Cint: A RISC Interpreter for the C Programming Language†

Jack W. Davidson
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Joseph V. Gresh
AT&T Communications and Information Systems
Lincroft, NJ 07738

ABSTRACT

Cint is an interpretation system for the C programming language. Like most interpretation systems, it provides "load and go" type execution as well as enhanced debugging and performance analysis tools. *Cint* consists of two phases—a translator and an interpreter. The translator compiles the source program into code for a virtual machine. The interpreter then loads and executes this code. While providing services similar to traditional interpreters, *Cint* differs from them in two important ways. First, the virtual machine languages used by many interpreters are quite large; machines with 100 to 200 operations are common. In contrast, *Cint*'s virtual machine has only 63 operations. Second, to achieve acceptable execution speeds, interpreters are often implemented in the assembly language of the host machine. *Cint*, however, is written entirely in C and is therefore portable. In fact, it has been transported to four machines without modification. Despite the compact size of the virtual machine language and the high-level language implementation, *Cint*'s execution speed is comparable to that of other interpreters. This paper describes the design of the virtual machine, the implementation of the interpreter, and the performance of the system.

†This work was supported in part by the National Science Foundation under Grant CCR-8611653.

1. INTRODUCTION

Cint is an interpretation system for the C programming language. Like many other high-level language interpretation systems, it is implemented via the technique of abstract machine modelling [13]. In this technique, the fundamental operators and data types required by the high-level language are used to define an instruction set for a virtual machine. The interpretation system is realized by constructing a *translator* and an *interpreter*. The translator compiles the source language programs into code for the virtual machine. The interpreter then loads and executes the code produced by the translator. This technique has been used to implement a number of successful systems [2, 5-7, 17].

Cint, however, differs from traditional high-level language interpreters in two important respects. First, the specification of *Cint*'s virtual machine was driven by the principles used to design reduced-instruction-set computers (RISC) [15]. As a result, *Cint*'s virtual machine has only 63 operations. In contrast, the virtual machines used by some interpreters are quite large with two to three times as many operations [2, 10, 16]. Second, *Cint* is implemented entirely in a high-level programming language. In order to achieve acceptable execution speeds, interpreters are often implemented in the assembly language of the host machine. Consequently, they may require considerable effort to be moved to a new machine. *Cint*, on the other hand, is easily portable and has been moved to four machines without modification.

Despite the use of a RISC-like virtual machine and its realization using a high-level language, *Cint* achieves execution speeds comparable to traditionally implemented interpreters. In this paper we provide an overview of *Cint* that focuses on the design of the C virtual machine, the implementation of the program that realizes the virtual machine, and its performance.

2. MACHINE DESIGN

A computer system can be seen as being made up of several layers or levels, where each level defines a different machine view. The lowest of these levels is defined by the individual devices that form the machine (e.g., transistors and resistors), while the higher levels are defined by the operating system and the programming languages available on the machine. It is natural to view these levels as a hierarchy of machines. Machines high in the hierarchy are often called *virtual* machines to distinguish them from the conventional architectural level. A machine at level *N* is implemented by a program that runs on the machine at level *N-1*. For example, the conventional machine level (i.e., assembly language level) is implemented by a microprogram that runs on the machine defined by the micro-architecture. Similarly, the virtual machine used to implement a high-level language interpreter is often implemented by a program that runs at the conventional machine level.

One of the primary goals of a machine designer is the construction of machines that support the efficient execution of programs that will run on them. A number of new principles have evolved for guiding the design of conventional level machines. The distinguishing characteristic of these machines is the reduced number of operations contained in the instruction set. Consequently, these machines have been termed RISCs—reduced-instruction-set computers [14].

Patterson [15] lists some of the RISC design principles:

1. Functions should be kept simple unless there is a very good reason to do otherwise.
2. Microinstructions should not be faster than simple instructions.
3. Moving software into microcode does not make it better.

These principles lead to a very simple definition of a RISC machine: *a RISC machine completes the execution of an instruction every cycle*. Indeed, the characteristics shared by existing RISC machines (e.g., register-to-register architecture, a reduced number of operations and addressing modes, simple instruction formats, and a pipelined execution unit), are simply techniques for realizing the above definition. Preliminary results from both experimental and commercial machines show that the RISC concepts do lead to machines that provide for the fast execution of high-level language programs.

It seems logical that if a set of design principles works well when applied to the lower end of the machine hierarchy, these same principles may also produce good results when used to design virtual machines. Based on this premise, the design of the C virtual machine (CVM) for *Cint* was guided, to a large extent, by the principles and arguments used to design RISCs. The following section describes the CVM and some of the motivation for its design.

2.1 The C Virtual Machine

There are a number of arguments for designing a small, instruction set. One argument is that a small, simple instruction set is easier, less error prone, and faster to implement than a large, complex instruction set. Abstract machine designers have long recognized this problem. In 1972, Newey, Poole, and Waite [13] observed that

> "problems ... suggest a number of specialized operations which could possibly be implemented quite efficiently on certain hardware. The designer must balance the convenience and utility of these operations against the increased difficulty of implementing an abstract machine with a rich and varied instruction set."

A second argument in favor of RISCs is that their compilers are simpler than compilers for complex-instruction set computers (CISCs). While it is debatable whether this argument is applicable to real machines, our experience is that it does apply to abstract machines used to produce retargetable compilers [3]. If the virtual machine does not contain special operations, the case-analysis code typically necessary to determine whether special operators can be emitted is not required. Section 3 describes in more detail the effect the use of a RISC-like virtual machine had on *Cint*'s translator.

Based on these observations, one of our primary design goals was to keep the CVM as small as possible, yet still obtain satisfactory execution performance from the interpreter. Consequently, the CVM has 49 executable instructions and 14 pseudo-operations. The full instruction set is described in Appendix A.

The CVM's design was also influenced by the decision to use a high-level language as the implementation machine rather than the conventional machine level. While our previous definition of a RISC machine (i.e., single cycle execution of instructions) does not directly apply to virtual machines, its intent can be applied. Consequently, each CVM

instruction performs a relatively simple operation that has a rather obvious and simple realization on the implementation machine.

Unlike most real RISC machines, the CVM is a stack architecture as opposed to a register architecture. The CVM is stack-oriented for several reasons. Most high-level programming languages do not give the programmer explicit control of a machine's registers†. Consequently, our choice of a high-level programming language as an implementation machine precluded making the CVM a register architecture. A second reason for favoring a stack-oriented virtual machine was the goal of using the translator as the front-end in a retargetable C compiler. While it is relatively straightforward to map a stack-oriented virtual machine onto a register machine (simply treat the registers like a stack), it is more difficult to map a register-oriented virtual machine onto a stack machine.

The CVM provides instructions for manipulating two distinct stacks. Most instructions manipulate values found on the evaluation stack or E-stack. A second stack, called the C-stack, is used to support the C calling sequence. Only five instructions (`PUSHA`, `PUSHV`, `STARG`, `CALL`, and `RET`) access this stack. Again, the decision to differentiate between the evaluation stack and the call/return stack was motivated by the desire to also use the CVM as an abstract machine for producing retargetable C compilers. Typically, a code generator maps the E-stack onto the target machine's allocable registers, while the C-stack is mapped onto the target machine's runtime stack. The difference between the E-stack and C-stack is a logical distinction that can be ignored. In fact, *Cint*'s interpreter maps the C-stack onto the E-stack.

The basic format of a CVM executable instruction is

```
opcode      type   [operands]
```

where `type` indicates the data type (i.e., short, long, float, etc.). Many virtual machines encode the type as part of the operation to remove a level of decoding in the interpreter. This approach is feasible for a language that supports a few basic types (e.g., Pascal with four basic types). However, for a language like C with a large set of basic types, such an approach is unwieldy.

†The register declaration in C is merely a hint to the compiler that the variable will be heavily used.

The CVM instruction set, like those of real RISC machines, could be reduced further. For example, the assignment operators (e.g., +=, -=, etc.) could be synthesized from other CVM instructions. Experimentation showed that further reductions of the CVM would be counter-productive because it resulted in the output of more verbose code from the translator that ran substantially slower.

3. CINT

Cint consists of a translator which compiles C source programs into code for the CVM, and an interpreter that loads and links CVM object modules and executes them. A schematic of *Cint* is shown in Figure 1. The following sections describe the translator and the operation of the interpreter.

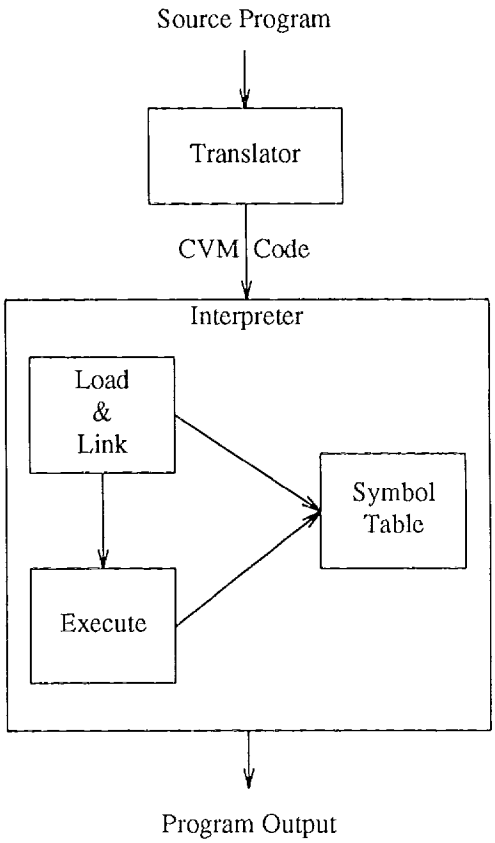

Figure 1. Schematic of *Cint*.

3.1 Translation

Cint's translator [19] supports the full C programming language as defined by Kernighan and Ritchie [11] including recently added features such as bit fields and enumeration types. The translator is written in C and is 7032 lines of code. It is retargeted by supplying two types of information about the target machine. The sizes of the basic data types on the host machine and the evaluation order of arguments must be specified. The latter is necessary for two reasons. One is that this allows interpreted code to call procedures that have been compiled and linked with the interpreter. A second reason is that some C programs (usually nonportable ones) are written assuming a particular evaluation order. One of the design goals of *Cint* was that the execution behavior of a program on the host machine should be indistinguishable whether it is interpreted or compiled and executed.

As noted in Section 2, one of the benefits of a RISC-like virtual machine is that it simplifies the implementation of the translator. Because the CVM provides exactly one obvious implementation for each source language construct, the task of code generation is made trivial. For example, typical virtual machines include a number of instructions for loading values directly from memory as well as a general indirect load instruction for when addresses must be computed (e.g., an array reference). Through case analysis, the code generator determines the best code sequence to emit. Such case analysis is often one of the more tedious and error prone parts of a compiler.

The CVM, on the other hand, has only one general-purpose operator for loading a value from memory. The dereference instruction (@) takes an address on the E-stack and replaces it with the contents of that memory location. Because this is the only way to load a value from memory, the code generator does not require any case analysis to determine the best code sequence to emit. Eliminating the necessity for case-analysis code markedly simplifies the implementation of the translator's code generator.

The code generator is about 1000 lines of C code. This includes the routines that emit the CVM code in a variety of styles and formats. Code is generated by performing a simple postorder walk of the trees produced by the semantic analyzer.

3.2 Interpretation

One of the secondary goals of this project was to investigate whether interpreters could be built that would be both portable yet provide satisfactory execution speeds. Most interpreters are implemented via the conventional machine level of the host architecture (i.e., assembly language). They sacrifice portability for execution speed. To make *Cint*'s interpreter portable, we chose an implementation machine higher in the machine hierarchy—the virtual machine defined by the C programming language and realized by a C compiler and runtime system. Thus, *Cint* is portable to any environment that supports the C programming language. As C has a rich set of operators that provide access to most of the operators provided by the underlying hardware, the "semantic gap" between C and the hardware is small. Our hypothesis was that by carefully designing the virtual machine so that it could be implemented efficiently using C, the resulting interpreter would run as fast as interpreters implemented using the assembly language of the host machine. Section 5 discusses the execution performance of the interpreter.

The interpreter is 4811 lines of C code. About half of the code, 2046 lines, implements the operations of the CVM. The interpreter is divided into two phases: loading/linking and execution. The following sections describes some interesting aspects of the implementation of these phases.

3.2.1 Loading and Linking

During the first phase of the loading process, *Cint* reads the CVM modules produced by the translator and places instructions and data in one of three memory segments: a program segment, a standard data segment, and a string segment. The program segment is loaded with CVM instructions and their operands. The string data segment holds string constants, while the standard data segment holds all other data.

The data structure used for the program segment is treated as an array of type `int`. CVM opcodes are naturally expressed as integers. All operands, whether constants or variable references, are converted to integer offsets from the base of the appropriate segment. The data structure for the standard data segment is treated as an array of type `char`. This allows data of all types to be stored in a single data structure. This data is accessed by casting a character pointer into the array to a pointer of the appropriate type. Furthermore, storing all data in a character array permits the correct calculation of the relative distance between elements of an array. This is necessary for address arithmetic of interpreted programs to be performed properly.

atof	atoi	atol	calloc	exit	_filbuf
_flsbuf	fclose	fflush	fgetc	fgets	fopen
fprintf	fputc	fputs	fread	free	freopen
fscanf	fseek	ftell	fwrite	getenv	malloc
printf	puts	realloc	rewind	scanf	setbuf
sprintf	sscanf	strcat	strcmp	strcpy	strlen
strncat	strncmp	strncpy	system	times	ungetc

Figure 2. Library routines supported by *Cint*.

One of the powerful concepts supported by C is separate compilation. *Cint* also supports separate compilation. After all the object modules have been read, the linker resolves external references among modules. All unsatisfied references are resolved from a set of standard I/O and utility library routines. Much of C's utility is due to a set of standard libraries that provide for input and output, operations on characters and strings, and storage allocation. *Cint* provides an interface to commonly used library routines. The routines currently supported are listed in Figure 2. *Cint* is designed so that additional external routines can be added easily. A number of these routines have different definitions depending on whether the supporting environment is BSD based or System V based. The interpreter can be configured to support either operating system view.

3.2.2 Execution

After loading, the CVM code is interpretatively executed. Klint [12] discusses and classifies three basic interpretation techniques. The three classifications are:

1. Classical interpretation with opcode table,
2. Direct threaded code, and
3. Indirect threaded code.

One of the major differences in these techniques is how the operation code (opcode) is encoded. In the classical technique, each operation is assigned a unique code. Some method of table lookup (usually indexing based on the opcode) is required to locate the routine that implements the operation. In the direct threaded code technique [1], the opcode is the address of the routine that implements the operation. To obtain more compact code and allow more flexibility in handling types, the indirect threaded code technique [4] adds a second level of indirection. The opcode is the address which points to a word that contains the address of the routine that implements the operation.

Klint measured the instruction fetch performance of these interpretation techniques on two machines. From these measurements, several interesting observations were made. First, the importance of the instruction fetch overhead is related to the complexity of the virtual's machine's instruction set. As the time to execute a virtual instruction increases, the impact of the time required to do an instruction fetch decreases. For a RISC virtual machine, reducing the instruction fetch overhead is critical. Klint's measurements showed that the instruction fetch overhead could be reduced substantially by keeping the virtual machine's program counter in a machine register. On the PDP-11, the instruction fetch time was two to three times faster when the program counter was held in a register as opposed to a memory location. Fortunately, by declaring the variable that holds the virtual machine's program counter to be a register variable, we were able to place the program counter in a register for all machines to which Cint was ported.

Klint's measurements also showed that the direct threaded code method resulted in the least instruction fetch overhead. Unfortunately, efficient implementation of the threaded code techniques requires access to the conventional machine level (i.e., assembly language implementation). For example, threaded code techniques can be implemented using C via pointers to functions, but the use of the C calling sequence and the resulting call/return overhead negates any advantage. Consequently, *Cint* uses the classical technique with instruction decoding being performed via the C switch statement. Because most architectures provide hardware support for switch or case statements, acceptable performance is achieved.

The heart of the interpreter is the runtime stack. The CVM supports two stacks: an evaluation stack and an activation record stack. As noted in Section 2.1, two stacks facilitated the development of portable compilers. Since interpretation only requires one

stack, *Cint* maps the C-stack onto the E-stack. All operations are performed on the top of the E-stack. The high-level implementation of the E-stack creates a number of difficulties. Interpreters implemented at the conventional machine level can directly access and manipulate values on the runtime stack. Furthermore, many machines provide for stack overflow checking and the automatic extension of the stack. To overcome the difficulty of manipulating values of different types on *Cint*'s runtime stack, the E-stack is implemented as an array of structures shown below:

```
typedef struct stktype {
   union {
      char c;
      int i;
      short s;
      long l;
      unsigned char uc;
      unsigned short us;
      unsigned int ui;
      unsigned long ul;
      float f;
      double d;
      char *p;
   } val;
   unsigned int type;
};
```

The field `type`, as the name suggests, contains the type of the item. This permits runtime checking of type compatibility. The field `val` contains the value of the stack item, and is therefore a union of the basic types supported by C. Since the basic stack element contains the union of all possible C basic types, the basic operations of the interpreter can be realized using the corresponding C operator applied to the correct field of the union. This ensures that interpreted programs exhibit the same behavior as compiled programs.

When pushing a pointer of any type onto the runtime stack, it is cast with the `(char *)` operator, and then cast back to the appropriate type when dereferenced. The assumption is that a pointer of type "pointer to character" should be capable of holding the equivalent of a pointer to any other type without loss of information. All C implementations that we know of satisfy this assumption.

Because the E-stack is an internal data structure, it is *Cint*'s responsibility to check for possible stack overflow. An early implementation of *Cint* performed a stack overflow check each time a value was pushed on the runtime stack. Execution profiles of the *Cint*'s operation showed that these checks were substantially increasing the cost of executing some instructions. To reduce this overhead, the responsibility for checking for stack overflow was moved to the function prologue code. During the loading process, the maximum amount of stack space a function could use is computed. This number is one of the operands of the CVM `FUNC` instruction. Part of the semantics of the this operation is to ensure that enough stack space is available to execute the function.

While realizing the E-stack as an array of structures solves the problem of pushing values of different types on the same stack, it creates a new problem. The translator generates code to perform pointer arithmetic and array indexing based on the assumption that array elements are stored contiguously in memory. For example, array indexing is performed by multiplying the array index by the size of an array element and adding the result to the base address of the array. Consequently, the E-stack cannot be used to hold local variables. The problem is overcome using the following scheme. On function entry, a block of contiguous memory is allocated and a pointer to this memory is held in a variable that serves as the local variable pointer. A pointer to a local variable may then be obtained (as in the `NAME` opcode) by simply adding that variable's offset to the value of the the local variable pointer. The local variable pointer is saved on a function call. When a function returns, the memory pointed to by its local variable pointer is freed and the caller's local variable pointer is restored.

A major advantage of a reduced instruction set virtual machine is that less effort is required to implement the program that realizes the virtual machine's operations. Reductions in the number of lines of code necessary to implement the interpreter stem from two characteristics of RISC machines. First, the reduced number of operations means fewer operations to implement. The CVM, for example, requires the implementation of only 49 executable operations. Second, the simplicity of the operations means that the code to implement them is shorter and easier to write. In *Cint*, with a few exceptions (e.g., `CALL`, `RET`, `FUNC`, etc.), each operator is naturally realized by one or two lines of simple C code. On the other hand, virtual machines with complex instructions require the implementation of a larger instruction set that has complex instructions that are more difficult to realize.

Program	PCC					Cint				
	Compile Time		Run Time		Total	Compile Time		Run Time		Total
	User	*Sys*	*User*	*Sys*		*User*	*Sys*	*User*	*Sys*	
banner	5.7	3.7	0.1	0.3	**9.8**	3.7	1.1	0.9	0.7	**6.4**
cal	5.4	2.6	0.3	0.4	**8.7**	1.8	1.0	2.4	0.9	**6.1**
cb	12.7	3.1	0.2	0.3	**16.3**	4.9	1.6	2.1	1.0	**9.6**
cmp	4.6	2.6	0.1	0.3	**7.6**	1.6	0.9	0.6	0.6	**3.7**
echo	1.9	2.3	0.1	0.2	**4.5**	0.6	1.0	0.2	0.5	**2.3**
grep	12.3	3.2	0.4	0.4	**16.3**	4.7	1.1	23.6	3.7	**33.1**
tr	5.1	2.6	0.6	0.4	**8.7**	2.0	0.8	25.2	3.2	**31.2**
wc	3.2	2.3	0.4	0.4	**6.3**	1.2	1.0	16.8	2.2	**21.2**

Table I. Comparisons of *PCC* and *Cint* on a VAX-11/780.

4. DEBUGGING FACILITIES

Cint provides the typical runtime checks and debugging facilities. It allows the user to set and clear breakpoints, trace the execution of the program at both the statement and function level, examine and dump the runtime stack, and examine and modify the value of variables. In addition, it includes a facility for producing a detailed (statement level) profile of a program's execution. On the Sun, *Cint* provides an interface to the window system that allows the user to watch the the flow of control through the program.

5. PERFORMANCE

Despite the use of a RISC virtual machine and its realization via a high-level language program, *Cint* provides performance comparable to interpreters using more complex machines and assembly language implementations. Two sets of benchmark data are presented. Table I compares *Cint*'s "load and go" execution performance to that of the portable C compiler (*PCC*) [8] on a number of standard Unix utilities. The benchmarks were performed on a VAX-11/780 with a floating-point accelerator. All optimizations were turned off for the *PCC* timings. This gave the fastest compile times, but slower execution speeds. Since compile time dominated for these benchmarks, this was more advantageous for *PCC*. The times reported are in seconds and are an average of five runs.

For the *banner*, *cal*, *cb*, *cmp*, and *echo* programs, the total time required by the interpreter is actually less than that required by *PCC*. *Cint*, however, was significantly slower on the *grep*, *tr*, and *wc* programs which were tested on reasonably large input files (more than 500 lines). When tested on smaller input files, the time required by the interpreter compares more favorably to that required by *PCC*. *Cint*'s ability to quickly compile and load programs makes it ideal for use in an instructional environment where compilation time typically dominates execution time.

We also compared *Cint*'s execution performance to the BSD Pascal *px* interpreter on both a VAX-11/780 and a Sun-3/75. This interpreter uses a complex virtual machine (over 100 opcodes) and is partially implemented using assembly language [10]. We tested both interpreters on a number of well-known benchmark programs [18, 20]. Tables II and III contain the results of the benchmark runs.

The execution times are reported in seconds with the exception of the dhrystone benchmark which is reported in number of dhrystones per second. The *cc* and *pc* timings were produced with all optimizations turned off. In addition, those timings were obtained by running larger problem instances and scaling the times appropriately. All timings were obtained by running the benchmarks five times on lightly-loaded machines. The times reported are the average of the five runs. As the tables show, the interpretation times for *Cint* and *px* compare favorably on both machines.

To remove any bias introduced by comparing the interpretation times of benchmarks written in different programming languages, we also computed the *interpretation efficiency* of *Cint* and *px*. The interpretation efficiency is the ratio of the execution time of a compiled program and the execution time of the pro-

Program	VAX-11/780					
	cc	Cint	pc	px	Cint/cc	pc/px
ackerman	4.70	58.70	6.06	72.11	12.4	11.9
bubblesort	8.20	328.58	12.40	286.95	40.0	23.0
matrixmult	4.18	96.06	5.95	89.90	22.9	15.1
puzzle	10.73	294.65	10.28	225.06	27.4	21.8
quicksort	0.13	4.96	0.21	4.08	37.3	18.9
shellsort	0.30	12.78	0.60	14.01	42.6	23.3
sieve	2.66	78.58	3.05	75.01	29.4	24.5
dhrystone†	1283.00	75.00	958.00	38.00	17.1	25.2
Average	–	–	–	–	28.6	20.4

Program	SUN-3/75					
	cc	Cint	pc	px	Cint/cc	pc/px
ackerman	1.06	38.48	1.20	53.95	36.1	44.9
bubblesort	4.58	224.50	2.96	231.30	48.9	77.9
matrixmult	2.26	64.96	1.71	59.81	28.6	34.8
puzzle	5.16	197.81	4.25	192.21	38.2	45.2
quicksort	0.08	3.21	0.05	3.86	38.7	77.3
shellsort	0.20	8.50	0.23	11.30	42.5	48.2
sieve	1.61	52.70	0.91	73.26	32.6	79.9
dhrystone†	2659.00	127.00	2222.00	60.00	20.9	37.0
Average	–	–	–	–	35.8	55.6

Tables II and III. Comparison of the Interpretation Efficiency of *Cint* and *px*.

gram when interpreted. These results are contained in the last two columns of Tables II and III. On the VAX-11/780, the interpretation efficiency of *Cint* and *px* averaged over the eight benchmark programs was 28.6 and 20.4 respectively. The average interpretation efficiency on the SUN-3/75 was 35.8 for *Cint* and 55.6 for *px*.

The difference between the interpretation efficiency of *px* on the VAX and the SUN is somewhat surprising. We conjecture that *px*'s implementation and its evolution is responsible for the large differences between its interpretation efficiency on the VAX and the SUN. The original *px* was written mostly in assembly language and ran on the PDP-11 computer family [9]. Version 2.0 was rewritten to run on the VAX-11 computer family [10]. The current version, 3.0, while maintaining the structure of the original versions, was rewritten in C. To obtain a working version of the interpreter and satisfactory performance, the assembly language produced by compiling the module that implements the virtual machine must be edited. For example, using knowledge of how the compiler generates code, the assembly language file is modified so that the interpreter can directly access the hardware-supported runtime stack. While such an implementation is more portable than one written entirely in assembly language, its performance is contingent on the ability to transform the assembly language produced by the C compiler to more efficient code. Our conjecture is that this technique works well for the VAX architecture and C compiler, but does not work as well for the Motorola 68020 architecture and C compiler on which a SUN-3/75 is based.

†The entries for dhrystone are dhrystones/second not seconds.

6. SUMMARY

RISC architectures offer several advantages over more complex architectures. They are easier to implement, they simplify code selection, and they support high-level languages at least as well. *Cint* shows that RISC design principles can be applied to the design of virtual machines for use in interpreters with similar benefits. The small size of the CVM's instruction set substantially reduced the effort required to construct a program to realize the CVM. Code generation was trivial because of the simple instruction set. Finally, our implementation compared favorably in performance to less portable implementations. While we did not find it necessary to implement the CVM using assembly language, a RISC virtual machine is attractive for conventional implementations also. The reduced number of instructions reduces the number of lines of assembly code that must be written.

7. REFERENCES

1. Bell, J. R., Threaded Code, *Communications of the ACM 16*, 6 (June 1973), 370-372.
2. Berry, R. E., Experience with the Pascal P-Compiler, *Software—Practice and Experience 8*, 5 (September 1978), 617-627.
3. Davidson, J. W. and Fraser, C. W., Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.
4. Dewar, R. B. K., Indirect Threaded Code, *Communications of the ACM 18*, 6 (June 1975), 330-331.
5. Dewar, R. B. K. and McCann, A. P., Macro Spitbol - a SNOBOL4 Compiler, *Software—Practice and Experience 7*, 1 (January 1977), 95-113.
6. Griswold, R. E., *The Macro Implementation of SNOBOL4*, W. H. Freeman and Co., San Fransico, CA, 1972.
7. Griswold, R. E., Linguistic Extension of Abstract Machine Modelling to Aid Software Development, *Software—Practice and Experience 10*, 1 (January 1980), 1-9.
8. Johnson, S. C., A Tour Through the Portable C Compiler, *Unix Programmer's Manual, 7th Edition 2B*, (January 1979), .
9. Joy, W. N. and McKusick, M. K., *Berkeley Pascal PX Implementation Notes Version 1.1—April, 1979*, University of California, Berkeley, April 1979.
10. Joy, W. N. and McKusick, M. K., *Berkeley Pascal PX Implementation Notes Version 2.0—January, 1979*, University of California, Berkeley, April 1981.
11. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
12. Klint, P., Interpretation Techniques, *Software—Practice and Experience 11*, 9 (September 1981), 963-973.
13. Newey, M. C., Poole, P. C. and Waite, W. M., Abstract Machine Modelling to Produce Portable Software, *Software—Practice and Experience 2*, (1972), 107-136.
14. Patterson, D. A. and Ditzel, D. R., The Case for the Reduced Instruction Set Computer, *Computer Architecture News 8*, 6 (October 1980), 25-33.
15. Patterson, D. A., Reduced Instruction Set Computers, *Communications of the ACM 28*, 1 (January 1985), 8-21.
16. Tanenbaum, A. S., Staveren, H. and Stevenson, J. W., Using Peephole Optimization on Intermediate Code, *ACM Transactions on Programming Languages and Systems 4*, 1 (January 1982), 21-36.
17. Tanenbaum, A. S., Staveren, H. V., Keizer, E. G. and Stevenson, J. W., A Practical Tool Kit for Making Portable Compilers, *Communications of the ACM 26*, 9 (September 1983), 654-660.
18. Object Code Quality Report Tartan C Compiler for Dec. VAX Berkeley Unix Software Distribution, TL-EXT-84-41, Tartan Laboratories, Pittsburgh, PA, November 1984.
19. Watts, J. S., *Construction of a Retargetable C Language Front-end*, Masters Thesis, University of Virginia, Charlottesville, VA, 1986.
20. Weicker, R. P., Dhrystone: A Synthetic Systems Programming Benchmark, *Communications of the ACM 27*, 10 (October 1984), 1013-1030 .

8. APPENDIX A

C Virtual Machine Instruction Set	
Arithmetic Operators	Description
++,--	addr←pop; v←pop; push(m[addr]); m[addr] *op*= v;
+=, -=, *=, /=, %=	addr←pop; v←pop; push(m[addr] *op*= v);
<<=, >>=, &=, ^=, |=	addr←pop; v←pop; push(m[addr] *op*= v);
+, -, *, /, %	v1←pop; v2←pop; push(v1 *op* v2);
<<, >>, &, ^, |	v1←pop; v2←pop; push(v1 *op* v2);
Unary Operators	Description
-, ~	v←pop; push(*op*v);
FLD *n*	v←pop; push(extract_field(v, *n*));
IFLD *n*	v←pop; push(insert_field(v, *n*));
Conversion Operators	Description
PCONV *ot nt*	Convert pointer of type *ot* to pointer of type *nt*.
VCONV *ot nt*	Convert value of type *ot* to type *nt*.
Data Movement	Description
CON *con*	push(*con*);
NAME *id class*	push(addr(*id*));
@	addr←pop; push(m[addr]);
=	addr←pop; v←pop; m[addr]←v
STASG *n*	dst←pop; src←pop; strncpy(dst, src, *n*);
Logical Test and Set	Description
==, >=, >, <=, <, !=	v1←pop; v2←pop; push(v1 *op* v2 ? 1 : 0);
Program Control and Jump	Description
CALL *nargs argsize*	addr←pop; push(environ); push(retaddr); pc←addr;
SWITCH *s r*	Switch stmt with starting value *s* and *r* consecutive values.
GOTO *n*	Jump to label *n*.
JT *n*	v1←pop; pc←v1 != 0 ? *n* : pc;
JF *n*	v1←pop; pc←v1 == 0 ? *n* : pc;
BRK *n*	Breakpoint.
FUNC *name n*	Define start of function. It requires *n* bytes of stack space.
RET	v←pop; addr←pop; pop(environ); push(v); pc←addr;
Argument Transmission	Description
PUSHA	addr←pop; pass(addr);
PUSHV	v←pop; pass(v)
STARG *n*	addr←pop; strncpy(v,addr,n); pass(v);
Pseudo Operations	Description
BGNBLK *level*	Begin block *level*.
BGNSTMT *n*	Begin code for statement *n*.
FILE *name*	Code was generated from source file *name*.
EFUNC *n*	End function that required *n* bytes for locals.
ENDBLK *level*	End block *level*.
EPDEF	End of prologue code for a procedure.
GBL *id class n*	Define global variable *id*. It requires *n* bytes.
DC *value*	Initialize a memory location with *value*.
LCL *id class n*	Define local variable *id*. It requires *n* bytes.
LLABEL *n*	Generate local label *n*.
PARM *id class n*	Define parameter *id*. It requires *n* bytes.
SLABEL *n*	Generate label for string.
SEG *n*	Signal start of segment *n*.

Notes:

1. Each executable opcode is followed by a type indicator.
2. ***Class*** denotes the scope of the variable (i.e., local, global, etc.).