A VCODE Tutorial

Dawson R. Engler
engler@lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

April 28, 1996

Abstract

This paper is a short tutorial on VCODE, a fast, portable dynamic code generation system. It should be read after [1].

1 Introduction

The VCODE system is a set of C macros and support functions that allow programmers to portably and efficiently generate code at runtime. The VCODE interface is that of an idealized load-store RISC architecture. VCODE instructions are simple primitives (e.g., add, sub, load) that map readily to modern architectures.

This tutorial is intended for programmers interested in using the VCODE system or in readers interested in details not covered in [1]. Section 2 discusses a simple example program, touching on parts of the VCODE system. Section 3 discusses the VCODE instruction set and Section 4 provides a global view of the system (most of this information is provided in [1], we provide it here for completeness). The remainder of the tutorial deals with different functional aspects: function construction is discussed in Section 5, storage management of registers and local variables in Section 6, control flow primitives (labels, branchs, jumps) in Section 7, function call construction is discussed in Section 8. Advanced VCODE features such as debuging, portable delay-slot scheduling and hard-wired register usage are discussed in Section 9. Finally, known bugs and future work are discussed in Section 10.

2 Overview

This section dissects a simple program that uses VCODE to dynamically create, compile and run a function to print a traditional greeting:

```
#include <stdio.h>
#include "vcode.h"            /* This header file defines all vcode insns. */

int main(void) {
        static v_code insn[1000];       /* Memory to hold code in. */

        /* Create a function to print standard greeting. */
        v_lambda("hello-world", "", 0, V_NLEAF, insn, sizeof insn);
                /* Generate simple call to printf. */
                v_scallv((v_vptr)printf, "%P", "hello, world\n");
        v_end(0).v();   /* Compile & call */
        return 0;
}
```

The second line of the program:

```
#include "vcode.h"
```

includes the public VCODE interface: macros for most VCODE instructions, data structures for VCODE objects (registers, labels, etc.), and varous library function prototypes. All clients must include this file. VCODE generates code into user-supplied storage. The declaration

```
static v_code insn[1000];
```

allocates space for 1000 machine instructions. At the start of code generation a pointer to this memory (along with its size) is given to VCODE. Currently the number of instructions allocated must be "big enough." VCODE will signal a fatal error if this space overflows. [1]

VCODE creates a single function at a time. Clients start code generation for a function by calling v_lambda:

```
v_lambda("hello-world", "", 0, V_NLEAF, insn, sizeof insn);
```

This statement begins code generation for a function called "hello-world" that takes no arguments (the format string "" would contain type declarations otherwise). Since the created function will call printf, the client indicates that it is not a leaf procedure using the enumerated type V_NLEAF. The final two arguments are a pointer to and the size of storage that VCODE will write machine instructions into.

The generated code will contain the code for the v_scallv instruction:

```
v_scallv((v_vptr)printf, "%P", "hello, world\n");
```

This instruction generates code to call the function pointer given as its first argument (in this case printf). Its second argument is a format string used to declare the function's argument types. This information is required in order to compute where arguments will be placed. In this case, the example code uses the format string "%P" to declare that it is calling a function that takes a single pointer as an argument (i.e., a pointer to the string "hello, world"). The fact that the example uses a capitalized type initial ("%P") instead of a lower-case type initial ("%p") tells VCODE that the argument is an immediate rather than a register.

VCODE types are equivalent to C types (int, unsigned, etc.) and are encoded using the initials of their C counterparts (e.g., void is encoded using the type initial v, unsigned long using ul). All VCODE instructions have a suffix indicating their type. In the given example, the function call is to printf, which we effectively treat as a function with a void return type. Therefore, a call to it is of type v: v_scallv.

Finally, the statement

```
v_end(0).v();
```

causes the current function to be compiled and then invoked it. v_end returns a union of function pointers based on common return types. Each of these function pointers is named by its return type, and can be accessed using this types type initial. In this example, the generated code has a void return type and, as a result, we call it using a void function pointer (i.e., the .v() portion of the statement).

To compile the program we need to link against the VCODE runtime library. Assuming the library is in the current working directory on Unix, this can be done as follows:

```
cc hello.c vcode.lib.a
```

With the exception of exploiting undefined behavior in the name of dynamic code generation (e.g., calling of heap-allocated data as a function), VCODE is ANSI compliant and should be palatable to any ANSI compiler.

[1] Instruction memory should not be allocated on the stack: some operating systems (e.g., DEC OSF/1) map the stack without execute permissions.

Type	C equivalent
v	void
c	signed char
uc	unsigned char
s	signed short
us	unsigned short
i	int
u	unsigned
p	void *
l	long
ul	unsigned long
f	float
d	double

Table 1: VCODE types

3 Instruction set architecture

The VCODE instruction set was designed by choosing and deriving instructions that closely match those of most existing RISC architectures. This process has also been influenced by a number of compiler intermediate representations, the strongest influence being the intermediate representation language of the lcc compiler [2].

The instruction set is built from a set of base operations (e.g., sub, mul) that are composed with a set of types (e.g., integer, unsigned). Each instruction takes register or immediate operands and, usually, performs a simple operation on them.

VCODE supports a full range of types: signed and unsigned bytes, halfwords, words and long words and single and double precision floating-point. The base VCODE types, named for their mappings to ANSI C types, are listed in Table 1. Some of these types may not be distinct (e.g., l is equivalent to i on 32-bit machines). Each VCODE instruction operates on some number of typed operands. To reduce the instruction set, and because most architectures only provide word and long word operations on registers, most non-memory VCODE operations do not take the smaller data types (i.e., c, uc, s, and us) as operands.

The VCODE instruction set consists of a single *core layer* that must be retargeted for each new machine and multiple *extension layers* that are built on top of this core.

The core layer consists of instructions not readily synthesized from other instructions, such as add. Table 2 lists the VCODE core. Extension layers provide additional functionality less general than that of the core (e.g., conditional move, floating-point square root). For porting convenience, most of these extensions are expressed in terms of the core itself. Therefore, once the core has been retargeted, extensions will work on the new machine as well. However, for efficiency, these default definitions can be overridden and implemented instead in terms of the resources provided by the actual hardware. This duality of implementation allows site-specific extensions and common idioms to be implemented in a portable manner without affecting ease of retargeting.

Table 3 lists the instructions in the primary extension layer. [2] This layer consists of instructions that can be synthesized from VCODE core instructions but are important enough to some class of applications that the VCODE implementor should attempt to provide them in terms of machine instructions. Other VCODE extension layers deal with bit manipulation (such as byte-swapping), control operations (e.g., branch prediction), and floating-point instructions.

4 Operational overview

Client programs specify code using VCODE's machine-independent instruction set. This instruction set is simple and regular. These properties are important because the instructions must be easily generated by client programs. In essence, every client program is a small compiler front-end.

VCODE transliterates the instructions selected by clients to machine code immediately, without the code generation passes typical of other code generators. VCODE omits any significant global optimizations and pipeline scheduling,

[2] Few of these instructions are currently implemented (or tested!) with any consistency across architectures. If you need them (or others) let me know.

Standard binary operations (*rd*, *rs1*, *rs2*†)

add	i u l ul p f d	addition
sub	i u l ul p f d	subtraction
mul	i u l ul f d	multiplication
div	i u l ul f d	division
mod	i u l ul	modulus
and	i u l ul	logical and
or	i u l ul	logical or
xor	i u l ul	logical xor
lsh	i u l ul	left shift
rsh	i u l ul	right shift; the sign bit is propagated for signed types

Standard unary operations (*rd*, *rs*)

com	i u l ul	bit complement
not	i u l ul	logical not
mov	i u l ul p f d	copy *rs* to *rd*
neg	i u l ul f d	negation
set	i u l ul p f d	load constant into *rd*; *rs* must be an immediate
cvi2	u ul l	convert integer to type
cvu2	i ul l	convert unsigned to type
cvl2	i u ul f d	convert long to type
cvul2	i u l p	convert unsigned long to type
cvp2	ul	convert pointer to type
cvf2	l d	convert float to type
cvd2	l f	convert double to type

Memory operations (*rd*, *rs*, *offset*†)

ld	c uc s us i u l ul p f d	load
st	c uc s us i u l ul p f d	store

Return to caller (*rs*)

ret	v i u l ul p f d	return value

Jumps (*addr*)

j	v p	jump to immediate, register, or label
jal	v p	jump and link to immediate, register, or label

Branch instructions (*rs1*, *rs2*†, *label*)

blt	i u l ul p f d	branch if less than
ble	i u l ul p f d	branch if less than equal
bgt	i u l ul p f d	branch if greater than
bge	i u l ul p f d	branch if greater than equal
beq	i u l ul p f d	branch if equal
bne	i u l ul p f d	branch if not equal

Nullary operation

nop		no operation

†This operand may be an immediate provided its type is not f or d.

Table 2: Core VCODE instructions.

Binary operations (*rd*, *rs1*, *rs2*†)

lt	i u l ul p f d	less than
le	i u l ul p f d	less than equal
gt	i u l ul p f d	greater than
ge	i u l ul p f d	greater than equal
eq	i u l ul p f d	equal
ne	i u l ul p f d	not equal
nand	i u l ul	logical not and
nor	i u l ul	logical not or
nxor	i u l ul	logical not xor
mulhi	s us i u l ul p	high bits of product

Network extensions (*rd*, *rs*)

ntoh	s us i u l ul	Network-to-host order
hton	s us i u l ul	Host-to-network order
bswap	s us i u l ul	Byteswap

Memory operations (*rd*, *imm*)

ldea	i u l ul f d	load effective address
stea	i u l ul f d	store effective address

Unary operations (*rd*, *rs*)

abs	f d	absolute value
sqrt	f d	square root
ceil	f d	ceiling
floor	f d	floor

Memory operations (*rd*, *rs*, *offset*†)

uld	s us i u l ul p	unaligned load
ust	s us i u l ul p	unaligned store

Conditional moves (*rd*, *rs*†, *b1*, *b2*†)

cmveq	i u l ul p f d	move *rs* to *rd* if $b1 = b2$
cmvne	i u l ul p f d	move *rs* to *rd* if $b1 \neq b2$
cmvlt	i u l ul p f d	move *rs* to *rd* if $b1 < b2$
cmvle	i u l ul p f d	move *rs* to *rd* if $b1 \leq b2$
cmvgt	i u l ul p f d	move *rs* to *rd* if $b1 > b2$
cmvge	i u l ul p f d	move *rs* to *rd* if $b1 \geq b2$

Prefetching instructions (*src*, *offset*†, *sz*, *flags*)

prefetch_R		prefetch *sz* bytes for reading
prefetch_W		prefetch *sz* bytes for writing

†This operand may be an immediate provided its type is not f or d.

Table 3: Primary VCODE extensions

```
typedef int (*iptr)(int);
/* Called at runtime to create a function which returns its argument + 1. */
iptr mkplus1(v_code *ip, int nbytes) {
   v_reg arg[1];

   /* Begin code generation. The type string ("%i") indicates that this routine takes a single integer (i)
    argument; the register to hold this argument is returned in arg[0]. V_LEAF indicates that this
    function is a leaf procedure. ip is a pointer to storage to hold the generated code. */
    v_lambda("%i", arg, V_LEAF, ip, nbytes);

    /* Add the argument register to 1. */
    v_addii(arg[0], arg[0], 1);   /* ADD Integer Immediate */
    /* Return the result. */
    v_reti(arg[0]);               /* RETurn Integer */

    /* End code generation. v_end links the generated code and performs
       cleanup. It then returns a pointer to the final code. */
    return (iptr)v_end(0);
}
```

Figure 1: VCODE specification for function corresponding to int plus1(int x) { return x + 1; }

which would require at least a single pass over some intermediate representation, slowing VCODE by an order of magnitude. (Clients that desire such optimizations can layer them on top of the generic VCODE system.) Global optimizations are the responsibility of the client, which has access to the low-level VCODE instruction set. VCODE is only responsible for emitting efficient code locally.

VCODE includes a mechanism to allow clients to perform register allocation in a machine-independent way. The client declares an allocation priority ordering for all register candidates along with a class (the two classes are "temporary" and "persistent across procedure calls"). VCODE allocates registers according to that ordering. Once the machine's registers are exhausted, the register allocator returns an error code. Clients are then responsible for keeping variables on the stack. In practice, modern RISC architectures provide enough registers that this arrangement is satisfactory. (This scheme works on CISC machines as well, since they typically allow operations to work on both registers and memory locations.) Although the VCODE register allocator has limited scope, it does its job well; it makes unused argument registers available for allocation, is intelligent about leaf procedures, and generates code to allow caller-saved registers to stand in for callee-saved registers and vice-versa.

Complete code generation includes instruction selection, binary code emission, and jump resolution. For most instructions, the first and second steps occur at the specification site of the VCODE instruction. The only complications are jump instructions and branches: VCODE marks where these instructions occur in the instruction stream and, when the client indicates that code generation is finished, backpatches unresolved jumps.

Currently VCODE creates code one function at a time.[3] A sample VCODE specification to dynamically create a function that takes a single integer argument and returns its argument plus one is given in Figure 1.

This example illustrates a number of boilerplate issues: VCODE macro names are formed by prepending a v_ prefix to the base instruction and appending the type letter. If the instruction takes an immediate operand, the letter i is appended to the end result. For example, the VCODE instruction specifying "add integer immediate" is named v_addii (ADD Integer Immediate).

The following actions occur during dynamic code generation:

1. Clients begin dynamic code generation of a new function with a call to v_lambda, which takes a type string listing the function's incoming parameter types, a vector of registers to put these parameters in, a boolean flag indicating whether the function is a leaf procedure, and finally a pointer to memory where the code will be stored and the size of this memory. The number and type of parameters a dynamically generated function takes do not have to be fixed at static compile time but, rather, can be determined at runtime.

[3] In the future, this interface will be extended so that clients can create several functions simultaneously.

2. In v_lambda, VCODE uses the parameter type string and the machine's calling conventions to compute where the function's incoming parameters are: if the arguments are on the stack, VCODE will, by default, copy them to a register. At this point, VCODE also reserves space for prologue code. Control is then returned to the client to begin code generation.

3. The client uses VCODE macros to dynamically generate code. During code generation the client can allocate a number of VCODE objects: registers (using v_getreg and v_putreg), local variables (using v_local), and labels (using v_genlabel). After all code has been generated, the client calls v_end to return control back to VCODE.

The VCODE backend performs rudimentary delay slot scheduling across jumps and strives to keep parameters in their incoming registers. The result is reasonably efficient code, as can be seen in the MIPS code generated by VCODE for plus1:

```
addiu   a0, a0, 1   # add 1 to argument (passed in a0)
j       ra          # jump to the return address
move    v0, a0      # delay slot: move result to the return register v0
```

For improved efficiency, VCODE provides mechanisms that clients can use to target specific registers (such as the register used to return results). For simplicity we do not present them here.

5 Functions

VCODE generates code a function at a time. Each function specification is bracketed by calls to v_lambda and v_end. v_lambda begins a function's specification; it moves incoming parameters into registers, initializes the function's context (e.g., by clearing linkage and register allocation data structures) and creates its activation record. v_end finalizes code generation for a procedure; typically this activity is limited to linking, inserting prologue and eplilogue instructions to save and restore callee-saved registers, and flushing the necessary portions of instruction and data caches. v_end returns a union of function pointers.

5.1 Interface

void v_lambda(char **name*, char **fmt*, v_reg_type **args*, int *leaf*, v_code **ip*, int *nbytes*)

> *v_lambda* begins code generation for a function *name* that takes incoming parameters whose number and type of arguments are specified by the *printf*-style format string, *fmt*. *fmt* is formed by, for each incoming parameter, preprending a % to the parameter's type initial (i.e., "%i" for integer, "%u" for unsigned, etc.) and concatentating these type strings in order. The registers these parameters are loaded into are returned to the client in *args*, which must contain space for as many parameters as are specified by *fmt*. *leaf* is an enumerated type (V_LEAF or V_NLEAF) used by the client to indicate whether or not the procedure is a leaf procedure. *ip* points to storage to hold the dynamically generated code and *nbytes* gives the size of this storage. Currently, the vector must be "big enough" to hold all the generated instructions (a future VCODE implementation will remove this restriction).

union v_fp _end(int *nbytes*)

> *v_end* compiles the current function under construction and returns a pointer (wrapped in a union) to the generated function. Client's access the function pointer of the required return type using its VCODE type abbreviation (e.g., i for a pointer to a function that returns an int, d for a function that returns a double, etc.). If *nbytes* is non-nil, VCODE will store the number of bytes consumed by generated code into it.

5.2 Examples

The main axis of use of v_lambda is specifying the number and type of parameters the function it creates takes. A simple example function that takes no arguments and returns the literal "0" is specified as follows:

```
#include <stdio.h>
#include "vcode.h"          /* This header file defines all vcode insns. */

int main(void) {
        static v_code insn[1000];      /* Memory to hold code in. */
        v_iptr ip;

        /* Create a leaf function taking no arguments that returns 0. */
        v_lambda("zero", "", 0, V_LEAF, insn, sizeof insn);
                v_retii(0);  /* Return Integer Immediate. */
        ip = v_end(0).i;      /* Compile and capture Integer function pointer. */
        printf("zero returned %d\n", ip());     /* Run it. */
        return 0;
}
```

This example is similar to the previous "hello" program with the addition that it introduces the return integer immediate instruction (v_retii).

As promised, v_lambda also can be used for functions that take arguments. A sample function that computes logical and could be coded as follows:

```
/* generate code to compute bitwise and: unsigned and(unsigned x, unsigned y) { return x & y; } */
v_uptr mk_and(void) {
        v_reg_type arg[2];      /* Two arguments. */

        /* Create and function that returns the result of anding its two unsigned inputs */
        v_lambda("and", "%u%u", arg, V_LEAF, (void *)malloc(512), 512);
                v_andu(arg[0], arg[0], arg[1]); /* And the two arguments */
                v_retu(arg[0]);                 /* Return the result. */
        return v_end(0).u;
}
```

and specifies it expects two unsigned inputs using the format string: "%u%u". The registers the arguments are stored in are accessed via the arg vector. arg[0] contains the register holding the first argument, arg[1] the second.

5.3 Details

The union returned by v_end is a union of commonly-used function pointer types. It allows clients to eliminate many of the casts they would otherwise need. It is structured as follows:

```
/*
 * v_fp: union of all vcode function pointer types (ignoring paramters).
 *
 * NOTE: to prevent C's default argument promotion these must be cast to having the
 * correct parameter types (e.g., char, short, float).
 */
union v_fp {
        void    (*v)();             /* void */
        int     (*i)();             /* int */
```

```
    unsigned (*u)();          /* unsigned */
    long (*l)();              /* long */
    unsigned long (*ul)();    /* unsigned long */
    void *(*p)();             /* pointer */
    float (*f)();             /* float */
    double (*d)();            /* double */
};
```

6 Storage

The VCODE storage classes are those of modern hardware: stack space and registers. Stack space is used to hold local variables; it is allocated using the v_local function. Since VCODE defines a load/store architecture, variables must be loaded into registers before being operated on. Registers are allocated using v_getreg and deallocated using v_putreg.

6.1 Register Interface

int v_getreg(v_reg_type **r*, int *type*, int *class*)

v_getreg allocates a register of *type* and stores its name into *r*. The *type* field is a VCODE enumerated type whose name is formed by capitalizing the *type*'s type initials and prepending a capital V_. For instance, an integer register is specified by V_I. *class* is an enumerated type to select what class a register should be allocated from. The register classes are either temporary registers (not persistent accross procedure calls) or variable registers (persistent across procedure calls). These classes are selected using the enumarated types V_TEMP and V_VAR. When the supply of registers of a given *type* and *class* is exhausted and *v_getreg* is unable to use the other class as a "stand-in", it will return an error code ("0"); otherwise it returns success ("1"). Currently, bogus arguments cause an assertion failure. A more desirable result would be to return an error condition code.

int v_putreg(v_reg_type *r*, int *type*)

v_putreg frees register *r* allocated by *v_getreg*. *r*'s type is indicated by *type*.

int v_save[type-initial](v_reg_type *r*)

v_save is used to preserve a temporary register across a procedure call. It will save *r* onto stack space preallocated by the VCODE activation record manager. If the register is not a temporary (e.g., if VCODE has used a variable register to stand in for a temporary) this instruction does not save the register.

int v_restore[type-initial](v_reg_type *r*)

v_restore restores a register saved by *v_save*. VCODE provides *v_save* and *v_restore* in order to derive more semantic information than would be possible if clients manually saved and restored registers across procedure calls. This information is important since, in many cases, no action is needed to save and restore a register (e.g., if register windows are used, or a "temporary" has been allocated using a callee-saved register). Without the extra information provided by v_save and v_restore, the backend would not be able to omit saves, since it could not be sure that the client was not intentionally updating memory to the value residing in the register. I.e., the aliasing problem would force gratuitous saves.

6.2 Local interface

int v_local(int *type*)

v_local allocates space for a variable of *type* and returns an integer corresponding to its stack offset. Accesses to this variable use the VCODE-defined v_lp register as a base and the returned integer as an offset. (We might be exposing too much here; may make formal instructions for accessing locals.)

int v_localb(unsigned *size*)

v_localb allocates space for a aggregate structure of *size* bytes. Accesses to this variable use the v_lp register as a base and the returned integer as an offset.

6.3 Examples

Armed with registers and the ability to dynamically generate code we can perform some interesting optimizations. The one we look at here is a simplified version of a partial-evaluation classic: the power function ($f(x, n) = x^n$). The main optimization we perform is the elimination of looping overhead by specializing the function to the specific n its input will be raised too (other tricks are possible as well, for simplicity we leave them to the reader).

```
/* Power: raise base to n-th power: n > 0; specialized for a runtime constant n. */
v_fptr specialize_power(int n) {
	v_reg_type x, sum;

	v_lambda("power", "%d", &x, V_LEAF, malloc(512), 512);
	{
		int i;

		/* Allocate accumulator */
		v_getreg(&sum, V_D, V_TEMP);
		v_movf(sum, x);		/* initialize sum */

		/* Specialize power to x^n by unrolling the loop to multiply
		   x n times: (x * x * ... * x). */
		for(i = 0; i < n - 1; i++)
			v_mulf(sum, sum, x);

		v_retf(sum);	/* return x ^ n */
	}
	return v_end(0).f;	/* return pointer to result. */
}
```

A simple client that creates two power instances, the first specialized to 3, the second to 10, would be as follows:

```
/* Simple program to specialize power to compute x^3. */
int main(void) {
	/* NOTE: we must cast the function pointer to specify
	   that the function expects a floating point parameter.
	   Otherwise C will, by default, treat the value as an
	   integer. */
	float (*pow3)(float) = (float (*)(float))specialize_power(3);
	float (*pow10)(float) = (float (*)(float))specialize_power(10);
```

```
    printf("9.0^3 = %f, 2.0^10 = %f\n", pow3(9.0), pow10(2.0));
    return 0;
}
```

Another interesting example is the use of VCODE to create functions that provide infinite streams. The trick it uses is to create multiple instances of the same function with private internal state for each. For instance, the following procedure, mk_stream, dynamically generates a function that returns the next element in the set of non-zero integers every time it is evaluated:

```
/* Create a leaf function that will return the next integer from 1 .. 2^32 */
v_uptr mk_stream(void) {
    /* Allocate space for code */
    v_code *code = (void *)malloc(512);

    v_lambda("int-stream", "", 0, V_LEAF, code, 512);
    {
        /* state is initially set to ''0'' */
        void *state = calloc(1, sizeof(unsigned));
        v_reg_type state_r, temp;

        /* allocate register to hold pointer to per-function storage. */
        v_getreg(&state_r, V_P, V_TEMP);
        /* allocate scratch register */
        v_getreg(&temp, V_P, V_TEMP);

        /* load pointer to state into register */
        v_setp(state_r, state);        /* SET Pointer */
        /* Load current state value */
        v_ldui(temp, state_r, 0);      /* LoaD Unsigned Immediate */
        /* Add 1 to it. */
        v_addui(temp, temp, 1);        /* ADD Unsigned Immediate */
        /* Store the new value into state */
        v_stui(temp, state_r, 0);      /* STore Unsigned Immediate */
        /* Return new value */
        v_retu(temp);                  /* RETurn Unsigned */
    }
    /* Compile and return an unsigned function pointer. */
    return v_end(0).u;
}
```

To manipulate stream data the function needs two registers: temp, a scratch register, and state_r, used to hold a pointer to the per-function state. These registers are allocated using v_getreg by giving their type (unsigned and void * respectively or, in VCODE argot, V_U and V_P) and a register class, in this case we only require temporary registers that are not persistent across procedure calls (V_TEMP).

The function works by allocating per-function state on the heap (using calloc). It then hard-codes a pointer to this state in the created function:

```
    v_setp(state_r, state);   /* load address of state into state_r */
```

v_setp generates code to load the runtime address pointed to by state into the register state_r allocated using the VCODE function v_getreg (discussed in Section 6).

To implement the stream, we load the value in state into a scratch register, increment it by one and then store it back into state.

```
v_ldui(temp, state_r, 0);
v_addui(temp, temp, 1);
v_stui(temp, state_r, 0);
```

We then generate code to return the computed value (stored in temp) to the caller:

```
v_retu(temp);
```

Finally, we compile the function and return a pointer to it:

```
v_end(0).u;
```

The function can now be called: each invocation will return the next integer in the series. Because we have embedded the stream state within the function itself, concurrent streams can be cleanly provided.

A sample user of this ability could be a sieve-based prime number tester. We provide a simpler client that creates two concurrent streams and prints out their first initial values:

```
int main(void) {
        v_uptr stream1, stream2;

        /* Create two streams */
        stream1 = mk_stream();
        stream2 = mk_stream();

        /* Demonstrate that they are independent */
        printf("stream 1 = [%d, %d ..]\n", stream1(), stream1());
        printf("stream 2 = [%d, %d ..]\n", stream2(), stream2());
        return 0;
}
```

6.4 Discussion

v_lambda treats all incoming argument registers as callee-saved registers: i.e., the backend will guarantee that they reside in persistent registers. Therefore, since incoming parameters are forcibly kept in registers, it is a good idea to pass as parameters only those parameters you actually want in registers! (This implementation feature will likely be altered in the future.)

To sum it up:

1. Variable registers (V_VAR) are persistent across function calls, as are incoming parameter registers.
2. Temporary registers (V_TEMP) are not persistent across function calls.
3. VCODE provides only as many registers as provided by the underlying machine.
4. Temporary registers should be saved and restored across function calls using the v_save and v_restore functions.

6.5 Details

A VCODE register is, by default, represented as a structure:

```
typedef struct { unsigned reg; } v_reg_type;
```

Using a structure allows C's weak type-checking to differentiate registers from integers. If registers were represented as integers, it would be easy to mistakenly use an integer in place of a register. The downside of this trick is that some compilers are stupid and don't allocate registers to word-sized structures. To coddle such implementations clients can define the preprocessor symbol __fast__ in vcode.h, which will cause registers to represented as integers.

An enumeration of valid VCODE types are :

```
/* vcode types */
enum {
    V_C,    /* char */
    V_UC,   /* unsigned char */
    V_S,    /* short */
    V_US,   /* unsigned short */
    V_I,    /* int */
    V_U,    /* unsigned */
    V_L,    /* long */
    V_UL,   /* unsigned long */
    V_P,    /* pointer */
    V_F,    /* floating */
    V_D,    /* double */
    V_V,    /* void */
    V_B,    /* block structure */
    V_ERR,  /* error condition */
};
```

Functions that use VCODE enumerated types use these names.

7 Control

VCODE provides control primitives similar to those found on modern hardware: branches, jumps, and jump-and-link. Jumps can be to registers, labels, and absolute addresses. Branches can be to labels. VCODE provides support for creating labels (using v_genlabel), placing labels in the instruction stream (using v_label), and creating jump tables of label addresses (using v_dlabel).

7.1 Interfaces

v_label_type v_genlabel(void)

> v_genlabel creates a label. The label can then be positioned in the instruction stream and used as a target of jumps and branchs.

void v_label(v_label_type *l*)

> v_label marks the current instruction stream position as the location of label *l*. *l* must have been allocated by v_genlabel.

void v_dlabel(v_code **addr*, v_label_type *l*) v_dlabel marks *addr* as a location to place the absolute address of *l* in (this value will be written after *l* is positioned in the instruction stream using v_label). v_dlabel is primarily used to create jump tables.

7.2 Examples

A simple example function to test whether its first argument is equal to five or not:

```
#include <stdio.h>
#include "vcode.h"          /* This header file defines all vcode insns. */

int main(void) {
        static v_code insn[1000];       /* Memory to hold code in. */
        v_vptr vp;
        v_reg_type x;

        /* Test branch instruction */
        v_lambda("branch", "%i", &x, V_NLEAF, insn, sizeof insn);
        {
                v_label_type true, end;

                true = v_genlabel();    /* allocate two labels */
                end = v_genlabel();

                /* test whether x is equal to 5 */
                v_beqii(x, 5, true);
                        v_scallv((v_vptr)printf, "%P", "Arg is not equal to 5\n");
                        v_jv(end);      /* jump over else */
                v_label(true);
                        v_scallv((v_vptr)printf, "%P", "Arg is equal to 5\n");
                v_label(end);
                v_retv();
        }
        vp = v_end(0).v;
        vp(5);          /* test */
        vp(6);
        return 0;
}
```

The example first allocates two labels:

```
true = v_genlabel();
end = v_genlabel();
```

These labels can now be used as targets of branch and jump instructions. Jumps to labels are of type v:

```
v_jl(end);      /* jump over else */
```

Figure 2 introduces the use of v_dlabel to create a (one element) jump table and the VCODE jump instructions v_jv and v_jalp. These mechanisms are not directly available in C and can be very powerful. For instance, clients can use them to implement their own activation record management and argument passing conventions on top of VCODE. Additionally, VCODE's low-level jump instructions allow it to be used to dynamically compile finite state machines at runtime. We illustrate their use with a contrived program that uses the VCODE jump-and-link instruction v_jalp to jump to a target address, print a message and then jumps back to instruction following the jump-and-link statement (the program counter value of this instruction is written into the register rr as a side-effect of v_jalp).

The example uses two registers, rdp and rr. These registers hold pointers (and so are of type V_P); since their values must be preserved across calls to printf, they are allocated as persistent registers (V_VAR).

```
v_getreg(&rdp, V_P, V_VAR);
v_getreg(&rr, V_P, V_VAR);
```

The statement

```
v_dlabel(&linked_addr, l);
```

tells VCODE to write the address that l is positioned at into linked_addr. To locate l in the instruction stream we use the v_label function:

```
v_label(l);
```

which associates the label with instruction address at this position. This address can then be loaded into a register and used as the destination address of a jump instruction:

```
v_jalp(rr, rdp);     /* jump to the address in rpd; the return address is written to rr */
```

This statement jumps to the value contained in rdp and writes the program counter of the next instruction into the register rr. The callee can then jump back to the callsite by performing a jump using the rr register:

```
v_jp(rr);     /* jump back to caller */
```

In this example, we did not need a jump table since we know the destination of the jump statically. We could have jumped to a label instead:

```
v_jalv(rr, l);
```

By convention, jumps that use labels are of type void (V_V), jumps that use addresses are of type void * (V_P).

7.3 Details

Labels are integers wrapped in a structure:

```
typedef struct { unsigned short label; } v_label_type;
```

8 Function calls

Function calls are the richest primitive supplied by VCODE. There are two types of function calls. The first, v_scall, is for simple, static calls whose number and type of arguments are known at compile-time. The second, v_ccall, allows argument lists to be constructed at runtime. This ability is crucial for constructing calls to functions whose type signatures are not known until runtime.

```
#include <stdio.h>
#include "vcode.h"              /* This header file defines all vcode insns. */

int main(void) {
        static v_code insn[1000];       /* Memory to hold code in. */

        /* Test jump and link instruction */
        v_lambda("jump-and-link", "", 0, V_NLEAF, insn, sizeof insn);
        {
                static v_code *linked_addr;
                v_reg_type rdp, rr;
                v_label_type l;

                /* Allocate two registers persistent accross procedure calls. */
                v_getreg(&rdp, V_P, V_VAR);
                /* Allocate register to hold return pointer */
                v_getreg(&rr, V_P, V_VAR);

                l = v_genlabel();       /* Allocate label */
                v_dlabel(&linked_addr, l); /* mark memory to hold target address */
                v_setp(rdp, &linked_addr); /* Load address */
                v_ldpi(rdp, rdp, 0);

                v_scallv((v_vptr)printf, "%P", "Jumping!\n");
                v_jalp(rr, rdp);        /* Jump to it. */

                v_scallv((v_vptr)printf, "%P", "Returning!\n");
                v_retv();               /* Done */

                v_label(l);
                        v_scallv((v_vptr)printf, "%P", "Jumping back!\n");
                        v_jp(rr);       /* Jump back to caller. */
        }
        v_end(0).v();
#if 0
{
        v_vptr vp = v_end(0).v;
        v_dump(vp);
        vp();
}
#endif

        return 0;
}
```

Figure 2: Use the VCODE jump-and-link instruction to return to callsite.

8.1 Interface

void v_scallv(v_vptr *ptr*, char **fmt*, ...)
v_reg_type v_scalli(v_iptr *ptr*, char **fmt*, ...)
v_reg_type v_scallu(v_uptr *ptr*, char **fmt*, ...)
v_reg_type v_scalll(v_lptr *ptr*, char **fmt*, ...)
v_reg_type v_scallul(v_ulptr *ptr*, char **fmt*, ...)
v_reg_type v_scallp(v_pptr *ptr*, char **fmt*, ...)
v_reg_type v_scallf(v_fptr *ptr*, char **fmt*, ...)
v_reg_type v_scalld(v_dptr *ptr*, char **fmt*, ...)

> v_scall dynamically generates a function call to *ptr*; the call will be constructed according to the type-signature partially specified by *fmt*. Arguments to the call (VCODE registers or constants) are provided after *fmt*. With the exception of v_scallv, scall returns the register that the function's result is stored in. This register is, for obvious reasons, a temporary: *clients must save it before calling another function*. In fact, given that some VCODE instructions map to subroutine calls (e.g., div on both SPARC and Alpha architectures) this value should be immediately moved. (Actually, providing this register probably over-constrains a VCODE implementation; this function might be modified.)

void v_ccallv(struct v_cstate **c*, v_vptr *ptr*)
v_reg_type v_ccalli(struct v_cstate **c*, v_iptr *ptr*)
v_reg_type v_ccallu(struct v_cstate **c*, v_uptr *ptr*)
v_reg_type v_ccalll(struct v_cstate **c*, v_lptr *ptr*)
v_reg_type v_ccallul(struct v_cstate **c*, v_ulptr *ptr*)
v_reg_type v_ccallp(struct v_cstate **c*, v_pptr *ptr*)
v_reg_type v_ccallf(struct v_cstate **c*, v_fptr *ptr*)
v_reg_type v_ccalld(struct v_cstate **c*, v_dptr *ptr*)

> v_ccall is used in conjunction with v_push_arg and v_push_init to dynamically construct a function call to *ptr* using the argument list that was built up in *c*. The number and type of arguments to *ptr* do not have to be known statically at compile time but, rather, can be built dynamically at runtime.

void v_push_init(struct v_cstate **c*)

> v_push_init initializes the call closure *c*. This closure is used to track for which call an argument list is being constructed. It is passed to all argument routines and to the v_ccall instructions.

void v_push_argi(struct v_cstate **c*, v_reg_type *r*)
void v_push_argu(struct v_cstate **c*, v_reg_type *r*)
void v_push_argl(struct v_cstate **c*, v_reg_type *r*)
void v_push_argul(struct v_cstate **c*, v_reg_type *r*)
void v_push_argp(struct v_cstate **c*, v_reg_type *r*)
void v_push_argf(struct v_cstate **c*, v_reg_type *r*)
void v_push_argd(struct v_cstate **c*, v_reg_type *r*)
void v_push_argii(struct v_cstate **c*, int *imm*)
void v_push_argui(struct v_cstate **c*, unsigned *imm*)
void v_push_argli(struct v_cstate **c*, long *imm*)
void v_push_arguli(struct v_cstate **c*, unsigned long *imml*)
void v_push_argpi(struct v_cstate **c*, void * *imm*)
void v_push_argfi(struct v_cstate **c*, float *imm*)
void v_push_argdi(struct v_cstate **c*, double *imm*)

> v_push_arg is used to allocate arguments "on-the-fly" for the call associated with *c*. Register instructions specify a register, *r*. Immediate values are provided in *imm*.

8.2 Examples

The benefits of dynamic code generation extend beyond efficiency. A general technique it enables is the hiding of information from clients by associating a function with internal state not visible to the caller (also known as *currying*). This technique provides information hiding while at the same time allowing functions to be parameterized with data (the previous stream example uses a varient of this idea). For instance, Figure 3 presents code that creates a function to write to a specific (fake) network connection; it allows the connection control block to be hidden from clients while still allowing operations on the connection (such as write in this example) to be parameterized with per-connection data.

As a bit of syntactic sugar, and to allow parameter registers to be easily targeted, clients are allowed to pass constants directly; to indicate that they are doing so, the type is capitalized in the type string. mkwrite uses this mechanism to dynamically bind a pointer to tcb in the call to write:

```
/* generate call to write: %P indicates the first argument (tcb) is a constant */
v_scalli(v_iptr)write, "%P%p%p", tcb, arg[0], arg[1]);
```

This feature can make the client's noticeably code cleaner.

Frequently, it is the case that the number and type of arguments are *not* known statically. An example domain where this situation is common is "compiling interpreters": interpreters that interactively consume user code and compile it on-the-fly to machine code. Obviously, no restrictions can be made *a priori* as to the possible number and type of functions that the user will key in. To support this requirement, VCODE provides a more sophisticated call mechanism that takes in a argument list created via the v_push command:

```
#include <stdio.h>
#include "vcode.h"          /* This header file defines all vcode insns. */

int main(void) {
        struct v_cstate c;

        /* Create a function to print standard greeting. */
        v_lambda("hello-world", "", 0, V_NLEAF, malloc(512), 512);
                v_push_init(&c);
                v_push_argpi(&c, "hello, world\n");
                v_ccallv(&c, (v_vptr)printf);
        v_end(0).v();   /* Compile & call */
        return 0;
}
```

Before being used, the call context (c) is initialized using v_push_init:

```
v_push_init(&c);        /* initialize context */
```

Arguments can then loaded into argument registers, in order, using this context. In our example, we load a single argument, a pointer to a string constant:

```
v_push_argpi(&c, "hello, world\n");
```

After all arguments have been pushed, the function can then be called:

```
v_ccallv(&c, (v_vptr)printf);
```

Note that the "push" in v_arg_push refers only to the fact that arguments are placed in order in the call context data structure. It is *not* an indication that VCODE uses a broken calling convention of pushing all arguments on the stack.

```
#include <stdio.h>
#include "vcode.h"              /* This header file defines all vcode insns. */

/* Fake connection */
struct tcb {
        unsigned short dst_port;
        unsigned short src_port;
        /* ... */
};

typedef int (*write_ptr)(char *, int);

static int write(struct tcb *tcb, char *msg, int nbytes) {
        printf("dst port = %d, src port = %d\n", tcb->dst_port, tcb->src_port);
        return 1;
}

/* Create a new function to call a write function with
   the given control block hardwired as its first argument. */
write_ptr mkwrite(struct tcb *tcb) {
        v_reg_type arg[2];      /* two arguments */
        v_reg_type res;         /* function result. */


        /* After currying its first argument, write takes a pointer (msg) and an int (nbytes) */
        v_lambda("curry-write", "%p%i", arg, V_NLEAF, malloc(512), 512);
                /* generate call to write: %P indicates the first argument (tcb) is a constant */
                res = v_scalli((v_iptr)write, "%P%p%u", tcb, arg[0], arg[1]);
                v_reti(res);    /* return result of calling write */
        return v_end(0).i;
}

int main(void) {
        struct tcb t1, t2;
        write_ptr write1, write2;

        t1.dst_port = 1;
        t1.src_port = 2;
        write1 = mkwrite(&t1);

        t2.dst_port = 3;
        t2.src_port = 4;
        write2 = mkwrite(&t2);

        /* Test write */
        write1(0,0);
        write2(0,0);

        return 0;
}
```

Figure 3: Dynamically curry write

9 Advanced Features

This section looks at some of the more advanced VCODE features.

9.1 Using gdb to debug code

Currently, VCODE has no symbolic debugger. As a result, clients are either reduced to using printf statements or debugging the machine-specific generated code. The latter method can be done using the gdb debugger, which provides a number of useful features:

- si - single step to the next instruction.
- ni - single step to the next instruction (skips calls).
- disass ptr - disassemble the code pointed to by ptr
- disass begin-addr end-addr - disassemble the instructions from begin-addr upto end-addr. A common idiom is disass $pc $pc + 64 to disassemble the instructions from the current pc to some end value (64 is arbitrary).
- p $reg-name - print the contents in reg-name (names are prefixed by the $ symbol). For instance, to print out the value of the stack pointer: p $sp.

The typical usage pattern debug dynamic code is to set a breakpoint at its callsite and then issues a series of si commands its code is executed (because calls to generated code can load argument registers and perform other operations, it will typically take more than one si command before it finally reaches the generated code).

A nice feature of gdb is that hitting $return$ will cause the previous command to be executed.

9.2 Disassembler interface

void v_dump(void **code*)

v_dump dumps the instruction stream pointed to by *code* to stdout. It can be used to examine the quality of VCODE-generated code or for debugging. Future versions will (optionally) dump a symbolic VCODE representation.

9.3 Instruction scheduling interface

#define v_schedule_delay(*branch-insn*, *delay-insn*)

v_schedule_delay is used to schedule delay slots in jumps and branchs. It is a macro that takes two VCODE instructions, *branch-insn* and *delay-insn* and, if the current machine has a delay slot and the delay instruction fits, places *delay-insn* in the delay slot after *branch-insn*. Otherwise it emits *delay-insn* and then *branch-insn*. *delay-insn* must not depend on being executed before or after the branch.

#define v_raw_load(*mem-insn*, *n*)

v_raw_load is used to schedule delay slots in load instructions. It is a macro that takes a VCODE instruction, *mem*, and the number of instructions before *mem* is used, *n*. It performs a memory operation without software interlocks for *n* instructions. If the load latency is greater than *n* instructions, the backend will insert nops.

```
/* Generate code to compute A = c * B.  c is treated as a
   runtime constant, and is strength-reduced by the vcode
   extension cmuli. */
void scale(int n, data_t **a, data_t **b, data_t c) {
        v_reg_type  dst,src, src_end, v0, v1, al[10];
        v_label_type  loop1;
        static unsigned insn[1024];
        v_vptr vp;

        /* simple unroll */
        assert(n ≥ 4 && (n % 4) == 0);

        v_lambda("foo", "%p%p", al, V_LEAF, insn, sizeof insn);

        /* dst = A,  and src = B; come in as parameters. */
        dst     = al[0];
        src     = al[1];
        if(!v_getreg(&src_end, V_P, V_TEMP))
                v_fatal("scale: out of registers\n");
        if(!v_getreg(&v0, V_US, V_TEMP))
                v_fatal("scale: out of registers\n");
        if(!v_getreg(&v1, V_US, V_TEMP))
                v_fatal("scale: out of registers\n");

        loop1   = v_genlabel();

        /* relies on the fact that the rows in a and b are allocated from contigous memory */
        v_raw_load(v_ldpi(src, src, (0)), 1);   /* perform loads without interlocks */
        v_raw_load(v_ldpi(dst, dst, (0)), 1);
        v_addpi(src_end, src, (n * n) * sizeof **a);

        v_label(loop1);
                  /* load 2 to get rid of delay slots */
                  v_raw_load(v_ldusi(v0, src, (0 * sizeof **a)), 1);
                  v_raw_load(v_ldusi(v1, src, (1 * sizeof **a)), 1);

                /* multiplies will be strength reduced */
                cmuli(v0, v0, c);
                v_addpi(dst, dst, (2 * sizeof **a));
                cmuli(v1, v1, c);
                v_stusi(v0, dst, -(2 * sizeof **a));
                v_addpi(src, src, (2 * sizeof **a));
        /* schedule delay slot instructions */
        v_schedule_delay(
                v_bltp(src, src_end, loop1),
                v_stusi(v1, dst, -(1 * sizeof **a))
        );

        v_retv();
        vp = v_end(0).v;
        if(disass)
                v_dump((void *)vp);
        vp(a,b);          /* Use it. */
}
```

Figure 4: Dynamically generate code to scale matrix by a runtime constant

9.4 Example

Figure 4 shows a sample program that computes $A = c * B$. For speed, it strength-reduces the multiplication by c to shifts and adds using a VCODE extension instruction and uses both v_raw_load and v_schedule_load to fill delay slots.

10 Discussion

10.1 Gotchas

There are a number of common errors made with VCODE.

- In C, implicit promotions happen in the absence of prototypes. Therefore, expecting a incoming parameter to be a float, but calling the generated code using a function-pointer that does not specify its argument types will be disastrous.
- VCODE format strings are derived from the VCODE type strings: *they do not match printf types*. In particular, integers are denoted by "%i" rather then "%d".

10.2 Todo

VCODE is a single-implementor system. As a result, there are more useful features than I have time to add (especially since my thesis work has nothing to do with dynamic code generation). I've taken a demand-driven approach to creeping featurism: if you need any of the following items, let me know.

1. Infinite registers.
2. Dynamic peephole optimizer/instruction scheduler.
3. Reduce the base executable size. This is primarily a matter of splitting VCODE up into more files so that the linker can deal (unix linkers are unbelievably stupid).
4. Introduce better sanity checking in registers to catch unallocated register uses.
5. Reintegrate new preprocessor (provides extensibility support and is much cleaner).
6. Write a debugger that presents symbolic VCODE.
7. Add more graceful error handling (currently, incorrect usage causes assertion failures).
8. Add marshalling and unmarshalling examples.
9. Grow instruction stream dynamically. This will require modifying the linker support to mark labels with offsets rather than absolute addresses. This is actually good, since it will require less space. Will also require care with modifications to the VCODE instruction pointer.
10. add linkage support so that functions can contain calls to as-yet-ungenerated functions (using a variant similar to label seems the best way).
11. Allow functions to be associated with a context so that we can create multiple functions at once.
12. Have VCODE perform optional storage management. Since we are working with large blocks of memory, should be able to specialize this to some degree.
13. Add a compaction call that will get rid of the extra space currently associtated with VCODE functions.
14. Tutorial is very incomplete. Add more exposition (grammer too).
15. Implement eager linkage.

16. Partially evaluate the header files to consume all the binary instruction names in order to reduce conflicts with clients.
17. x86 port.
18. Optimize leaf procedures on SPARC.
19. Common usage is to generate code, use it, generate new code, etc., constantly overwriting storage. Add support for this.
20. Do more sophisticated (i.e., fast) cache coherence management: track whether we have generated code into a given storage vector, and set up cache conflicts so that we can flush the icache without calling the OS.
21. Have link table grow automatically.
22. Alter v_lambda so that it can indicate that VCODE should not allocate parameters to registers. This might be difficult to support. Also, provide option to use temporaries instead of vars.
23. Modify argpush so that it defers argument loading of memory locations.
24. Add a set of macro's that will move an instruction up the instruction stream to fill any available nops (this is actually pretty easy, and would be a win on mips and sparc).
25. Take a union of all call registers that are used. Make the difference between this set and all possible call arguments available for allocation. This optimization is actually not too difficult.

10.3 Known Bugs

1. Concurrent construction of function calls is not supported (each construction will trash the other's registers).

References

[1] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[2] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10), October 1991.