vmgen - A Generator of Efficient Virtual Machine Interpreters

M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan
Presented by Peter Bailey

May 6, 2011
vmgen generates fast interpreters from instruction descriptions
also generates parts of associated tools
- profiler
- debugger
- disassembler
- code generator
Motivation

- writing/modifying an interpreter toolset is tedious and error-prone
  - many parts can be automated
- can generated interpreters compete with those hand-written in assembly?
Motivation

- C compiler does most of the complicated things
- vmgen makes modifying an instruction set easier than rewriting *anything* in assembly
Process

- inputs: description of instruction set
- outputs: C code
  - interpreter
  - profiler
  - debugger
  - VM code disassembly
  - VM code generation

M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan P

vmgen - A Generator of Efficient Virtual Machine Interpreters
Process

- producing a working interpreter requires a bit more work
  - C code for interpreter skeleton
  - C code from vmgen
  - C compiler
Figure: vmgen process
input format:

iadd:

iadd ( i1 i2 -- i )

i = i1 + i2;

- name
- stack effect, input and output types
- C implementation code
I_iadd:

```
int i1, i2, i;
NEXT_P0;
i1 = vm_Cell2i(sp[1]);
i2 = vm_Cell2i(sp[0]);
sp += 1;
{
    i = i1 + i2;
}
NEXT_P1;
sp[0] = vm_i2Cell(i);
NEXT_P2;
```
- designed and optimized for stack-based VMs
  - but register-based VMs are possible
- generated interpreter uses direct threading
  - but indirect threading is possible
- flexible!
Optimizations

- vmgen interpreters are designed for optimization
- built-ins
  - TOS caching, software pipelining, efficient stack usage
- tail duplication for branch prediction
- superinstructions
Existing optimizations

- TOS caching
- software pipelining/scheduled dispatch
  - interleave instruction execution with instruction fetch
- superinstructions
Superinstructions

- not superoperators
  - superoperators are tree operators
  - superinstructions are DAG operators, work on stack-based interpreters
- arbitrary combination of previously-defined instructions
Superinstructions

- consequences
  - C compiler ideally generates more efficient code
  - VM code generator generates fewer instructions
  - interpreter interprets fewer instructions
  - profiler can recommend superinstructions
store elimination
  example:
    dup ( i -- i i )
  avoid creating a temporary variable and pushing it twice
  doesn’t work with superinstructions

tail duplication for branch prediction
two interpreters built with vmgen

- Gforth: Forth interpreter
- Cacao int: JVM interpreter, with threaded code instead of byte code
Gforth is faster than Win32Forth
- Win32Forth is written in assembly, but uses indirect threading and PIC

Gforth is slower than BigForth
- BigForth compiles Forth to native code
Cacao int is faster than the DEC JVM native JIT compiler for some benchmarks.

Cacao int is slower than Cacao native, but only by a factor of two for most benchmarks.

- Cacao int and Cacao native share synchronization and garbage collection mechanisms, and Cacao int spends 30% of its time in these routines.
optimizations were generally beneficial
but architecture-dependent
  example: TOS caching improved performance on PPC by 20%,
  but net effect on a particular Alpha machine was 5%
and benchmark-dependent
quality of resulting interpreter depends on quality of compiler used to build interpreter

authors claim GCC does a good job, but did not verify all compiled code

authors manually allocated registers in Gforth because GCC inappropriately spilled important interpreter registers