Building an interpreter with vmgen

M. Anton Ertl[*], TU Wien
David Gregg, Trinity College, Dublin

Abstract. Vmgen automates many of the tasks of writing the virtual machine part of an interpreter, resulting in less coding, debugging and maintenance effort. This paper gives some quantitative data about the source code and generated code for a vmgen-based interpreter, and gives some examples demonstrating the simplicity of using vmgen.

1 Introduction

Interpreters are a popular approach for implementing programming languages, because only interpreters offer *all* of the following benefits: ease of implementation, portability, and a fast edit-compile-run-cycle.

The interpreter generator vmgen[1] automates many of the tasks in writing the virtual machine (VM) part of an interpretive system; it takes a simple VM instruction description file and generates code for: executing and tracing VM instructions, generating VM code, disassembling VM code, combining VM instructions into superinstructions, and profiling VM instruction sequences to find superinstructions. Vmgen has special support for stack-based VMs, but most of its features are also useful for register-based VMs.

Vmgen supports a number of high-performance techniques and optimizations. The resulting interpreters tend to be faster than other interpreters for the same language.

This paper presents an example of vmgen usage. A detailed discussion of the inner workings of vmgen and performance data can be found elsewhere [1].

2 Example overview

The running example in this paper is the example provided with the vmgen package: an interpretive system for a tiny Modula-2-style language that uses a JVM-style virtual machine. The language supports integer variables and expressions, assignments, if- and while-structures, function definitions and calls.

Our example interpreter consists of two conceptual parts: the front-end parses the source code and generates VM code; the VM interpreter executes the VM code.

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

[1] Vmgen is available at http://www.complang.tuwien.ac.at/anton/vmgen/.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Compiler Construction (CC'02), pages 5--9, Springer LNCS 2304

Name	Lines	Description
Makefile	67	
mini-inst.vmg	139	VM instruction descriptions
mini.h	72	common declarations
mini.l	42	front-end scanner
mini.y	139	front-end (parser, VM code generator)
support.c	220	symbol tables, `main()`
peephole-blacklist	3	VM instructions that must not be combined
disasm.c	36	template: VM disassembler
engine.c	186	template: VM interpreter
peephole.c	101	template: combining VM instructions
profile.c	160	template: VM instruction sequence profiling
stat.awk	13	template: aggregate profile information
seq2rule.awk	8	template: define superinstructions
	504	template files total
	682	specific files total
	1186	total

Fig. 1. Source files in the example interpreter

Figure 1 shows quantitative data on the source code of our example. Note that the numbers include comments, which are sometimes relatively extensive (in particular, more than half of the lines in `mini-inst.vmg` are comments or empty). Some of the files are marked as templates; in a typical vmgen application they will be copied from the example and used with few changes, so these files cost very little. The other files contain code that will typically be written specifically for each application.

Among the specific files, `mini-inst.vmg` contains all of the VM description; in addition, there are VM-related declarations in `mini.h`, calls to VM code generation functions in `mini.y`, and calls to the VM interpreter, disassembler, and profiler in `support.c`.

Vmgen generates 936 lines in six files from `mini-inst.vmg` (see Fig. 2). The expansion factor from the source file indicates that vmgen saves a lot of work in coding, maintaining and debugging the VM interpreter.

Name	Lines	Description
mini-disasm.i	103	VM disassembler
mini-gen.i	84	VM code generation
mini-labels.i	19	VM instruction codes
mini-peephole.i	0	VM instruction combining
mini-profile.i	95	VM instruction sequence profiling
mini-vm.i	635	VM instruction execution
	936	total

Fig. 2. Vmgen-generated files in the example interpreter

In addition to the reduced line count there is another reason why vmgen reduces the number of bugs: a new VM instruction just needs to be inserted in one place in mini-inst.vmg (and code for generating it should be added to the front end), whereas in a manually coded VM interpreter a new instruction needs code in several places.

The various generated files correspond mostly directly to template files, with the template files containing wrapper code that works for all VMs, and the generated files containing code or tables specific to the VM at hand.

3 Simple VM instructions

A typical vmgen instruction specification looks like this:

```
sub ( i1 i2 -- i )
i = i1-i2;
```

The first line gives the name of the VM instruction (sub) and its stack effect: it takes two integers (i1 and i2) from the stack and pushes one integer (i) on the stack. The next line contains C code that accesses the stack items as variables. Loading i1 and i2 from and storing i to the stack, and instruction dispatch are managed automatically by vmgen.

Another example:

```
lit ( #i -- i )
```

The lit instruction takes the immediate argument i from the instruction stream (indicated by the # prefix) and pushes it on the stack. No user-supplied C code is necessary for lit.

4 VM code generation

These VM instructions are generated by the following rules in mini.y:

```
expr: term '-' term { gen_sub(&vmcodep); }
term: NUM           { gen_lit(&vmcodep, $1); }
```

The code generation functions gen_sub and gen_lit are generated automatically by vmgen; gen_lit has a second argument that specifies the immediate argument of lit (in this example, the number being compiled by the front end).

Parsing and generating code for all subexpressions, then generating the code for the expression naturally leads to postfix code for a stack machine. This is one of the reasons why stack-based VMs are very popular in interpreters. The programmer just has to ensure that all rules for term and expr produce code that leaves exactly one value on the stack.

The power of yacc and its actions is sufficient for our example, but for implementing a more complex language the user will probably choose a more sophisticated tool or build a tree and manually code tree traversals. In both cases, generating code in a post-order traversal of the expression parse tree is easy.

5 Superinstructions

In addition to simple instructions, you can define superinstructions as a combination of a sequence of simple instructions:

```
lit_sub = lit sub
```

This defines a new VM instruction `lit_sub` that behaves in the same way as the sequence `lit sub`, but is faster.

After adding this instruction to `mini-inst.vmg` and rebuilding the interpreter, this superinstruction is generated automatically whenever a call to `gen_lit` is followed by a call to `gen_sub`.

But you need not even define the superinstructions yourself, you can generate them automatically from a profile of executed VM instruction sequences:

You can compile the VM interpreter with profiling enabled, and run some programs representing your workload. The resulting profile lists the number of dynamic executions for each static occurence of a sequence, e.g.,

```
      18454929  lit sub
                ...
       9227464  lit sub
```

This indicates that the sequence `lit sub` occured in two places, for a total of 27682393 dynamic executions. These data can be aggregated with the `stat.awk` script, then the user can choose the most promising superinstructions (typically with another small awk or perl script), and finally transform the selected sequences into the superinstruction rule syntax with `seq2rule.awk`.

The original intent of the superinstruction features was to improve the run-time performance of the interpreter (and it achieves this goal), but we also noticed that it makes interpreter construction easier: In some places in an interpretive system, we can generate a sequence of existing instructions or define a new instruction and generate that; in a manually written interpreter, the latter approach yields a faster interpreter, but requires more work. Using `vmgen`, you can just take the first approach, and let the sequence be optimized into a superinstruction if it occurs frequently; in this way, you get the best of both approaches: little effort and run-time performance.

References

1. M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. `vmgen` — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 2002. Accepted for publication.