A Language and Tool for Generating Efficient Virtual Machine Interpreters

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

[1] Department of Computer Science, Trinity College, Dublin 2, Ireland.
David.Gregg@cs.tcd.ie
[2] Institut für Computersprachen, TU Wien, A-1040 Wien, Austria.
anton@complang.tuwien.ac.at

Abstract. Stack-based virtual machines (VMs) are a popular representation for implementing programming languages, and for distributing programs in a target neutral form. VMs can be implemented with an interpreter to be simple, portable, quick to start and have low memory requirements. These advantages make VM interpreters especially useful for minority-use and domain-specific languages. VM interpreters contain large amounts of repeated or similar code. Furthermore, interpreter optimisations usually involve a large number of similar changes to many parts of the interpreter source code. We present a domain-specific language for describing the instruction set architecture of a VM. Our generator takes the instruction definition and produces C code for processing the instructions in several ways: execution, VM code generation and optimisation, disassembly, tracing and profiling. The generator can apply optimisations to the generated C code for each VM instruction, and across instructions. Using profile-directed feedback and experimentation the programmer can rapidly optimise the interpreter for new architectures, environments and applications.

1 Introduction

Interpreters are a popular approach for implementing programming languages, especially where simplicity and ease of construction are important. A common choice when implementing interpreters is to use a virtual machine (VM), i.e., an intermediate representation similar to real machine code, which can be interpreted efficiently. Well-known examples of virtual machines are Java's JVM [1], Prolog's WAM [2], and Smalltalk's VM [3].

An unusual feature of VM interpreters when compared with other programs is that they contain large amount of repeated similar code. Writing and maintaining this code can be time consuming and expensive. Furthermore, common approaches to optimising VM interpreters often require similar (but not identical) modifications to large numbers of places in the interpreter's source code. Such optimisations must usually be performed by the programmer on the source code of the interpreter, because compilers do not have enough domain-specific knowledge of interpreters to identify such opportunities. The result is that VM interpreters

usually remain under-optimised because the programming effort and maintenance cost of experimenting with and performing these optimisations would be too high.

An alternative to writing this source code manually is to generate the repeated similar code using a program generator. In this chapter we describe ongoing work on vmIDL (VM Instruction Definition Language), a domain-specific language [4] for describing the instruction set architecture of a VM. Our interpreter generator vmgen accepts a vmIDL specification as input and outputs C source code for an optimised interpreter, as well as code to implement support software such as tracing, profiling, and optimising VM code at the time it is generated. The goal is to automate much of the construction of interpreters, in much the same way that routine tasks in manufacturing of physical goods are routinely automated.

This chapter presents an overview of our research on generating virtual machine generators and the relationship of our work to domain-specific program generation. Our work is not only an example of a domain-specific language and generator. It is also an enabling technology for other DSLs. By automating many routine tasks in building and optimising an interpreter, it allows portable, reasonably efficient implementations of languages to be constructed quickly.

The rest of this chapter is organised as follows. We first explain the advantages of interpreters for implementing minority-use and domain-specific languages (section 2). Section 3 makes the case for automating the construction of VM interpreters. Section 4 introduces vmIDL, a domain-specific language for describing virtual machine instructions. In section 5 we show how vmgen uses the vmIDL specification to generate C code for the major components of an interpreter system. Section 6 describes the three large interpreters that have been built using vmIDL. In section 7 optimisations performed by vmgen are presented. Finally, section 8 describes existing related work on interpreter generators and interpreters.

2 Interpreters and DSLs

Implementing a domain-specific language (DSL) is very different from implementing a widely-used general-purpose language [5]. General-purpose languages such as Fortran, C, and Java are typically rich in features, Turing complete and often have a relatively complicated syntax. Such languages usually have a large base of users, which makes it economical for sophisticated programming environments to be developed for such languages. For example, languages such as C++ have highly-optimising compilers, special syntax support in editors such Emacs, debuggers and even automatic code restructuring tools.

In contrast, domain-specific languages are smaller and simpler [6]. There are special purpose constructs to enable domain-specific optimisations or code generation. Many DSLs have a small user base. Thus, it is not economical to invest in developing sophisticated compilers and development environments. The

cost of developing, maintaining and porting implementations of the language is very significant compared to the potential user base.

Interpreters have many advantages for implementing domain-specific and minority-use languages [7, 8]. First, interpreters are relatively small, simple programs. Simplicity makes them more reliable, quicker to construct and easier to maintain. Debugging an interpreter is simpler than debugging a compiler, largely because interpreters are usually much smaller programs than compilers. Second, they can be constructed to be trivially portable to new architectures. An interpreter written in a high-level language can be rapidly moved to a new architecture, reducing time to market. There are also significant advantages in different target versions of the interpreter being compiled from the same source code. The various ports are likely to be more reliable, since the same piece of source code is being run and tested on many different architectures. A single version of the source code is also significantly cheaper to maintain. Interpreters allow a fast edit-compile-run cycle, which can be very useful for explorative programming and interactive debugging. Although just in time compilers offer similar advantages, they are much more complicated, and thus expensive to construct. Finally, interpreters require much less memory than compilers, allowing them to be deployed in environments with very limited memory, a useful feature for embedded systems.

These advantages, especially simplicity and portability, have made interpreters very popular for minority-use languages, whose use is usually restricted to a limited domain. For example, Python has always been implemented using an interpreter, which has allowed a non-profit organization to implement and maintain one of the most widely ported languages available. Similarly, most implementations of Forth and Perl are based on interpreters, as are all implementations of Ruby, Logo, `sed` and `awk`. Interpreters allow a language implementation to be constructed, maintained and ported much more cheaply than using a compiler.

3 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). Similarly, when we optimise the source code for an interpreter, we apply similar transformations to the code for each VM instruction. Applying those optimisations manually would be very time consuming and expensive, and would inevitably lead us to exploring only a very small part of the design space for interpreter optimisations. This would most likely limit the performance of the resulting interpreter, because our experience is that the correct mix of interpreter optimisations is usually found by experimentation.

Our system generates C source code, which is then fed into a compiler. With respect to optimisation, there is a clear division of labour in our system. Vmgen performs relatively high-level optimisations while generating the source code.

These are made possible by vmgen's domain-specific knowledge of the structure of interpreters, and particularly of the stack. On the other hand, lower-level, traditional tasks and optimisations such as instruction selection, register allocation and copy propagation are performed by an existing optimising C compiler. This allows us to combine the benefits of domain-specific optimisation with the advantages of product-quality compiler optimisation.

For VM code disassembly and VM code generation (i.e. generating VM code from a higher-level language, or from another format at load time), a large amount of routine, similar code also appears in interpreters. 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. To change or add a VM instruction, typically all of these places have to be updated. These issues suggest that much of the source code for interpreters should be generated from a high-level description, rather than hand-coded using expensive programmer time.

We present vmIDL, a high-level domain-specific language for describing stack VM instructions. Virtual machines are often designed as stack architectures, for three main reasons: 1) It is easy to generate stack-based code from most languages; 2) stack code is very compact, requiring little space in memory; 3) stack-based code can easily be translated into other formats. Our approach combines a small DSL for describing stack effects with general purpose C code to describe how the results of the VM instruction are generated from the inputs. In addition, vmIDL makes it easy to write fast interpreters by supporting efficient implementation techniques and a number of optimisations.

4 A Domain-Specific Language

Our domain-specific language for describing VM instructions, vmIDL, is simple, but it allows a very large amount of routine code to be generated from a very short specification. The most important feature of vmIDL is that each VM instruction defines its effect on the stack. By describing the *stack effect* of each instruction at a high level, rather than as simply a sequence of low-level operations on memory locations, it is possible to perform domain-specific optimisations on accesses to the stack.

4.1 Instruction Specifications

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

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

The stack effect (which is described by the first line) contains the following information: the number of items popped from and pushed onto the stacks, their order, which stack they belong to (we support multiple stacks for implementing VMs such as Forth, which has separate integer and floating point stacks), 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` onto the data stack.

A significant amount of C code can be automatically generated from this simple stack effect description. For example, C variables are declared for each of the stack items, code to load and store items from the stack, code to write out the operands and results while tracing the execution of a program, for writing out the immediate arguments when generating VM code from source code, and when disassembling VM code. Similarly, because the effect on the stack is described at a high level, code for different low-level representations of the stack can be generated. This feature allows many of the stack optimisations described in section 7.

4.2 Special Keywords

In addition to the stack effects, our language also provides some keywords that can be used in the C code which have special meaning to our generator.

`SET_IP` This keyword sets the VM instruction pointer. It is used for implementing VM branches.

`TAIL` This keyword indicates that the execution of the current VM instruction ends and the next one should be invoked. Using this keyword is only necessary when there is an early exit out of the VM instruction from within the user-supplied C code. `Vmgen` automatically appends code to invoke the next VM instruction to the end of the generated C code for each VM instruction, so `TAIL` is not needed for instructions that do not branch out early.

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. To improve branch prediction accuracy, we use `TAIL` inside the body of the `if` statement, to allow separate dispatch code for the *taken* and *not taken* (fall-through) branches of the `if` (see section 7.4).

```
vmIDL_spec  → { simple_inst | super_inst | comment | declaration }
simple_inst → inst_id ( stack_effect ) { C_code }
stack_effect → {item_id} -- {item_id}
super_inst  → inst_id = inst_id {inst_id}
comment     → \ comment_string
declaration → \E (stack_def | stack_prefix | type_prefix)
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
```

Fig. 1. Simplified EBNF grammar for vmIDL

4.3 Types

The type of a stack item is specified through its prefix. 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 vmIDL file:

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

The `s" int"` indicates the C type of the prefix (`int`). In our current implementation, this line is executable Forth code, and the slightly unorthodox `s"..."` syntax is used to manipulate the string `"int"`. The qualifier `single` indicates that this type takes only one slot on the stack, `data-stack` is the default stack for stack items of that type, and `i` is the name of the prefix. If there are several matching prefixes, the longest one is used.

4.4 Programming Language Issues

Figure 1 shows a simplified grammar for vmIDL. Terminal symbols are shown in bold font. A vmIDL specification consists of zero or more instances of each of the major features of the language, which include the following. A simple instruction is a standard instruction specification of the type shown in section 4.1. It consists of the name of the instruction, a stack effect and some C code to perform the computation in the instruction.

A superinstruction is a compound instruction that consists of a sequence of simple instructions, but that incurs only the interpreter overhead of executing a single instruction (see section 7.5). A programmer specifies a superinstruction by writing a name for the superinstruction, followed by the sequence of names of the simple instructions that the superinstruction consists of. Given this simple declaration, vmgen automatically constructs source code to implement the superinstruction from the instruction definitions of the component instructions. No further programmer intervention is needed. Note that the grammar description of a superinstruction allows the list of simple instructions to have only one element. In a current research (i.e. unreleased) version of vmIDL, we use this feature to implement multiple versions of simple instructions (see section 7.3).

Comments in vmIDL are specified with a backslash followed by a space at the start of a line. Our work on interpreter generators originated in an implementation of the Forth language [9], and for this reason the Forth comment character is used.

The final major feature in vmIDL is a declaration. A stack declaration is used to declare the name of a stack, the name of the stack pointer used to access that stack, and the type of the data stored in that stack (typically some neutral type, such as `void *`). When a VM uses multiple stacks, a stack prefix can be declared. Finally, type prefixes are used to identify how data on the stack should be interpreted (such as whether the value at the top of the stack should be interpreted as an integer or floating point number).

Note that the syntax for declarations is rather unusual for a programming language. As with comments, the syntax originates with Forth. The current version of our interpreter generator system is implemented in Forth, and `\E` denotes that vmgen should escape to the Forth interpreter. Everything appearing after the `\E` is actually executable Forth code. For example, the vmIDL keyword `stack` is a Forth procedure, which is called by vmgen to declare a new stack. Although it is intended that this escape facility will only be used for declarations, it allows our vmgen to be enormously flexible, since any valid Forth code can be inserted in an escape line.

A great deal of research on domain-specific languages is concerned with semantic issues such as reasoning about the properties of the described system, checking for consistency, and type systems [5]. Our work on vmIDL does not address these issues at all. The burden of finding semantic errors in the instruction definition falls entirely on the programmer, in much the same way as if the interpreter were written entirely in C, without the help of a generator. In fact, our current system is deliberately lightweight, with only just enough functionality to automatically generate the C code that would normally be written manually. Our experience is that this is sufficient for the purposes of building efficient VM interpreters, although occasionally we must examine the generated C code to identify errors. Our work on vmIDL operates under the same economics as many other domain-specific languages; the user base is not sufficiently large to support features that are not central to building the interpreter.

5 Generator Output

Given an instruction definition in vmIDL, our generator, vmgen, creates several different files of code, which are included into wrapper C functions using the C preprocessor `#include` feature. By generating this code from a single definition, we avoid having to maintain these different sections of code manually.

5.1 Interpreter Engine

Figure 2 shows the vmgen output for the `iadd` VM instruction. It starts with the label of the VM instruction. Note that vmgen supports both interpreters with

```
LABEL(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 */
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) */
}
```

Fig. 2. Simplified version of the code generated for the `iadd` VM instruction.

switch and threaded dispatch, as well as other dispatch methods, such as using function pointers. The C macro `LABEL()` must be defined appropriately by the programmer to allow the C code to be the target of a switch, goto, or function call.

Next, the stack items used by the instruction are declared. `NEXT_P0`, `NEXT_P1`, and `NEXT_P2` are macros for the instruction dispatch sequence, which facilitate prefetching the next VM instruction (see section 7.1). 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). Next 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 load and store values from the stack. The variable `spTOS` is used for top-of-stack caching (see section 7.2), while `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. Note that if the VM instruction uses the `TAIL` keyword to exit an instruction early, then the outputted C code will contain an additional copy of the code to write results to the stack and dispatch the next instruction at the early exit point.

This C code looks long and inefficient (and the complete version is even longer, since it includes trace-collecting and other code), but GCC[3] optimises it

[3] Other compilers (such as Intel's Compiler for Linux) usually produce similar assembly code for the stack access. Our experience is that most mainstream compilers perform copy propagation and register allocation at least as well as GCC. However, instruction dispatch is more efficient with GCC, since GNU C's labels-as-values extension can be used to implement threaded dispatch, rather than switch dispatch [10].

```
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
```

Fig. 3. Alpha code produced for `iadd`

quite well and produces the assembly code we would have written ourselves on most architectures we looked at, such as the Alpha code in figure 3.

5.2 Tracing

A typical C debugger is not well suited for debugging an interpreter because the C debugger works at a too-low level and does not know anything about the interpreted program. Figure 4 shows the tracing code that we left out of figure 2. `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 user defines the `printarg_` functions and can thus control how the arguments and results are displayed.

```
LABEL(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 */
```

Fig. 4. Tracing code generated for the `iadd` VM instruction.

In addition to a tracing mechanism, we believe that a VM-level debugger would also be useful. This would allow us to set breakpoints, single-step

through the program, examine the contents of the stack, instruction pointer, stack pointer, etc. A version of the interpreter that supports such interaction would be relatively easy to generate from the instruction definition, and would greatly enhance Vmgen's debugging facilities. At the time of writing, the current version does not yet generate such a debugger.

5.3 VM Code Generation

Vmgen generates functions for writing VM instructions and immediate arguments to memory. For each VM instruction, a function is generated which places the opcode and any operands in memory. Using standard functions makes the code more readable and avoids error-prone repeated code to store opcodes and operands. More importantly, using these functions allow the VM code to be automatically optimised as it is generated in memory. For example, if we generate the VM instructions `iload` followed by `iadd` and our interpreter offers the superinstruction `iload_iadd`, then these functions will automatically make the replacement. Similarly, other optimisations, such as instruction replication, that modify the VM code can also be automatically applied, at the time the VM code is generated.

5.4 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;
}
```

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. A similar piece of code is generated for all the VM's instruction set. 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.

5.5 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 VM instructions to replicate and sequences to combine into superinstructions.

The profiler counts the execution frequency of each basic block. At the *end of the run* the basic blocks are disassembled, and output with attached frequencies. There are scripts for aggregating this output into totals for static occurrences and dynamic execution frequencies, and to process them into superinstruction and instruction replication rules. The profiler overhead is low (around a factor of 2), allowing long-running programs to be profiled.

6 Experience

We have used vmgen to implement three interpreters: Gforth, Cacao and CVM. Our work on interpreter generators began with Forth and was later generalised to deal with the more complicated Java VM. This section describes the three implementations, and provides a discussion of integrating a vmIDL interpreter into the rest of a sophisticated JVM with such features as dynamic class loading and threads.

6.1 The Implementations

Gforth [11] is a portable product-quality interpretive implementation of Forth. Forth is a stack-based language, meaning that all computations are performed by manipulating a user-visible stack. It is primarily used for low-level systems programming and embedded systems. Forth can be implemented in only a few kilobytes of memory, and the standard Forth coding style of aggressive code factoring allows extremely compact user code. Furthermore, the Forth language is designed to be parsed and compiled to VM code very efficiently, allowing interactive Forth systems in very small amounts of memory. Thus, many small embedded systems such as camera and remote sensor systems provide a small, interactive version of Forth, to allow engineers with a terminal to interact with the system easily. Perhaps the most mainstream desktop Forth application is the OpenBoot system which is used to boot all Sun Workstations.

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 project started in 1992 and Gforth has been distributed as a GNU package since 1996. The current version has been ported to six different architectures, and to Unix, DOS and Windows. GForth is the perfect example of a system where portability, simplicity, maintainability and code size are more important than execution speed. On average, Gforth is just under five times slower than BigForth [12] a popular Forth native code compiler, but is about 40% faster than Win32Forth, a widely used Forth interpreter implemented in assembly language; and more than three times faster than PFE (Portable Forth Environment), a widely-used C implementation of Forth [13].

Our second implementation is an interpreter-based variant of the *Cacao* JVM JIT compiler for the Alpha architecture [14]. The goals of building the Cacao

interpreter were to see how useful vmgen is for implementing interpreters other than Gforth, to add any missing functionality, and to be able to measure the performance of our optimisations compared with other interpreters and JIT compilers. The Cacao interpreter performs well compared to other JVMs running on Alpha. It is more than ten times faster than the Kaffe 1.0.5 and DEC JVM 1.1.4 interpreters. On large benchmarks, the overall running time is less than 2.5 times slower than the Cacao JIT compiler. However, much of this time is spent in Cacao's slow run-time system, so the true slowdown of our interpreter over the Cacao JIT compiler is closer to a factor of 10 [13].

The Cacao interpreter implementation is rather unstable, and does not implement all aspects of the JVM standard correctly. Furthermore, as it runs only on the Alpha architecture, it is difficult to compare with other JVM implementations. For this reason, we have recently embarked on a new JVM implementation, based on Sun's CVM, a small implementation of the Java 2 Micro Edition (J2ME) standard, which provides a core set of class libraries, and is intended for use on devices with up to 2MB of memory. It supports the full JVM instruction set, as well as full system-level threads. Our new interpreter replaces the existing interpreter in CVM. Our CVM interpreter is similar to the Cacao implementation, except that it follows the standard JVM standard fully, and it is stable and runs all benchmark programs without modification. Experimental results [15] show that on a Pentium 4 machine the Kaffe 1.0.6 interpreter is 5.76 times slower than our base version of CVM without superinstructions on standard large benchmarks. The original CVM is 31% slower, and the Hotspot interpreter, the hand-written assembly language interpreter used by Sun's Hotspot JVM is 20.4% faster than our interpreter. Finally, the Kaffe JIT compiler is just over twice as fast as our version of CVM.

6.2 Integration Issues

Our work originates in Forth, and a number of issues arose when implementing a full version of the JVM, which is much more complicated than Forth VMs. One important difference between Forth and the JVM is that Forth uses the same stack for all functions, whereas the JVM has a separate stack for each method. The result is that call and return instructions in the JVM must save and restore the stack pointer and stack cache. This was not particularly well supported in vmgen because it happens so rarely in Forth, so new features have been added to the experimental version of vmgen being used for the CVM implementation.

A similar problem arises with exceptions. Several JVM instructions, such as array access and integer division can throw an exception. The result is that control moves to the most recent exception handler for that type of exception, which may be in the current method, or may be in another method further up the call stack. Implementing exception handling correctly is not simple, but it is mostly orthogonal to vmgen. Although it appears that exceptions could complicate vmgen's stack optimisations, in fact the operand stack is cleared when an exception is thrown. So while stack cache and other variables must be reloaded after an exception, it is not much more complicated than writing

vmIDL code for method calls and returns. The complicated exception handling code must be written by the programmer outside vmIDL.

A more difficult problem arose with the JVM's dynamic loading and initialisation of classes. New classes can be loaded at any time, so that the currently executing method may contain references to objects of a class that has not yet been loaded. Furthermore, each class contains an initialiser which must be executed exactly the first time an object or a static field or method of that class is accessed [1]. The standard way to implement JVM instructions that can access the fields and methods of other classes is to have two versions of each such instruction. The first version loads and initialises the class, if necessary. It also finds offsets for any field or method references to avoid costly lookups on future executions. This instruction then replaces itself in the instruction stream with its corresponding *quick* version, which does not perform the initialisations, and has the necessary offsets as immediate operands, rather than symbolic references.

Our CVM implementation does not interpret original Java bytecode. Instead we take Java bytecode, and produce direct-threaded code [16] using vmgen's VM code generation functions. These generated functions replace sequences of simple VM instructions with superinstructions as the VM code is generated. However, quick instructions make this process much more complicated, since the VM code modifies itself after it is created. Our current version performs another (hand-written) optimisation pass over the method each time an instruction is replaced by a quick version. This solution effective, but makes poor use of vmgen's features for automatic VM code optimisation. It is not clear to us how vmgen can be modified to better suit Java's needs in this regard, while still remaining simple and general.

CVM uses system-level threads to implement JVM threads. Several threads can run in parallel, and in CVM these run as several different instances of the interpreter. As long as no global variables are used in the interpreter, these different instances will run independently. Implementing threads and monitors involves many difficult issues, almost all of which are made neither simpler nor more difficult by the use of vmIDL for the interpreter core. One exception to this was with quick instructions. The same method may be executed by simultaneously by several different threads, so race conditions can arise with quick instructions which modify the VM code. We eventually solved this problem using locks on the VM code when quickening, but the solution was not easily found. If we were to implement the system again, we would implement threading within a single instance of the interpreter, which would perform its own thread switches periodically. Interacting with the operating system's threading system is complicated, and reduces the portability of the implementation.

A final complication with our CVM interpreter arose with garbage collection. CVM implements precise garbage collection, using stack maps to identify pointers at each point where garbage collection is possible. In our implementation, at every backward branch, and at every method call, a global variable is checked to see whether some thread has requested that garbage collection should start. If it has, then the current thread puts itself into a garbage collection safe-state and

waits for the collection to complete. The use of vmIDL neither helps nor hinders the implementation of garbage collection. Entering a safe state involves saving the stack pointer, stack cache and other variables in the same way as when a method call occurs. It seems possible that in the future, vmgen's knowledge of the stack effect of each instruction could be used to help automatically generate stack maps. However, the current version contains no such feature, and items on the stack remain, essentially, untyped. The main thrust of our current vmgen work is interpreter optimisation, as we show in the next section.

7 Optimisations

This section describes a number of optimisations to improve the execution time of interpreters, and how they can be automatically applied by vmgen to a vmIDL definition.

7.1 Prefetching

Perhaps the most expensive part of executing a VM instruction is dispatch (fetching and executing the next VM instruction). One way to help the dispatch branch to be resolved earlier is to fetch the next instruction early. Therefore, vmgen generates three macro invocations for dispatch (`NEXT_P0`, `NEXT_P1`, `NEXT_P2`) and distributes them through the code for a VM instruction (see figure 2).

These macros can be defined to take advantage of specific properties of real machine architectures and microarchitectures, such as the 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 uses this on the PowerPC architecture to good advantage (about 20% speedup).

7.2 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 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 one or more results on that stack. This halves the number of data-stack memory accesses in Gforth [17]. The benefits can be seen in figure 3 (only one memory access for three stack accesses).

The downside of this optimisation is that it requires an additional register, possibly spilling a different VM register into memory. Still, we see an overall speedup of around 7%-10% for Gforth even on the register-starved IA32 (Pentium, Athlon) architecture [13]. On the PowerPC the speedup is even larger (just over 20%) as would be expected on a machine with many registers.

Vmgen performs this optimisation by replacing `sp[0]` with the local variable name `spTOS` when referencing stack items. Presuming the compiler allocates this variable to a register, the benefits of top-of-stack caching occur. In addition, C

code is generated to flush or reload the stack for those instructions that affect the stack height without necessarily using the topmost stack element.

7.3 Instruction Replication

Mispredictions of indirect branches are a major component of the run-time of efficient interpreters [10]. Most 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.

The machine code to implement a VM instruction always ends with an indirect branch to dispatch the next instruction. As long as, say, each `iload` instruction is followed by, say, an `iadd`, the indirect branch at the end of the `iload` will generally be predicted correctly. However, this is rarely the case, and it often happens that the same VM instruction appears more than once in the working set, each time with a different following VM instruction.

Instruction replication splits a VM instruction such as `iadd` into several copies: `iadd1`, `iadd2`, etc. When generating VM code and an `iadd` instruction is needed, one of the replicated versions of `iadd` is actually placed in the generated code. The different versions will have separate indirect branches, each of which is predicted separately by the BTB. Thus, the different versions can have different following VM instructions without causing mispredictions. The VM instructions to replicate are selected using profiling information. The list of instructions to replicate is included in the vmIDL input file, and vmgen automatically generates separate C source code for each replication.

We tested this optimisations on several large Forth programs, and found that it can reduce indirect branch mispredictions in Gforth by almost two thirds, and running time by around 25% [18]. The experimental version of vmgen that implements this optimisation uses superinstructions of length one to replicate instructions.

7.4 VM Branch Tails

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. The vmIDL language supports this optimisation with the keyword `TAIL`. Vmgen expands this macro into the whole end-part of the VM instruction.

We evaluated the effect of using different indirect jumps for the different outcomes of VM conditional branches in GForth. We found speedups of 0%–9%, with only small benefits for most programs. However, we found a small reduction in the number of executed instructions (0.6%–1.7%); looking at the assembly language code, we discovered that GCC performs some additional optimisations if we use `TAIL`.

7.5 Superinstructions

A superinstruction is a new, compound VM instruction that performs the work of a sequence of simple VM instructions. Superinstructions are chosen using the output of the profiler generated by vmgen. The list of selected sequences to make into superinstruction is included in the vmIDL input file by the programmer. Given this list, vmgen automatically generates C code to implement the superinstructions from the instruction definition of the component VM instructions.

In a superinstruction, vmgen keeps all intermediate stack-values in local variables (which we hope will be allocated to registers), allowing values to be passed from one component VM instruction to another without the usual loads and stores for stack accesses. In addition stack pointer updates from the different component instructions are combined, sometimes allowing the update to be eliminated entirely if the two component updates cancel one another. For example, the generated code for the superinstruction `iload-iadd` is actually shorter than that for either of its component VM instructions on x86 machines [15], because all stack memory accesses and stack pointer updates can be eliminated. Overall, adding superinstructions gives speedups of between 20% and 80% on Gforth [13].

7.6 Multiple-State Stack Caching

As mentioned in section 7.2 keeping the topmost element of the stack in a register can reduce memory traffic for stack accesses by around 50%. Further gains are achievable by reserving two or more registers for stack items. The simplest way to do this is to simply keep the topmost n items in registers. For example, the local variable[4] `TOS_3` might store the top of stack, `TOS_2` the second from top and `TOS_1` the next item down. The problem with this approach can be seen if we consider an instruction that pushes an item onto the stack. This value of this item will be placed in the variable `TOS_3`. But first, the current value of `TOS_3` will be copied to `TOS_2`, since this is now the second from topmost item. The same applies to the value in `TOS_1`, which must be stored to memory. Thus, any operation that affects the height of the stack will result in a ripple of copies, which usually outweigh the benefits of stack caching [17].

A better solution is to introduce multiple states into the interpreter, in which the stack can be partially empty. For example, a scheme with three stack-cache registers would have four states:

- **State 0**: cache is empty
- **State 1**: one item in the cache; top-of-stack is in `TOS_1`
- **State 2**: two items in the cache; top-of-stack is in `TOS_2`
- **State 3**: three items in the cache; top-of-stack is in `TOS_3`

[4] By keeping stack cache items in local variables, they become candidates to be allocated to registers. There is no guarantee that C compiler's register allocator will actually place those variables in registers, but they are likely to be good candidates because the stack cache is frequently used.

In this scheme, there will be four separate versions of each virtual machine instruction — one for each state. Each version of each instruction will be customised to use the correct variable names for the topmost stack items. All of this code is generated by vmgen automatically from the vmIDL definition. Figure 5 shows the output of vmgen for one state of the `iadd` VM instruction. The two operands are in the cache at the start of the instruction, and so are copied from `TOS_1` and `TOS_2`. The result is put into the new topmost stack location, `TOS_1`, and the state is changed[5] to state 1, before the dispatch of the next VM instruction. Note that there is no stack update in this instruction; the change in the height of the stack is captured by the change in state.

```
LABEL(iadd_state2) {   /* label */
int i1;                /* declarations of stack items */
int i2;
int i;
NEXT_P0;               /* dispatch next instruction (part 0) */
i1 = vm_Cell2i(TOS_1); /* fetch argument stack items */
i2 = vm_Cell2i(TOS_2);
{                      /* user-provided C code */
i = i1+i2;
}
NEXT_P1;               /* dispatch next instruction (part 1) */
TOS_1 = vm_i2Cell(i);  /* store result stack item(s) */
CHANGE_STATE(1);       /* switch to state 1 */
NEXT_P2;               /* dispatch next instruction (part 2) */
}
```

Fig. 5. Simplified version of the code generated for state 2 of the `iadd` instruction with multiple-state stack caching.

Multiple-state stack caching is currently implemented in an experimental, unreleased version of vmgen. Preliminary experiments show that memory traffic for accessing the stack can be reduced by more than three quarters using a three register cache.

7.7 Instruction Specialisation

Many VM instructions take an immediate argument. For example, the IGET-FIELD_QUICK instruction loads an integer field of an object, and takes as an immediate argument the offset at which at the field appears. Through profiling, we might find that a very commonly used offset is zero (indicating the first field in the object). Thus we might introduce a special version of the instruction, with the immediate operand hardwired to zero.

[5] A common way to implement the state is to use a different dispatch table or `switch` statement for each state.

An experimental version of vmgen supports instruction specialisation. A modified version of the profiler is used to measure the values of immediate arguments on sample programs. The commonest immediate values for the most frequent instructions are selected to be specialised instructions based on the profiling information. The experimental version of vmgen automatically generates C source code for these specialised instructions from the instruction definition, by setting the immediate argument to a constant value, rather than loading it from the instruction stream.

Preliminary results show that specialisation has the potential to significantly improve performance, both because it reduces the work involved in executing the instruction by removing the operand fetch, and also because having several different versions of an instruction each specialized for different operands has a similar effect on indirect branch prediction as instruction replication.

8 Related Work

Our work on generating VM interpreters is ongoing. The best reference on the current release version of vmgen is [13], which gives a detailed description of vmgen output, and presents detailed experimental results on the performance of the GForth and Cacao implementations. More recent work presents newer results on superinstructions and instruction replication [18] and the CVM implementation [15].

The C interpreter `hti` [19] 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. It uses a tree-based VM (linearized into a stack-based form) derived from `lcc`'s intermediate representation. A variation of this scheme is used for automatically generating interpreters of compressed bytecode [20, 21].

Many of the performance-enhancing techniques used by vmgen have been used and published earlier: threaded code and decoding speed [16, 22], scheduling and software pipelining the dispatch [11, 23, 24], stack caching [11, 17] and combining VM instructions [19, 25, 24, 26]. Our main contribution is to automate the implementation of these optimisations using a DSL and generator.

9 Conclusion

Virtual machine interpreters contain large amounts of repeated code, and optimisations require large numbers of similar changes to many parts of the source code. We have presented an overview of our work on vmIDL, a domain-specific language for describing the instruction sets of stack-based VMs. Given a vmIDL description, our interpreter generator, vmgen, will automatically generate the large amounts of C source code needed to implement a corresponding interpreter system complete with support for tracing, VM code generation, VM code disassembly, and profiling. Furthermore, vmgen will, on request, apply a variety of optimisations to the generated interpreter, such as prefetching the next VM

instruction, stack caching, instruction replication, having different instances of the dispatch code for better branch prediction, and combining VM instructions into superinstructions. Generating optimised C code from a simple specification allows the programmer to experiment with optimisations and explore a much greater part of the design space for interpreter optimisations than would be feasible if the code were written manually.

Availability

The current release version of the `vmgen` generator can be downloaded from: `http://www.complang.tuwien.ac.at/anton/vmgen/`.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed comments, which greatly improved the quality of this chapter.

References

1. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley, Reading, MA, USA (1999)
2. Aït-Kaci, H.: The WAM: A (real) tutorial. In: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991)
3. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley (1983)
4. Weiss, D.M.: Family-oriented abstraction specification and translation: the FAST process. In: Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland, IEEE Press (1996) 14–22
5. Czarnecki, K., Eisenecker, U.: Generative programming — methods tools and applications. Addison-Wesley (2000)
6. Lengauer, C.: Program optimization in the domain of high-performance parallelism (2004) In this volume.
7. Grune, D., Bal, H., Jacobs, C., Langendoen, K.: Modern Compiler Design. Wiley (2001)
8. Ertl, M.A.: Implementation of Stack-Based Languages on Register Machines. PhD thesis, Technische Universität Wien, Austria (1996)
9. Moore, C.H., Leach, G.C.: Forth – a language for interactive computing. Technical report, Mohasco Industries, Inc., Amsterdam, NY (1970)
10. Ertl, M.A., Gregg, D.: The behaviour of efficient virtual machine interpreters on modern architectures. In: Euro-Par 2001, Springer LNCS 2150 (2001) 403–412
11. Ertl, M.A.: A portable Forth engine. In: EuroFORTH '93 conference proceedings, Mariánské Láznè (Marienbad) (1993)
12. Paysan, B.: Ein optimierender Forth-Compiler. Vierte Dimension **7** (1991) 22–25
13. Ertl, M.A., Gregg, D., Krall, A., Paysan, B.: `vmgen` — A generator of efficient virtual machine interpreters. Software—Practice and Experience **32** (2002) 265–294
14. Krall, A., Grafl, R.: CACAO – a 64 bit JavaVM just-in-time compiler. Concurrency: Practice and Experience **9** (1997) 1017–1030

15. Casey, K., Gregg, D., Ertl, M.A., Nisbet, A.: Towards superinstructions for Java interpreters. In: 7th International Workshop on Software and Compilers for Embedded Systems. LNCS 2826 (2003) 329 – 343
16. Bell, J.R.: Threaded code. Communications of the ACM **16** (1973) 370–372
17. Ertl, M.A.: Stack caching for interpreters. In: SIGPLAN '95 Conference on Programming Language Design and Implementation. (1995) 315–327
18. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 03), San Diego, California, ACM (2003) 278–288
19. Proebsting, T.A.: Optimizing an ANSI C interpreter with superoperators. In: Principles of Programming Languages (POPL '95). (1995) 322–332
20. Ernst, J., Evans, W., Fraser, C.W., Lucco, S., Proebsting, T.A.: Code compression. In: SIGPLAN '97 Conference on Programming Language Design and Implementation. (1997) 358–365
21. Evans, W.S., Fraser, C.W.: Bytecode compression via profiled grammar rewriting. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001) 148–155
22. Klint, P.: Interpretation techniques. Software—Practice and Experience **11** (1981) 963–973
23. Hoogerbrugge, J., Augusteijn, L.: Pipelined Java virtual machine interpreters. In: Proceedings of the 9th International Conference on Compiler Construction (CC' 00), Springer LNCS (2000)
24. Hoogerbrugge, J., Augusteijn, L., Trum, J., van de Wiel, R.: A code compression system based on pipelined interpreters. Software—Practice and Experience **29** (1999) 1005–1023
25. Piumarta, I., Riccardi, F.: Optimizing direct threaded code by selective inlining. In: SIGPLAN '98 Conference on Programming Language Design and Implementation. (1998) 291–300
26. Clausen, L., Schultz, U.P., Consel, C., Muller, G.: Java bytecode compression for low-end embedded systems. ACM Transactions on Programming Languages and Systems **22** (2000) 471–489