VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System

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

Abstract

Dynamic code generation is the creation of executable code at runtime. Such "on-the-fly" code generation is a powerful technique, enabling applications to use runtime information to improve performance by up to an order of magnitude [4, 8, 20, 22, 23].

Unfortunately, previous general-purpose dynamic code generation systems have been either inefficient or non-portable. We present VCODE, a retargetable, extensible, very fast dynamic code generation system. An important feature of VCODE is that it generates machine code "in-place" without the use of intermediate data structures. Eliminating the need to construct and consume an intermediate representation at runtime makes VCODE both efficient and extensible. VCODE dynamically generates code at an approximate cost of six to ten instructions per generated instruction, making it over an order of magnitude faster than the most efficient general-purpose code generation system in the literature [10].

Dynamic code generation is relatively well known within the compiler community. However, due in large part to the lack of a publicly available dynamic code generation system, it has remained a curiosity rather than a widely used technique. A practical contribution of this work is the free, unrestricted distribution of the VCODE system, which currently runs on the MIPS, SPARC, and Alpha architectures.

1 Introduction

Dynamic code generation is the generation of machine code at runtime. It is typically used to strip a layer of interpretation by allowing compilation to occur at runtime. One of the most well-known applications of dynamic code generation is by interpreters that compile frequently used code to machine code and then execute it directly [2, 6, 8, 13]. Hardware simulators and binary emulators can use the same techniques to dynamically translate simulated instructions to the instructions of the underlying machine [4, 22, 23]. Runtime partial evaluation also uses dynamic code generation in order to propagate runtime constants to feed optimizations such as strength reduction, dead-code elimination, and constant folding [7, 20].

Unfortunately, portability and functionality barriers limit the use of dynamic code generation. Because binary instructions are generated, programs using dynamic code generation must be retargeted for each machine. Generating binary instructions is non-portable, tedious, error-prone, and frequently the source of latent bugs due to boundary conditions (e.g., constants that don't fit in immediate fields) [21]. Many of the amenities of symbolic assemblers are not present, such as detection of scheduling hazards and linking of jumps to target addresses. Finally, dynamic code generation requires a working knowledge of chip-specific operations that must be performed, the most common being programmer maintenance of cache coherence between instruction and data caches.

Once the goals of portability and usability have been satisfied, the main focus of any dynamic code generation system must be speed, both in terms of generating code (since code generation occurs at runtime) and in terms of the generated code.

This paper describes the VCODE dynamic code generation system. The goal of VCODE is to provide a portable, widely available, fast dynamic code generation system. This goal forces two implementation decisions. First, to ensure wide availability across different languages and dialects with modest effort, VCODE cannot require modifications to existing compilers (or require its own sophisticated compiler). Second, in order to generate code efficiently VCODE must generate code *in place*. I.e., VCODE must dynamically generate code without the luxury (and expense) of representing code in data structures that are built up and consumed at runtime. The main contribution of VCODE is a methodology for *portably* generating machine code at speeds that formerly required sophisticated compiler support or the use of hand-crafted, non-portable code generators.

The VCODE machine-independent interface is that of an idealized load-store RISC architecture. This low-level interface allows VCODE to generate machine code from client specifications at an approximate cost of six to ten instructions per generated instruction. This overhead is roughly equivalent to that of a highly tuned, non-portable dynamic code generator (or faster; compare [21]). Furthermore, the low-level instruc-

This work was supported in part by the Advanced Research Projects Agency under Contract N00014-94-1-0985. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

To appear in the *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, May 1996, Philadelphia, PA.

tion set can be used by clients to write portable VCODE that translates to high-quality code. VCODE is extensible, allowing clients to dynamically modify calling conventions and register classifications on a per-generated-function basis; it also provides a simple modular mechanism for clients to augment the VCODE instruction set. Finally, the VCODE system is simple to retarget, typically requiring one to four days to port to a RISC architecture. VCODE currently runs on MIPS, SPARC, and Alpha processors.

As discussed above, VCODE generates code in place. Eliminating the need to build and then consume an intermediate representation at runtime has powerful effects. Code generation is more efficient, since intermediate data structures are not constructed and consumed. Elimination of intermediate structures also removes the need for VCODE to understand instruction semantics. As a result, extending the VCODE instruction set is simple, usually requiring the addition of a single line to the VCODE machine specification.

VCODE is a *general-purpose* dynamic code generation system in that it allows clients to portably and directly construct arbitrary code at runtime. It is the fastest such system in the literature (by more than an order of magnitude). It is also the first general-purpose system to generate code without the use of an intermediate representation; the first to support extensibility; and the first to be made publicly available.

This paper is structured as follows: we discuss related work in Section 2. We provide an overview of the system, retargeting, and some details of the client/VCODE interface in Section 3. We measure the performance of three experimental clients in Section 4. Section 5 presents some key implementation details and Section 6 reports on our experiences using VCODE. Finally, we conclude in Section 7.

2 Related Work

Dynamic code generation has a long history. It has been used to increase the performance of operating systems [20], windowing operations [18], dynamically typed languages [2, 13, 6], simulators [23] and matrix manipulations [10]. In [14], Keppel, Eggers and Henry survey many advantageous uses for dynamic code generation.

ParcPlace sells an implementation of Smalltalk-80 that uses a dynamic code generator for SPARC, Motorola 68k, PowerPC, Intel x86, and other architectures. Unlike VCODE, their system is designed specifically for the compilation of Smalltalk-80, and not as a stand-alone system for dynamic code generation.

Engler and Proebsting [10] describe DCG, a general-purpose dynamic code generation system. VCODE grew out of my experiences building DCG and the subsequent use of DCG in building a compiler for the ‘C language [7]. Compared to DCG, VCODE is both substantially simpler and approximately 35 times faster. Both of these benefits come from eschewing an intermediate representation during code generation; in contrast, DCG builds and consumes IR-trees at runtime. VCODE also provides an extensible framework and generates more efficient code than DCG.

Engler, Hsieh, and Kaashoek [7] describe the language ‘C (*tick C*), a superset of ANSI C that is designed for the high-level, efficient, and machine-independent specification of dynamically generated code. Their implementation uses the DCG dynamic code generation system that, as we described above, is both substantially less efficient and more complex than VCODE. Poletto, Engler, and Kaashoek [19] describe a reimplementation of ‘C that uses VCODE as its target machine. As a result, ‘C can automatically generate code for any architecture VCODE has been ported to, and gains the advantages of VCODE: fast code generation and efficient generated code. ‘C and VCODE are complementary. The primary advantage of VCODE over ‘C is that VCODE is, in principle, language independent; in contrast, since it is a language, ‘C requires relatively sophisticated modifications to existing compilers. Also, VCODE’s low-level nature allows greater control over low-level details (calling conventions, register names, etc.).

VCODE is a manual code generation system. An alternative approach is to dynamically generate code automatically [5, 15, 16]. While an automatic system can be easier to use, it does require complex compiler support and can be less applicable than a manual system. The reason for reduced applicability is that automatic systems are primarily *users* of dynamic code generation rather than *providers* of it. In contrast, VCODE clients control code generation and can create arbitrary code at runtime. For instance, clients can use VCODE to dynamically generate functions (and function calls) that take an arbitrary number and type of arguments, allowing them to construct efficient argument marshaling and unmarshaling code [7]. It does not seem possible to efficiently perform such operations with current automatic systems [5, 15, 16].

Leone and Lee [15] describe an interesting automatic dynamic code generation system that performs compile-time specialization of a primitive functional language. Recently, they have extended their compiler, FABIUS, to accept a functional subset of ML [16]. FABIUS generates code quickly by using techniques developed by programmers to dynamically generate code “by hand”: dynamic code is emitted by inline expanded macros that create instructions whose operand register names are determined at static compile time. In contrast, VCODE provides a new technique for portably generating code at equivalent speeds without the support of a sophisticated compiler. As a result, VCODE can be used, practically speaking, in more arenas than FABIUS (e.g., in the context of a pointer and side-effect rich language such as C).

Another interesting automatic code generation system is Tempo [5], a general-purpose dynamic specializer for C. Tempo can be easier to use than VCODE, but like other automatic systems, it requires complex compiler support and can be less applicable. For example, the scope of Tempo’s optimizations is limited by the usual challenges C presents to optimizing compilers (e.g., unrestricted aliasing).

Ramsey and Fernandez have developed a tool kit for the concise specification of functions to emit and disassemble machine code [21]. Like VCODE, their system can be used to dynamically generate code quickly, is extensible, and is freely distributed. Unlike VCODE, their system’s client interface is not portable and, therefore, requires clients be rewritten for each new architecture.

3 System Overview

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 mapping is direct enough that most VCODE instructions are translated to their machine code equivalents “in-place” in client code.

An important benefit of VCODE’s in-place code generation is that it consumes little space. Other than the memory needed to store emitted instructions, VCODE need only store pointers to labels and unresolved jumps. At a cost of a few words per label, this is a relatively insignificant amount of memory. In contrast, a system that uses intermediate data structures requires space proportional to the number of instructions.

VCODE has been designed with the main aim of reconciling the conflicting goals of fast code generation and efficient generated code. VCODE achieves these goals via a low-level interface (i.e., the assembly language of an idealized RISC architecture). This interface has three main benefits. The first is that it allows clients to perform many of the expensive code generation operations (notably, virtual register allocation) at static compile time, leaving VCODE to concentrate on the simple job of translating its instruction set into machine code, a process that is neither difficult nor expensive.

The second benefit is that it allows code to be generated quickly without the need for compiler support. This characteristic has three desirable features. First, it makes the VCODE system simple to implement, which has obvious pragmatic results. Second, and more subtly, the close match between VCODE instructions and modern architectures allows VCODE to generate code in "zero passes": VCODE instructions are translated directly to the machine instructions that they correspond to. In some sense, code generation has been replaced with transliteration. Finally, since VCODE does not need compiler support, it is (in principle) language independent.

The final benefit is that VCODE's low-level interface allows code to be written that is not possible from within a higher-level language such as C. For example, VCODE clients can portably generate function calls "on-the-fly" and access instructions that have no natural higher-level idioms such as prefetching, branch prediction, and byte swapping.

3.1 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 [12].

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.

3.2 Client/VCODE interface

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

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

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	jump to immediate, register, or label
jal	v	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.

```
typedef int (*iptr)(int);
/* Called at runtime to create a function which returns its argument + 1. */
iptr mkplus1(struct v_code *ip) {
   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);

   /* 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();
}
```

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

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. 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.
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.
4. VCODE then backpatches prologue and epilogue code, links the client code, and, if necessary, ensures instruction and data cache coherency. VCODE then returns a pointer to the generated code to the application. This pointer must be cast to the appropriate function pointer type before being used.
5. The client can then run the dynamically generated code.

The VCODE backend performs rudimentary delay slot scheduling 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:

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

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.

3.3 Retargeting VCODE

Retargeting VCODE involves (1) constructing macros to generate executable code for each machine instruction to be used, (2) mapping the core VCODE instruction set onto these macros, and (3) implementing the machine's default calling conventions and activation record management. Generating the code to emit binary instructions can be done using either VCODE's preprocessor or programs such as the New Jersey Toolkit [21]. Since the VCODE core is small and simple, mapping its instructions to their corresponding binary emitters is straightforward. To aid this process, we have developed a concise preprocessor specification language in the spirit of Fraser [11] that handles much of the details of this mapping. Complete mapping specifications for the MIPS, SPARC, and Alpha architectures take approximately 40–100 lines each. Finally, the construction of calling conventions and activation record management can typically be based on existing code. For instance, the Alpha port of VCODE is largely based on the MIPS port. To aid in the retargeting process VCODE includes a script to automatically generate regression tests for errors in instruction mappings and calling conventions. As a result of VCODE's simplicity and porting assistance, a RISC retarget typically takes one to four days.

The VCODE instruction set is heavily RISC based. This model can conflict with the underlying hardware if its architecture is very different. Surprisingly, there is no real conflict between VCODE's interface and that of the most widely used CISC on the market, the x86. The main conflict that could arise from mapping VCODE instructions to the x86 integer instruction set is the x86's lack of registers. However, the x86 can use memory operands in instructions instead of registers, with little to no loss of performance. Such an ability can be used to, in effect, support an unlimited virtual register set.

The most serious conflict with the VCODE architecture arises on stack-based architectures. Generating code on a stack architecture using VCODE seems to require a post-construction pass over the generated code to couch VCODE register names in terms of stack positions. Fortunately, stack architectures are relatively rare.

4 Experimental Clients

We discuss three experimental clients. The first client is a compiler for the ‘C language that demonstrates VCODE's viability as a code generation substrate [19]. The last two are network subsystems within the Aegis exokernel operating system [8]. They demonstrate VCODE's suitability for use in an operating system context, and general effectiveness as a "real-world" code generation tool. (Note that since our operating system only runs on MIPS machines, these two experiments were done on the MIPS platform only.)

4.1 tcc: a ‘C compiler

VCODE's low-level interface makes it a good compiler target language: compilers can rely on it to emit code efficiently while retaining sufficient control to perform many optimizations. Furthermore, since VCODE is portable, a compiler that compiles to it has the benefit of its generated code working on the machines that VCODE supports.

We have implemented a ‘C compiler, tcc, that uses VCODE as an abstract machine to generate code dynamically [19]. As discussed in Section 2, ‘C is a superset of ANSI C that provides language constructs programmers can use to generate code at runtime. For example, programmers use ‘C to specify expressions and statements that will be generated at runtime; these code fragments can be dynamically composed and compiled at runtime. tcc is based on the lcc ANSI C compiler. We modified lcc to use two code generation backends. The first backend is used to emit assembly for traditional statically generated code. The second backend is used to compile ‘C constructs to VCODE. The emitted VCODE is then executed at runtime to generate the code specified by the application programmer. The use of VCODE as a target machine has allowed us to build a ‘C compiler that runs on a variety of machines with modest effort.

Our experience using VCODE as a target machine has been positive. Compiling to VCODE has been easier than compiling to more traditional RISC architectures. This ease is due both to the regularity of the VCODE instruction set and to the fact that VCODE handles calling conventions. The use of VCODE has allowed us to isolate most machine dependencies from the tcc compiler itself. For instance, tcc uses the same VCODE generation backend on the two architectures it supports (MIPS and SPARC).

4.2 DPF

There have been many interpreters that dynamically compile frequently used code at runtime [2, 4, 6, 13, 22, 23]. In a similar vein, we used VCODE as a dynamic compiler for a packet-filter message demultiplexer [8, 9].

Message demultiplexing is the process of determining which application an incoming message should be delivered to. Packet filters are a well-known technique used to implement extensible kernel demultiplexing [17]. A packet filter is a piece of user-level code downloaded into the kernel that is used to claim packets belonging to a given application. Packet filters are predicates written in a small safe language. This approach allows new protocols to be implemented outside of the kernel (and then downloaded into the packet filter driver), greatly increasing flexibility.

Filter	Classification Time
MPF	35.0
PATHFINDER	19.0
DPF	1.5

Table 3: Average time on a DEC5000/200 to classify TCP/IP headers destined for one of ten TCP/IP filters; times are in microseconds. DPF uses VCODE to dynamically compile packet-filters, while PATHFINDER and MPF are both interpreter-based.

Traditionally, packet filters are interpreted, which entails a high computational cost. As a result, most high-performance networking systems do not use them, despite the flexibility and extensibility they provide. To remedy this situation, we have implemented Dynamic Packet Filters (DPF), a new packet filter system that is over an order of magnitude more efficient than previous systems [8, 9]. The key to our approach is dynamic code generation. DPF exploits dynamic code generation in two ways: (1) by using it to eliminate interpretation overhead by compiling packet filters to executable code when they are installed into the kernel and (2) by using filter constants to aggressively optimize this executable code. As a result, DPF is equivalent in performance to hand-crafted message classifying routines (when it can exploit runtime information, it is even faster) while still retaining the flexibility of the packet filter model.

An example of how DPF exploits runtime information is how it optimizes the common situation where concurrently active filters examine the same part of a message and compare against different values. For instance, all TCP/IP packet filters will look in messages at identical fixed offsets for port numbers. In static systems these values are not known at compile time, and so a general-purpose, possibly expensive hash function must be used, along with checks for collisions, etc. However, since DPF knows both the number and the actual values that must be compared, it can optimize the comparison in a manner similar to how optimizing compilers treat C `switch` statements: a small range of values is searched directly, sparse values are matched using binary search, and dense ranges are matched using an indirect jump. Additionally, since the number and value of keys are known at runtime, DPF can select among several hash functions to obtain the best distribution, and then encode the chosen function directly in the instruction stream. Furthermore, since DPF knows at code-generation time whether keys have collided, it can eliminate collision checks if no collisions have occurred.

We measure DPF's time to classify packets destined for one of ten TCP/IP filters, and compare its time to measurements for PATHFINDER [1], the fastest packet filter engine in the literature, and MPF [24], a widely used packet filter engine. To ensure that the comparison is meaningful, we perform the same experiment described in [1]: the average of 100,000 trials is taken as the base cost of message classification. Table 3 presents the time to perform this message classification. This experiment is more fully described in [8]. In this experiment, DPF is 20 times faster than MPF and 10 times faster than PATHFINDER. The bulk of this performance improvement is due to the use of dynamic code generation,

4.3 ASHs

ASHs are user message handlers that are safely downloaded into the operating system kernel in order to direct message processing [8]. VCODE has been used as part of the ASH system to provide support for dynamic and efficient modular composition of network protocols.

Modular composition of different network protocols has long been a goal in the networking community [3]. Unfortunately, modular composition is expensive. The problem it presents is that each protocol layer frequently has data-touching operations associated with it (e.g., to perform checksumming or byte swapping). Each operation typically touches all (or most) bytes in a message. Separating these operations into modular pieces has meant that data is manipulated multiple times. As a result, modularity has exacted a high performance penalty [3], both because many excess scalar operations are performed (e.g., looping overhead) and because touching memory multiple times stresses the weak link in modern workstations, the memory subsystem.

To solve this problem we have constructed a network subsystem that uses VCODE to integrate protocol data operations into a single optimized pass over memory (e.g., integrating checksumming and byte swapping into a memory copy operation).

We gain efficiency from VCODE in two ways. First, it allows access to machine instructions that have no natural high-level idiom. By writing each data processing step in terms of VCODE it is possible for clients to write code that is more efficient than if it were written in a high-level language. Second, it is used to compose multiple data processing steps dynamically into a single specialized data copying loop generated at runtime. This system is described and measured in [9].

Table 4 shows the performance benefit of integrating checksumming and byte swapping routines into the copying loop from a network buffer to an application's memory. Measurements are taken both when the data is in the cache and when it has been flushed. Even without an intervening cache flush, the integration provides a performance benefit of 20–50%, and is clearly worthwhile. In the case where there is a flush, the integration almost always provides a factor of two performance improvement. The table also shows the relative efficiency of our emitted copying routines (labeled "ASH") by comparing them to hand-integrated, non-modular loops written in C. The main reason for ASH's better performance relative to the C routines is that the ASH system dynamically generates a memory copying loop specialized to the operations performed by each layer.

The use of VCODE allows flexibility not previously possible (i.e., the *dynamic* composition of data manipulation routines) while simultaneously making ASHs as efficient as previous systems that provided neither modular nor dynamic composition (i.e., systems where data manipulation steps were merged by hand). Due to the use of dynamic code generation, the ASH system is a rare case where flexibility has been provided "for free."

5 VCODE Generator Design

VCODE's code generator emits machine code "in place." This section provides details of this approach and discusses a few of its consequences.

5.1 The life of one instruction

VCODE can generate machine code in place because (1) clients associate each VCODE instruction with virtual registers at static compilation time and (2) VCODE's instructions closely match those of modern architectures. To make (2) concrete, we explain what happens to a single VCODE instruction, v_addu. Its macro definition and expansion is given in Figure 2, along with the machine code required to dynamically generate it. In this case, the total cost is nine MIPS instructions. With a contiguous run of VCODE instructions, some of the operations used to generate code can be reused by the compiler compiling the VCODE macros, bringing the total cost even lower.

5.2 Challenges of in-place code generation

Generating code in place is challenging, since it requires that VCODE emit code without global knowledge about the function it is creating. For instance, VCODE does not know if the function it is generating is a leaf procedure, the number of local variables it allocates, the amount of memory (and the number of instructions!) it needs to save floating point and temporary registers, etc. Traditionally, such knowledge would be derived by making at least one pass over intermediate data structures. Unfortunately, similar methods would have added unacceptable overhead to VCODE. We briefly look at four of the main challenges to in-place code generation below.

Instructions that access a procedure's stack typically require information about the size of its activation record. For instance, on many machines, instructions that save and restore registers must know the activation record size in order to compute offsets into the "register save area." Similarly, accesses to local variables must know the size in order to compute a given local's stack offset. Without the activation record size, these instructions cannot be emitted. Unfortunately, this size is not known until all code is constructed. Therefore, in order to generate code in place, VCODE must finesse the need to accurately know the activation record size. An awkward solution to this problem would be to force clients to allocate all locals and registers before any code generation occurs. In practice, this restriction would limit client flexibility. Furthermore, in many cases, this restriction would hurt performance since it would require that clients generate VCODE in two passes: the first pass to compute the number of local variables and registers they need, the second pass to actually dynamically generate code using VCODE. VCODE's solution is inelegant but workable: it simply allocates the space needed to save all machine registers and then locates space for locals above this fixed-sized area (or below, depending on how the stack grows). With this solution, register offsets are known at code emission time, and offsets for local variables can be computed as the variables are allocated. VCODE also marks where in the generated instruction stream the activation record is allocated and then backpatches this instruction when the final activation record size is known (i.e., after all locals have been allocated). VCODE's solution is a tradeoff of space for time. On modern RISC machines it wastes, in the worst case, the stack space required to save 32 integer and floating point registers (about 64 words). For clients that find this tradeoff unacceptable it would not be hard to turn it off on a per-function basis. We have not yet found the need to do so.

The second challenge deals with handling the saving and restoring of callee-saved registers in a function's prologue. The problem VCODE must solve is that when a client begins generating code for a function, VCODE does not know the number

Machine	Method	copy + checksum	copy + checksum + byte swap
DEC3100	separate / uncached	1630	3190
	separate	1290	2230
	C integrated	1120	1750
	ASH	1060	1600
DEC5000	separate / uncached	812	1640
	separate	656	1280
	C integrated	597	976
	ASH	455	836

Table 4: Cost of integrated and non-integrated memory operations. Times are in microseconds.

```
/* vcode instruction to add two unsigned integer registers on the MIPS architecture. */
#define v_addu(rd,rs1,rs2) addu(rd, rs1, rs2)

/* Macro to generate the MIPS addu instruction (opcode is 0x21). */
#define addu(dst, src1, src2) (*v_ip++ = (((src1) << 21) | ((src2) << 16) | ((dst) << 11) | 0x21))

# MIPS assembly code generated by gcc –O2 to implement the "addu" macro.
        lw      v1,1244(gp)     # allocate instruction
        sll     a1,a1,21        # shift and then or in the register values
        sll     a2,a2,16
        or      a1,a1,a2
        sll     a0,a0,11
        or      a1,a1,a0
        addiu   v0,v1,4         # bump instruction pointer
        sw      v0,1244(gp)     # store the new instruction pointer
        ori     a1,a1,0x21      # or in the opcode
        sw      a1,0(v1)        # store instruction in memory
```

Figure 2: Top-down expansion of the VCODE v_addu instruction.

of callee-registers that the client will use and so cannot generate the function's prologue code. As with activation record size, VCODE only knows how many callee-saved registers a client used after all client code generation for the function is complete. VCODE solves the problem by reserving space in the instruction stream to save the maximum number of callee-saved registers an architecture supports. This reservation is done when code generation for a function is initiated (i.e., during the call to v_lambda). Register saves are then inserted into this prologue area after the client is finished generating code. In practice, the amount of space wasted is small (e.g., 32–64 words of memory per generated function). The dual of this problem is that during code generation the epilogue code VCODE needs to generate is also not known. VCODE uses the traditional compiler method of jumping to a piece of per-function epilogue code when a client indicates control should return from the function. While back-patching jumps, VCODE does some simple optimizations to eliminate this jump if the procedure did not use any callee-saved registers.

On many architectures, leaf procedures can be profitably optimized. For instance, activation record allocation can be elided and caller-saved registers can be used to hold persistent variables. Unfortunately, VCODE cannot determine if a procedure is a leaf procedure until all code is generated. More importantly there is, in general, no easy way to assume a procedure is a leaf for optimization purposes and then "rollback" any code transformations: the effects of leaf optimizations can be pervasive and, as a result, make code backpatching expensive or difficult to implement. To address this problem VCODE allows programmers to indicate if a function is a leaf procedure. If the client attempts to call a procedure from the function, VCODE signals an error. However, this is not a complete solution: VCODE instructions may cause function calls. For instance, on machines that do not provide division in hardware, the VCODE integer division instructions require subroutine calls. To preserve portability, these calls should not cause the user-program to get a runtime error from VCODE. In the worst case VCODE ignores client hints when running on those machines. Fortunately, routines that emulate common machine instructions frequently obey different calling conventions than normal subroutines in that they save all caller-saved registers. In this case VCODE is able to call the emulated routine after saving the registers used to pass its arguments.

A final problem not necessarily unique to in-place code generation is the handling of floating point immediates. Unlike integer immediates, many architectures do not provide support for encoding floating point immediates in arithmetic instructions. As a result, a dynamic code generation system must allocate space for them and load them from this space. To make garbage collection of floating-point constants easy, VCODE places them at the end of a function's instruction stream. In this way, the space for the immediates is easily reclaimed when the function is deallocated.

5.3 Violating VCODE abstractions

An often-ignored aspect of fast systems is the question of how to allow high-level interfaces to be gracefully violated for improved performance or control. Since we use VCODE in situations where every cycle is precious and where the environment has peculiar constraints (e.g., operating system interrupt handlers), we have designed mechanisms into VCODE to allow clients to violate VCODE abstractions in order to achieve both more efficient code generation and more efficient generated code. We briefly outline three of these mechanisms below.

Clients can dynamically control the register class VCODE assigns to each physical register (callee-saved, caller-saved, or unavailable). This allows clients to use VCODE's generated code in situations where normal register conventions do not hold. For instance, in an interrupt handler all registers are "live." Therefore, for correctness, VCODE must treat all registers as "callee-saved."

Clients that wish to trade register allocation flexibility can obtain faster code generation speeds by using hard-coded register names. This reduces the cost of code generation by approximately a factor of two. To support this optimization, VCODE provides architecture-independent names for temporary ("T0", "T1", etc.) and callee-saved registers (e.g., "S0", "S1", etc.). Clients (in this case, typically compilers) use these names in their VCODE instructions. Since these names are constants, the C compiler can perform constant folding and reduce the overhead of dynamic code generation to the loading of a 32-bit immediate into a register and the storing of this register into memory (approximately five instructions). We note that while this mechanism was originally intended to make code generation faster, it also supports a form of "register assertion": clients that use it will get compilation errors if the machine they are being compiled for does not support the number of temporary and variable registers that they require. Such errors could be used to select different code to compile (e.g., code that either uses virtual registers or fewer physical registers).

Finally, clients that are willing to deal with a lower-level and more dangerous interface are able to perform instruction scheduling of loads and branch delay slots without impacting code generation speed. The basic problem with portably exposing delay slots within the VCODE instruction set is that reconciling them with the underlying machine can be expensive. In general, it requires some form of per-instruction check. For instance, if VCODE jump delay slots were one instruction and the underlying machine did not have delay slots, then every instruction would have to check if it was in a "logical" delay slot and, if so, swap places with the already generated jump instruction. Performing this switch would not be trivial, since a jump instruction could have expanded to be several instructions. Fortunately, the solution is simple. VCODE includes two macros: v_schedule_delay and v_raw_load that are used by clients to schedule instructions across branches and loads respectively. The first macro takes a VCODE branch instruction definition (e.g., v_bneii(rs, 10, label)) and an instruction to put in its delay slot. If the machine has delay slots and the instruction fits in the delay slot, VCODE inserts the instruction in the delay slot. Otherwise this instruction is simply placed before the branch in the instruction stream. The v_raw_load macro is even simpler: it takes a memory instruction as an operand along with a count of the number of VCODE instructions that will be emitted before the result is to be used. If this number is less than the number of cycles required to safely make the result of the load available, VCODE will insert the required number of "nops." With these macros, clients are able to portably and efficiently schedule load and branch delay slots without affecting code generation speed.

5.4 Extensibility

The VCODE system is simple and modular. These characteristics make it easy to dynamically reconfigure and extend. For instance, we have built a sophisticated strength reducer for multiplication and division by integer constants on top of VCODE; support for unlimited virtual registers could be added in a similar manner. Clients can dynamically substitute calling conventions on a per-generated-function basis.

An example of VCODE's extensibility is the ease with which its instruction set can be extended. As discussed in Section 3, the elimination of intermediate structures contributes the most to this feature as it removes the need for VCODE to understand instruction semantics. With intermediate data structures, extension is more complex, since any extension must be couched within the context of a well-formed data structure.

VCODE provides a preprocessor that consumes a concise instruction specification and automatically generates the specified set of VCODE instruction definitions. A simplified form of this specification is:

```
'(' base_insn_name
    '(' param_list ')'
    [ '(' type_list mach_insn [ mach_imm_insn ] ')' ]+
')'
```

Each base_insn_name is composed with each type_list and mapped to the associated register-only machine instruction (mach_insn) and, if it is given, the associated immediate instruction (mach_imm_insn). Machine instructions are usually provided by VCODE itself. However, on machines with large instruction sets, VCODE may neglect to specify all instructions and the client must then provide any missing instructions that it needs.

For instance, the following client specification adds a square-root instruction on the MIPS:

```
(sqrt (rd, rs) (f fsqrts) (d fsqrtd))
```

This specification composes the base instruction type sqrt with the two types f (float) and d (double) and associates them with the target MIPS instructions fsqrts and fsqrtd, respectively. It generates the following VCODE instruction definitions:

```
#define v_sqrtf(rd,rs)  fsqrts(rd,rs)
#define v_sqrtd(rd,rs) fsqrtd(rd,rs)
```

To simplify instruction extension, our specification language includes facilities for constructing more complicated sequences of instructions, acquiring access to scratch registers, and couching new instructions in terms of existing VCODE instructions.

Thus, a single line in a preprocessing specification can add a new family of instructions. By additionally couching an extension in terms of the VCODE core (or other extensions), a client can ensure that its extensions will be present on all machines.

6 Discussion

A reasonable question to ask is how fast a dynamic code generation system must be before it is "fast enough." The main determinant of this question what actions a client must take during code generation: dynamically generated code is generally not created *ex nihilo* but rather is based on client data structures. Traversing these data structures to determine what code to emit can consume tens or even thousands of instructions (e.g., in the case of compilation from a high-level source). As a result, VCODE's performance is likely to be more than sufficient for most applications. Furthermore, as we show in Section 5 clients can use hard-coded register names and reduce this overhead by about a factor of two.

This section reports on our experience using VCODE and some limitations of the current system.

6.1 Experience

VCODE has been in daily use in real systems for over a year. It has proved to be a useful tool and has performed well in demanding situations. For instance, the DPF system we described earlier in the paper is used as the packet filter system for the Aegis exokernel operating system [8].

A nice practical feature of VCODE is that its complexity is mostly horizontal rather than vertical: each additional piece of the VCODE system usually does not depend on others. As a result, each extension increases the number of system states roughly additively rather than multiplicatively. This feature makes testing more simple than the DCG system that VCODE descends from. The most common error we have found is the mis-mapping of VCODE instructions to machine instructions. Compared to the complexities of finding code generation bugs in full-fledged compilers, this is a fairly benign error, easily caught with automatically generated regression tests.

One disappointment in the implementation of VCODE is the inadequacies it exposed in current compiler and linker implementations. There were three main problems the VCODE implementation continually ran into: linkers which brought in all routines from a file rather than those needed, compilation of inlined routines that were not referenced, and inefficient handling of word-sized structures. A VCODE machine specification generates many procedures (one for each VCODE instruction). Most of these procedures will not be used by a given application. Unfortunately, if these procedures are stored in a single file, all Unix linkers we used linked in the entire file rather than simply pulling in the VCODE procedures that were needed. This space overhead was unacceptable, especially when the solution was not difficult to implement. The second problem was less easy to solve, but just as detrimental. For high performance, it is appropriate to inline most VCODE instruction invocations. Unfortunately, those few compilers that provide an inline directive will parse all inline functions, even if they are "static" and not referenced by any procedure in scope. Since VCODE creates a procedure per VCODE instruction this problem leads to long compilation times. VCODE's solution to this problem and the previous one was to use macros to implement VCODE instructions: macros do not take up space when they are not referenced, and they are fast to parse. Unfortunately, this is far from a perfect solution, since it can cause a space explosion if many VCODE instructions are used. The final problem was how badly many compilers handled word-sized structures: rather than allocating the structure to a register and operating on it directly, most compilers we used would load and store the structure to memory on every operation. This was true even when the structure contained a single unsigned integer. Since VCODE registers are represented as C structures (to allow stronger type checking than would be possible with a unwrapped integer), this overhead could add a noticeable cost to every VCODE instruction. We solved this problem by allowing VCODE to be optionally compiled to represent registers using integers.

6.2 VCODE limitations

While VCODE has been useful in practice, it has four main drawbacks: the lack of a symbolic debugger, limited registers, no peephole optimizer and no instruction scheduler. Of the four, the first is the most critical: debugging dynamically generated code currently requires stepping through it at the level of host-specific machine code. If one does not have a working knowledge of this instruction set, making sense of the debugging output is challenging. Fortunately, fixing this problem is not difficult. With modifications, the preprocessor could automatically generate a debugger from the machine specification files. Each instruction would be "interpreted" by generating it using VCODE and recording its effects. A nice consequence of this arrangement is that client-added instructions can be incorporated into the debugger automatically.

We are currently adding support for unlimited virtual registers as a VCODE extension layer. Preliminary results indicate that the addition of this (optional) support would increase code generation cost by roughly a factor of two. However, we note that in the systems we have built so far the current VCODE arrangement of client-managed registers has not presented problems.

On machines where VCODE instructions map more or less directly to native instructions (e.g., the MIPS architecture), the lack of a peephole optimizer has not noticeably hurt the quality of VCODE's generated code. However, on machines where a single VCODE instruction can map to multiple machine instructions, a peephole optimizer would be useful. For instance, the current generation of Alpha chips lack byte and short word operations. As a result, VCODE must synthesize its load and store byte instructions from multiple Alpha instructions. This emulation can require a large number of instructions: in the worst case an "unsigned store byte" instruction requires eleven Alpha instructions and five temporary registers! These emulated instructions typically compute useful intermediate results that a peephole optimizer could reuse. Unfortunately, the current implementation of VCODE cannot similarly exploit these regularities since it has an extremely local view of code generation (i.e., a single VCODE instruction). Future work will include implementing a VCODE-level peephole optimizer for clients that wish to trade runtime compilation overhead for better generated code.

Of all drawbacks, the lack of an instruction scheduler is the least critical. Most instruction scheduling can actually be handled above VCODE by separating register uses from definitions. Clients that wish to micro-schedule delay slots can do so using VCODE's portable instruction scheduling interface. Furthermore, current architectural trends are to deep instruction reorder buffers (e.g., 64-entries deep). At this level, static instruction scheduling is not particularly important. [2]

7 Conclusion

We have presented VCODE, a fast, retargetable and extensible dynamic code generation system. It generates machine code at an approximate cost of ten instructions per generated instruction, which is roughly 35 times faster than the fastest equivalent system in the literature [10]. It can be retargeted to RISC machines in approximately one to four days. Finally, it allows many levels of the system to be parameterized at runtime.

VCODE provides a portable, idealized RISC instruction set to clients. Clients use this interface to specify dynamically

[2] This insight is due to Todd Proebsting.

generated code from within applications. VCODE generates machine code in place, without the intermediate passes typical of other code generation systems. As a result, VCODE is substantially faster than previous systems and consumes little memory during code generation. An important side-effect of eliminating intermediate data structures is that the VCODE instruction set can be readily extended by clients since VCODE does not enforce any particular semantics on instructions.

VCODE is not a toy system. We have used it extensively in the networking subsystem of our experimental exokernel operating system [8] and as a compiler backend for an extension of ANSI C that supports dynamic code generation.

VCODE both generates code efficiently and generates efficient code. Its interface and abilities make it useful for a broad class of clients. VCODE is the first general-purpose dynamic code generation system to generate code without the use of an intermediate representation, the first to support extensibility, and the first system to be made publicly available (source can be obtained from the author). We hope that the availability of VCODE will help raise dynamic code generation from a curiosity to its rightful position as a common optimization tool.

8 Acknowledgments

I thank Massimiliano Poletto for his patience while porting ‘C to VCODE and Marcus Daniels for his role as beta-tester. I thank Eddie Kohler for extensive editing and typesetting assistance; Greg Ganger and Deborah Wallach for thorough readings of the paper. I especially thank Professor Frans Kaashoek for his support of this research and Todd Proebsting for many insightful discussions about dynamic code generation.

References

[1] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123, November 1994.

[2] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI ’89*, pages 146–160, Portland, OR, June 1989.

[3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, September 1990.

[4] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[5] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, St. Petersburg, FL, January 1996.

[6] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of 11th POPL*, pages 297–302, Salt Lake City, UT, January 1984.

[7] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 22th Annual Symposium on Principles of Programming Languages*, 1995.

[8] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[9] D. R. Engler, D. Wallach, and M. F. Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT, March 1995.

[10] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of ASPLOS-VI*, pages 263–272, October 1994.

[11] C. W. Fraser. A language for writing code generators. In *Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation*, pages 238–245, 1989.

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

[13] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of PLDI ’94*, pages 326–335, Orlando, Florida, June 1994.

[14] D. Keppel, S.J. Eggers, and R.R. Henry. A case for runtime code generation. TR 91-11-04, Univ. of Washington, 1991.

[15] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Copenhagen, Denmark, June 1994.

[16] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN ’96 Conference on Programming Language Design and Implementation*, May 1996.

[17] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[18] R. Pike, B.N. Locanthi, and J.F. Reiser. Hardware/software tradeoffs for bitmap graphics on the Blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A template-based compiler for ‘C. In *Workshop on Compiler Support for Systems Software*, Tucson, AZ, February 1996.

[20] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[21] N. Ramsey and M. F. Fernandez. The New Jersey Machine-Code Toolkit. Technical Report 95-082, Purdue University, Dept of Computer Sciences, December 1995. Submitted to *ACM Transactions on Programming Languages and Systems* .

[22] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[23] J.E. Veenstra and R.J. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Modeling and Simulation of Computers and Telecommunications Systems*, 1994.

[24] M. Yahara, B. Bershad, C. Maeda, and E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.