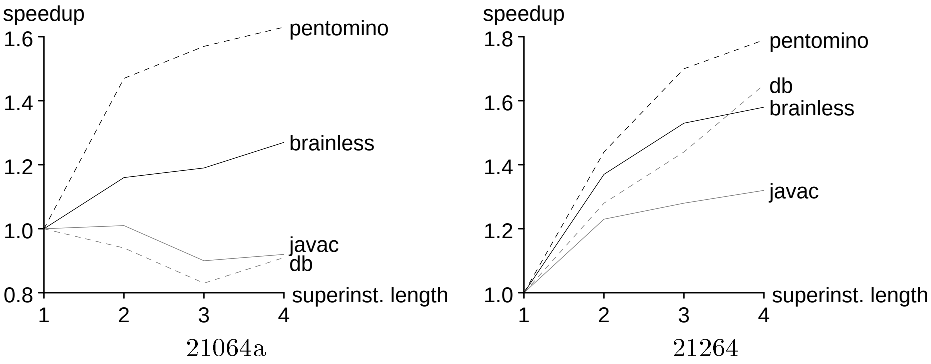


Figure 14: Speedup achieved on 21064a and 21264 by using superinstructions consisting of simple instructions with varying maximal length (trained using *brew* or *db*)

tion, and they are responsible for the high misprediction rate (58%–37%); mispredictions are caused by several occurences of a VM instruction in the working set, and superinstructions reduce that. The number of I-cache misses is low and its performance effect negligible; given that all the engines used fit in the Athlon's I-cache, this is not surprising.

These performance counter results also help explain the differences between the various processors. The Athlon, Pentium III, and 21264 have BTB-like indirect branch predictors and mispredict penalties of 10–12 cycles, and they profit much from the branch prediction improvements, so they see the greatest speedups from superinstructions. They also have a large I-Cache or (Pentium III) a fast L2 cache. The other processors do not predict indirect branches and profit mainly from the reduction in indirect branches. On the 21064a and 21164a the small, direct-mapped I-caches may reduce the benefit of superinstructions.

Figure 14 shows the effect of superinstructions on various large benchmarks. The 21064a results for the JVM benchmarks show slowdowns from using superinstructions.

One explanation for these slowdowns is I-cache misses: the 21064a uses a 16KB direct-mapped I-cache backed by a slow L2 cache (60ns load-to-load latency in a pointer-chasing benchmark). We used the performance counters of the 21064a (under Digital Unix 4.0F) to investigate the I-cache misses (see Fig. 15[k]): javac and db do indeed spend significantly more time in I-cache misses than pentomino and brainless. However, in db they spend even more time in D-cache misses. Together these results explain the slowdowns and the missing speedups. We believe that the additional D-cache misses for db are conflict misses caused by different placement of the VM code (VM code is smaller with superinstructions, so capacity misses should be reduced).

You can see the interpreter sizes in Fig. 15 (the interpreters used for measuring I-cache misses are *gforth brew* and *cacao db*). The base Cacao interpreter engine (without support functions) almost fits in the 21064a's I-cache, and the misses in the base version mostly come from conflicts with or between support-

[k]Memory constraints prohibited compiling larger interpreters on Digital Unix.

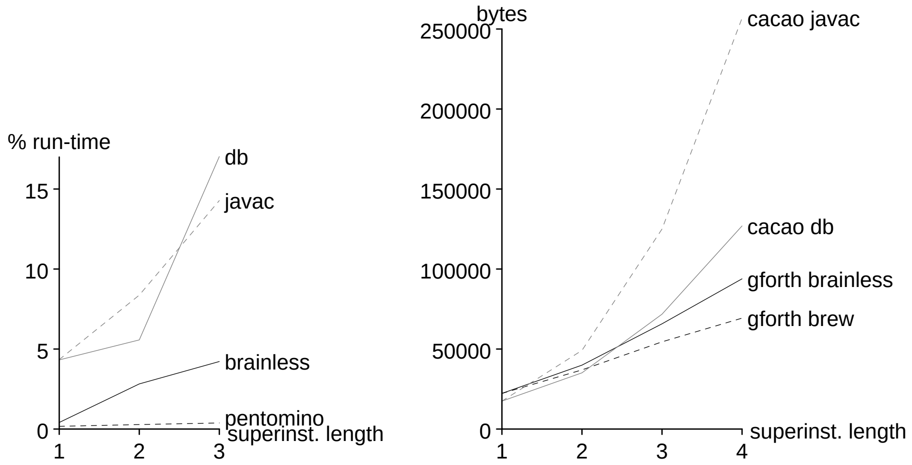


Figure 15: Left: Time spent in I-cache misses on the 21064a (running on interpreters trained with *brew* or *db*). Right: interpreter engine size (without supporting functions) on Alpha.

ing functions like garbage collection or synchronization (only 36% of the I-cache misses for javac and 8.5% for db are in the threaded-code interpreter); however, for the larger interpreters, the majority of the misses are in the threaded-code interpreter.

However, large threaded-code interpreters do not necessarily cause I-cache thrashing: e.g., Gforth brew with max. superinstruction length 3 is more than three times as large as the 21064a's I-cache, but brainless and pentomino do not thrash the I-cache when running on this interpreter. These differences may be caused by locality differences in the interpreted programs.

We also investigated if the different speedup behaviour of the Forth and the JVM benchmarks was caused by the differences in selecting superinstructions for Gforth and Cacao: We selected the Cacao superinstructions like we had selected the Gforth superinstructions (i.e., only sequences with more than 10000 dynamic executions). While we saw an improvement, there are still slowdowns and hardly any speedup for the Java benchmarks on the 21064a, and there is still a large difference compared with the Gforth speedups.

It is also interesting to compare the effect of training on other programs or with other input data to the perfect training of using the measured run as training run, to see what we can gain at the most by improving superinstruction selection (see Fig. 16). We see that even when training with just one large program, we get surprisingly close to the optimum.

In conclusion, superinstructions provide a large optimization potential, especially on processors with BTB-like branch predictors.

8 Related work

There are many generators for producing the front end of compilers and interpretive systems (e.g., `yacc`), and there are code generator generators for producing

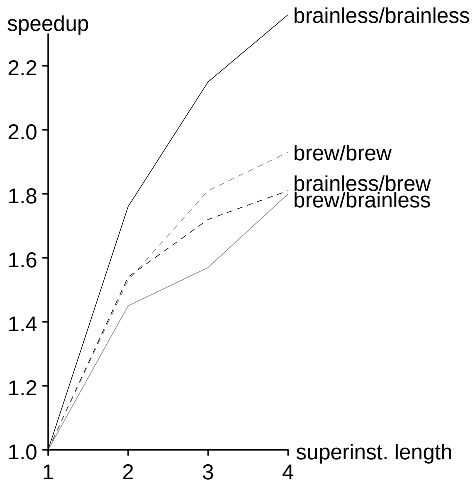


Figure 16: Speedup achieved on Athlon for superinstructions selected with different training runs; in x/y, x is the measured run, and y is the training run.

the back end of native-code compilers (e.g., `burg`). However, we know of no other general tool for constructing the back-end of an interpretive system.

There are more ambitious projects than vmgen that take a description of the syntax and semantics of a language as input and produce an interpreter for the language as output (e.g., ULC[1]). The main disadvantage of such systems is that they are restricted to a certain class of languages, whereas vmgen is more flexible, because it can be combined with any front-end (as long as it can call C functions).

The work closest to vmgen are the generators used by the authors of sophisticated VM interpreters; the main difference to these generators is that vmgen has been extended and tested as a general VM interpreter generator instead of being specific to one project.

The Scheme 48 system [KR94] uses an accumulator/stack-based VM. The VM interpreter is specified in the Scheme subset Pre-Scheme and is compiled into C code. The generated C code has some similarities with vmgen's output: e.g., it uses VM-instruction-scope variables as intermediate storage. The input also appears to have some similarities, but is not described in sufficient detail to allow a detailed comparison (the interpreter specification macros are portrayed more as a Pre-Scheme application than a separate package). Apart from being specific to one language implementation, this generator differs from vmgen in not performing implicit top-of-stack caching; instead, the VM uses an accumulator to support explicit stack caching visible at the VM level (an extra push instruction is required to get values from the accumulator to the stack).

The C interpreter `hti` [Pro95] uses a tree-based VM (linearized into a stack-based form) derived from `lcc`'s intermediate representation. The VM interpreter is created using a tree parser generator and can contain superoperators. The VM instructions are specified in a tree grammar; superoperators correspond to non-trivial tree patterns. The main difference from our work is that our generator

[1] `http://www.imm.dtu.dk/~jsm/ulc/`

works for general stack-based VMs (corresponding to DAGs when represented as data-flow graphs); e.g., in the vmgen input you can specify `dup ( i -- i i )`, which cannot be expressed as a tree operator. The differences between superoperators and our superinstructions have the following consequences: superoperators can combine instructions that are not in a sequence (e.g., `iload ... isub`; other trees may intervene); superinstructions can combine VM instructions that are not tree patterns (e.g., the most frequent sequence in our JVM interpreter is `iload iload`). Proebsting reports higher speedups from superoperators than we see from superinstructions; we believe that this is due to the differences in the VMs used (but differences in the benchmarks and the hardware may also play a role), so we cannot draw conclusions about the value of superoperators vs. superinstructions.

The VVM project allows defining VMs through VMlets [FPR98]. VMlets define new VM instructions in terms of existing VM instructions (starting with a base of primitive instructions), like superinstructions in vmgen. The main differences from vmgen are: With vmgen, you can define arbitrary VM instructions in terms of C, whereas you are limited to the predefined primitives in VVM; as a consequence, you have to use vmgen at interpreter-build time, whereas VVM is much more dynamic and allows loading VMlets at run-time.

IL [ASW91] is a low-level language used for writing and porting APL interpreters; it is easier to retarget the IL compiler than to port an APL interpreter written in assembly language; i.e., IL is used as a portable assembly language, like C in vmgen-based interpreters. IL does not automate the tasks that vmgen automates.

Hand-written interpreters usually make extensive use of macros (e.g., Kaffe); this demonstrates that many tasks in writing interpreters are quite repetitive. The advantages of vmgen over a general macro processor are that the users do not need to design their own macro package, and that vmgen does many things (e.g., optimizations) that typical macro packages do not do. However, vmgen only performs tasks that cannot be done easily by a macro processor; so, vmgen defers some tasks to the C macro processor, and in Gforth we use the m4 macro processor to preprocess vmgen input.

Most of the performance-enhancing techniques used by vmgen have been used and published earlier: threaded code and decoding speed [Bel73, Kli81], scheduling and software pipelining the dispatch [Ert93, HA00, HATvdW99], stack caching [Ert93, Ert95] and combining VM instructions [Pro95, PR98, HATvdW99]. New optimizations: the store optimization (Section 5.3), which comes up only in a generator like vmgen (whereas a human programmer would not generate the stores in the first place); and using tail duplication to improve indirect branch prediction (Section 5.4).

There are also other interesting works on interpreters in general [DV90, Kra83], semantic content, virtual machine design and time/space tradeoffs [Pit87], and various optimizations [SC99].

Code generation libraries such as VCODE [Eng96] and GNU Lightning allow the user to generate native code by using a fixed, register-based programming interface. They can be used to implement programming languages and probably offer higher execution speed than vmgen-based interpreters, but also have some shortcomings: lack of portability (only a few targets are supported by each system), a fixed interface (in contrast to defining a VM tailored for the language) and no debugging support.

9 Conclusion

Vmgen is a generator for virtual machine (VM) interpreters. It automates much of the repetitive work in writing an interpreter, and supports writing high-performance interpreters.

It automatically generates code for accessing VM instruction operands and results, for dispatching the next VM instruction, for tracing VM instruction execution, for VM code disassembly, for supporting VM code generation, and for profiling frequent VM instruction sequences (aiding the selection of superinstructions).

Vmgen supports and encourages writing high-performance interpreters by supporting techniques like threaded code, keeping the top-of-stack in a register (speedup 1.2–1.3 on PowerPCs), scheduling the dispatch code to fetch the next VM instruction early (prefetching speedup 1.2 on PowerPCs), having different instances of the dispatch code for better branch prediction, eliminating superfluous stores, and combining VM instructions into superinstructions (speedup 1.8 on Athlon). Even without superinstructions, the interpreters generated with vmgen are faster than competing interpreters and the slowdown over native-code compilers is usually less than a factor of 10 (and often as low as 2).

Vmgen is available through http://www.complang.tuwien.ac.at/anton/vmgen/.

Acknowledgments

The referees, John Aycock and Manfred Brockhaus provided valuable comments on earlier drafts of this paper.

References

[AK91] Hassan Aït-Kaci. The WAM: A (real) tutorial. In *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[ASW91] M. Alfonseca, D. Selby, and R. Wilks. The APL IL interpreter generator. *IBM Systems Journal*, 30(4):490–497, 1991.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Dew75] Robert B.K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[DV90] Eddy H. Debaere and Jan M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.

[EG01] M. Anton Ertl and David Gregg. The behaviour of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001*, pages 403–412. Springer LNCS 2150, 2001.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[FPR98] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *SIGOPS European Workshop*, 1998.

[GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient Java interpreter. In *High-Performance Computing and Networking (HPCN Europe 2001)*, pages 613–620. Springer LNCS 2110, 2001.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HA00] Jan Hoogerbrugge and Lex Augusteijn. Pipelined Java virtual machine interpreters. In *Proceedings of the 9th International Conference on Compiler Construction (CC' 00)*. Springer LNCS, 2000.

[HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software—Practice and Experience*, 29(11):1005–1023, September 1999.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–973, 1981.

[KP98] Andreas Krall and Mark Probst. Monitors and exceptions: How to implement Java efficiently. *Concurrency: Practice and Experience*, 10(11–13):837–850, 1998.

[KR94] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[Kra83] Glen Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[Pay91] Bernd Paysan. Ein optimierender Forth-Compiler. *Vierte Dimension*, 7(3):22–25, September 1991.

[Pit87] Thomas Pittman. Two-level hybrid interpreter/native code execution for combined space-time efficiency. In *Symposium on Interpreters and Interpretive Techniques (SIGPLAN '87)*, pages 150–152, 1987.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 150–159, 1996.

[SC99] Vítor Santos Costa. Optimising bytecode emulation for Prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.