Optimizing an ANSI C Interpreter with Superoperators

Todd A. Proebsting*
University of Arizona

Abstract

This paper introduces *superoperators*, an optimization technique for bytecoded interpreters. Superoperators are virtual machine operations automatically synthesized from smaller operations to avoid costly per-operation overheads. Superoperators decrease executable size and can double or triple the speed of interpreted programs. The paper describes a simple and effective heuristic for inferring powerful superoperators from the usage patterns of simple operators.

The paper describes the design and implementation of a hybrid translator/interpreter that employs superoperators. From a specification of the superoperators (either automatically inferred or manually chosen), the system builds an efficient implementation of the virtual machine in assembly language. The system is easily retargetable and currently runs on the MIPS R3000 and the SPARC.

1 Introduction

Compilers typically translate source code into machine language. Interpreter systems translate source into code for an underlying virtual machine (VM) and then interpret that code. The extra layer of indirection in an interpreter presents time/space tradeoffs. Interpreted code is usually slower than compiled code, but it can be smaller if the virtual machine operations are properly encoded.

Interpreters are more flexible than compilers. A compiler writer cannot change the target machine's instruction set, but an interpreter writer can customize the virtual machine. For instance, a virtual machine can be augmented with specialized operations that will allow the interpreter to produce smaller or faster code. Similarly, changing the interpreter implementation to monitor program execution (e.g., for debugging or profiling information) is usually easy.

This paper will describe the design and implementation of `hti`, a hybrid translator/interpreter system for ANSI C that has been targeted to both the MIPS R3000 [KH92], and the SPARC [Sun91]. `hti` will introduce *superoperators*, a novel optimization technique for customizing interpreted code for space and time. Superoperators automatically fold many atomic operations into a more efficient compound operation in a fashion similar to *supercombinators* in functional language implementations [FH88]. Without superoperators `hti` executables are only 8-16 times slower than unoptimized natively compiled code. Superoperators can lower this to a factor of 3-9. Furthermore, `hti` can generate *program-specific* superoperators automatically.

The hybrid translator, `hti`, *compiles* C functions into a tiny amount of assembly code for function prologue and interpreted bytecode instructions for function bodies. The bytecodes

*Address: Todd A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721. Internet: `todd@cs.arizona.edu`

POPL '95 1/ 95 San Francisco CA USA
© 1995 ACM 0-89791-692-1/95/0001....$3.50

represent the operations of the interpreter's virtual machine. By mixing assembly code and bytecodes, `hti` maintains all native code calling conventions; `hti` object files can be freely mixed with compiled object files.

The interpreter is implemented in assembly language for efficiency. Both the translator, `hti`, and the interpreter are quickly retargeted with a small machine specification.

2 Translator Output

`hti` uses `lcc`'s front end to translate ANSI C programs into its intermediate representation (IR) [FH91b, FH91a]. `lcc`'s IR consists of expression trees over a simple 109-operator language. For example, the tree for `2+3` would be `ADDI(CNSTI,CNSTI)`, where `ADDI` represents integer addition (`ADD+I`), and the `CNSTI`'s represent integer constants. The actual values of the `CNSTI`'s are IR node attributes.

`hti`'s virtual machine instructions are bytecodes (with any necessary immediate values). The interpreter uses an evaluation stack to evaluate all expressions. In the simplest `hti` virtual machines, there is a one-to-one correspondence between VM bytecodes and `lcc` IR operators (superoperators will change this). Translation is a left-to-right postfix emission of the bytecodes. Any necessary node attributes are emitted immediately after the corresponding bytecode. For this VM, the translation of `2+3` would be similar to the following:

```
.byte 36  # CNSTI
.word 2   #   immediate value
.byte 36  # CNSTI
.word 3   #   immediate value
.byte 8   # ADDI
```

The interpreter implements operations via a jump-table indexed by bytecodes. The interpreter reads the first `CNSTI`'s bytecode (36), and jumps to `CNSTI`'s implementation. `CNSTI` code reads the attribute value (2) and pushes it on the stack. The interpreter similarly handles the "3." After reading the `ADDI` bytecode, the interpreter pops two integers off the evaluation stack, and pushes their sum.

The evaluation stack for each translated procedure exists in its own activation record. Local stacks allow programs to behave correctly in the presence of interprocedural jumps (e.g., `longjmp`).

`hti` produces an assembly file. Most of the file consists of the bytecode translation of C function bodies, and data declarations. `hti` does, however, produce a tiny amount of assembly language for function prologues. Prologue code tells the interpreter how big the activation record should be, where within it to locate the evaluation stack, where to find the bytecode instructions, and ultimately for transferring control to the interpreter. A prologue on the R3000 looks like the following:

```
main:
li $24, 192          # put activation
                     # record size in $24
li $8, 96            # put location of
                     # evaluation stack in $8
la $25, $$11         # put location of
                     # bytecode in $25
j _prologue_scalar # jump to interpreter
```

(`_prologue_scalar` unloads scalar arguments onto the stack — the R3000 calling conventions require a few different such prologue routines. Once the arguments are on the stack, the interpreter is started.) Prologue code allows natively compiled procedures to call interpreted procedures without modification.

3 Superoperator Optimization

Compiler front ends, including `lcc`, produce many IR trees that are very similar in structure. For instance, `ADDP(INDIRP(x),CNSTI)` is the most common 3-node IR pattern produced by `lcc` when it compiles itself. (`x` is a placeholder for a subtree.) This pattern computes a pointer value that is a constant offset from the value pointed to by `x` (i.e., the l-value of `x->b` in C).

With only simple VM operators, translating `ADDP(INDIRP(x),CNSTI)` requires emitting three bytecodes and the `CNSTI`'s attribute. Interpreting those instructions requires

1. Reading the `INDIRP` bytecode, popping `x`'s value off the stack, fetching and pushing the referenced value,

2. Reading the `CNSTI` bytecode and attribute, and pushing the attribute,

3. Reading the `ADDP` bytecode, popping the two just-pushed values, computing and pushing their sum.

If the pattern `ADDP(INDIRP(x),CNSTI)` were a single operation that takes a single operand, `x`, the interpreter avoids 2 bytecode reads, 2 pushes, and 2 pops. This new operator would have one attribute — the value of the embedded `CNSTI`. These synthetic operators are called *superoperators.*

Superoperators make interpreters faster by eliminating pushes, pops, and bytecode reads. Furthermore, superoperators decrease code size by eliminating bytecodes. The cost of a superoperator is an additional bytecode, and a correspondingly larger interpreter. Experiments in §8 show that carefully chosen superoperators result in smaller and significantly faster interpreted code.

3.1 Inferring Superoperators

Superoperators can be designed to optimize the interpreter over a wide range of C programs, or for a specific program. The `lcc` IR includes only 109 distinct operators, thus leaving 147 bytecodes for superoperators. Furthermore, if the interpreter is being built for a specific application, it may be possible to remove many operations from the VM if they are never generated in the translation of the source program (e.g., floating point operations), thereby allowing the creation of even more superoperators.

The addition of superoperators increases the size of the interpreter, but this can be offset by the corresponding reduction of emitted bytecodes. Specific superoperators may optimize for space or time. Unfortunately, choosing the optimal set of superoperators for space reduction is NP-complete — External Macro Data Compression (SR22 [GJ79]) reduces to this problem. Similarly, optimizing for execution time is equally complex.

3.1.1 Inference Heuristic

`hti` includes a heuristic method for inferring a good set of superoperators. The heuristic reads a file of IR trees, and then decides which adjacent IR nodes should be merged to form new superoperators. Each tree is weighted to guide the heuristic. When optimizing for space, the weight is the number of times each tree is emitted by the front end of `lcc`. When optimizing for time, the weight is each tree's (expected) execution frequency.

A simple greedy heuristic creates superoperators. The heuristic exams all the input IR trees to isolate all pairs of adjacent (parent/child) nodes. Each pair's weight is the sum of the weights of the trees in which it appears. (If the same pair appears N times in the same tree, that tree's weight is counted N times.) The pair with the greatest cumulative weight becomes the superoperator formed by merging that pair. This new superoperator then replaces all occurrences of that pair in the input trees. For example, assume that the input trees with weights are

```
I(A(Z,Y))  10
A(Y,Y)      1
```

The original operators's frequencies of use are

```
Y          12
Z          10
I          10
A          11
```

The frequencies of the parent/child pairs are

```
I(A(*))  10
A(Z,*)   10
A(*,Y)   11
A(Y,*)    1
```

Therefore, `A(*,Y)` would become a new superoperator, `B`. This new *unary* operator will replace the occurrences of `A(*,Y)` in the subject trees. The resulting trees are

```
I(B(Z))  10
B(Y)      1
```

The new frequencies of parent/child pairs are

```
I(B(*))   10
B(Z)      10
B(Y)       1
```

Repeating the process, a new superoperator would be created for either `I(B(*))` or `B(Z)`. Ties are broken arbitrarily, so assume that `B(Z)` becomes the new leaf operator, `C`. Note that `C` is simply the composition of `A(Z,Y)`. The rewritten trees are

```
I(C)      10
B(Y)       1
```

The frequencies for the bytecodes is now

```
Y          1
Z          0
I         10
A          0
B          1
C         10
```

It is interesting to note that the `B` superoperator is used only once now despite being present in 11 trees earlier. Underutilized superoperators inhibit the creation of subsequent superoperators by using up bytecodes and hiding constituent pieces from being incorporated into other superoperators. Unfortunately, attempting to take advantage of this observation by breaking apart previously created, but underutilized superoperators was complicated and ineffective.

Creating the superoperators `B` and `C` eliminated the last uses of the operators `A` and `Z`, respectively. The heuristic can take advantage of this by reusing those operators's bytecodes for new superoperators. The process of synthesizing superoperators repeats until exhausting all 256 bytecodes. The heuristic may, of course, merge superoperators together.

The heuristic implementation requires only 204 lines of Icon [GG90]. The heuristic can be configured to eliminate obsolete operators (i.e., reuse their bytecodes), or not, as superoperators are created. Not eliminating obsolete operators allows the resulting translator to process all programs, even though not specifically optimized for them.

4 Translator Design

4.1 Bytecode Emitter

`hti` translates `lcc`'s IR into bytecodes and attributes. Bytecodes can represent simple IR operators, or complex superoperator patterns. The optimal translation of an IR tree into bytecodes is automated via tree pattern matching using `burg` [FHP92]. `burg` takes a cost-augmented set of tree patterns, and creates an efficient pattern matcher that finds the least-cost cover of a subject tree. Patterns describe the actions associated with bytecodes. Some sample patterns in the `burg` specification, `interp.gr`, follow:

```
stk : ADDP(INDIRP(stk),CNSTI)   = 5 (1) ;
stk : ADDP(stk,stk)             = 9 (1) ;
stk : CNSTI                     = 36 (1) ;
stk : INDIRP(stk)               = 77 (1) ;
```

The nonterminal `stk` represents a value that resides on the stack. The integers after the `=`'s are the `burg` rule numbers, and, are also the actual bytecodes for each operation. Rule 9, for example, is a VM instruction that pops two values from the stack, adds them, and pushes the sum onto the stack. The `(1)`'s represent that each pattern has been assigned a cost of 1. The pattern matcher would choose to use rule 5 (at cost 1) over rules 9, 36, and 77 (at cost 3) whenever possible.

The `burg` specification for a given VM is generated automatically from a list of superoperator patterns. To change the superoperators of a VM — and its associated translator and interpreter — one simply adds or deletes patterns from this list and then re-builds `hti`. `hti` can be built with inferred, or hand-chosen superoperators.

4.2 Attribute Emitter

`hti` must emit node attributes after appropriate bytecodes. In the previous example, it is necessary to emit the integer attribute of the `CNSTI` node immediately after emitting the bytecodes for rules 5 or 36. This is simple for single operators, but superoperators

may need to emit many attributes. The pattern ADDI(MULI(x,CNSTI),CNSTI) requires two emitted attributes — one for each CNSTI.

To build hti, a specification associates attributes with IR operators. A preprocessor builds an attribute emitter for each superoperator. The attribute specification for CNSTI is

```
reg: CNSTI = (1)
  "emitsymbol(%P->syms[0]->x.name, 4, 4);"
```

The pattern on the first line indicates that the interpreter will compute the value of the CNSTI into a register at cost 1. The second line indicates that the translator emits a 4-byte value that is 4-byte aligned. The preprocessor expands %P to point to the CNSTI node relative to the root of the superoperator in which it exists. %P->syms[0]->x.name is the emitted value. For the simple operator, stk: CNSTI, the attribute emitter executes the following call after emitting the bytecode

```
emitsymbol(p->syms[0]->x.name, 4, 4);
```

where p points to the CNSTI. For stk: ADDP(INDIRP(STK), CNSTI), the attribute emitter executes

```
emitsymbol(p->kids[1]->syms[0]->x.name,
   4, 4);
```

where p->kids[1] points to the CNSTI relative to the root of the pattern, ADDP.

A preprocessor creates a second burg specification, mach.gr, from the emitter specification. The emitter specification patterns form the rules in mach.gr. The mach.gr-generated pattern matcher processes *trees* that represent the VM's superoperators. For every emitter pattern that matches in a superoperator tree, the associated emitter action must be included in the translator for that superoperator. This is done automatically from the emitter specification and the list of superoperator trees. (Single node VM operators are always treated as degenerate superoperators.) Automating the process of translating chosen superoperators to a new interpreter is key to practically exploiting superoperator optimizations.

5 Interpreter Generation

The interpreter is implemented in assembly language. Assembly language enables important optimizations like keeping the evaluation stack pointer and interpreter program counter in hardware registers. Much of the interpreter is automatically generated from a target machine specification and the list of superoperators. The target machine specification maps IR nodes (or patterns of IR nodes) to assembly language. For instance, the mapping for ADDI on the R3000 is

```
reg: ADDI(reg,reg) = (1)
      "addu %0r, %1r, %2r\n"
```

This pattern indicates that integer addition (ADDI) can be computed into a register if the operands are in registers. %0r, %1r, and %2r represent the registers for the left-hand side nonterminal, the left-most right-hand side nonterminal, and the next right-hand side nonterminal, respectively.

The machine specification augments the emitter specification described above — they share the same patterns. Therefore, they can share the same burg-generated pattern matcher. The pattern matcher processes superoperator trees to determine how to best translate each into machine code. Below is a small specification to illustrate the complete translation for an ADDI operator.

```
reg: ADDI(reg,reg) = (1)
      "addu %0r, %1r, %2r\n"

reg: STK = (1)
      "lw %0r, %P4-4($19)\n"

stmt: reg = (0)
      "sw %0r, %U4($19)\n"
```

STK is a terminal symbol representing a value on the stack. The second rule is a *pop* from the evaluation stack into a register. %P4 is a 4-byte pop, and $19 is the evaluation stack pointer register. The third rule is a *push* onto the evaluation stack from a register. %U4 is the 4-byte push.

To generate the machine code for a simple ADDI operation, the interpreter-generator reduces the tree ADDI(STK,STK) to the nontermi-

nal stmt using the pattern matcher. The resulting code requires two instances of the second rule, and one each of the first and third rules:

```
lw $8, 0-4($19)     # pop left operand
                    # (reg: STK)
lw $9, -4-4($19)    # pop right operand
                    # (reg: STK)
addu $8, $8, $9     # add them
                    # (reg: ADDI(reg,reg))
sw $8, -8($19)      # push the result
                    # (stmt: reg)
addu $19, -4        # adjust stack
```

The interpreter-generator automatically allocates registers $8 and $9 and, generates code to adjust the evaluation stack pointer.

The interpreter-generator selects instructions and allocates temporary registers for each superoperator. In essence, creating an interpreter is traditional code generation — except that it is done for a static set of IR trees *before* any source code is actually translated.

The emitter and machine specifications use the same patterns, so only one file is actually maintained. The juxtaposition of the emitter code and machine code makes their relationship explicit. Below is the complete R3000 specification for CNSTI.

```
reg: CNSTI = (1)
   "addu $17, 7;
    srl $17, 2;
    sll $17, 2;
    lw %0r, -4($17)\n"
   "emitsymbol(%P->syms[0]->x.name, 4, 4);"
```

Register $17 is the interpreter's *program counter* (pc). The first three instructions advance the pc past the 4-byte immediate data and round the address to a multiple of 4. (Because of assembler and linker constraints on the R3000, all 4-byte data must be word aligned.) The lw instruction loads the immediate value into a register.

Machine/emitter specifications are not limited to single-operator patterns. Complex IR tree patterns may better express the relationship between target machine instructions and lcc's IR. For example, the R3000 lb instruction loads and sign-extends a 1-byte value into a 4-byte register. This corresponds to the IR pattern, CVCI(INDIRC(x)). The specification for this complex pattern follows.

```
reg: CVCI(INDIRC(reg)) = (1)
      "lb %0r, 0(%1r)\n"
      ""
```

The interpreter-generator may use this rule for any superoperators that include CVCI(INDIRC(x)).

5.1 Additional IR Operator

To reduce the size of bytecode attributes, one additional IR operator was added to lcc's original set: ADDRb. lcc's ADDRLP node represents the offset of a local variable relative to the frame pointer. hti emits a 4-byte offset attribute for ADDRLP. ADDRLb is simply an abbreviated version of ADDRLP that requires only a 1-byte offset. Machine-independent back end code does this translation.

6 Implementation Details

Building an hti interpreter is a straightforward process. The following pieces are needed to build hti's translator and interpreter:

- A target machine/emitter specification.
- lcc back end code to handle data layout, calling conventions, etc.
- A library of interpreter routines for observing calling conventions.
- Machine-dependent interpreter-generator routines.

Figure 1 summarizes the sizes of the machine dependent and independent parts of the system (lcc's front end is excluded).

The R3000-specific back end code and the interpreter library are much bigger than the SPARC's because of the many irregular argument passing conventions observed by C code on the R3000.

Function	Language	Sizes (in lines)		
		Machine Independent	R3000	SPARC
Target Specification	grammar	-	351	354
`lcc` back end	C	434	244	170
interpreter library	asm	-	130	28
interpreter generator	C	204	72	70

Figure 1: Implementation Details

7 System Obstacles

Unfortunately, `hti`'s executables are slower and bigger than they ought to be because of limitations of system software on both R3000 and SPARC systems. The limitations are not intrinsic to the architectures or `hti`; they are just the results of inadequate software.

Neither machine's assembler supports unaligned initialized data words or halfwords. This can cause wasted space between a bytecode and its (aligned) immediate data. Consequently, the interpreter must execute additional instructions to round its `pc` up to a 4-byte multiple before reading immediate 4-byte data. Initial tests indicate that approximately 17% of the bytes emitted by `hti` are wasted because of alignment problems.[1]

The R3000 assembler restricts the ability to emit position-relative initialized data. For instance, the following is illegal on the R3000:

```
L99:
        .word 55
        .word .-L99
```

Position relative data would allow `hti` to implement pc-relative jumps and branches. Pc-relative jumps can use 2-byte immediate values rather than 4-byte absolute addresses, thus saving space.

8 Experimental Results

`hti` compiles C source into object code. Object code for each function consists of a native-code prologue, with interpreted bytecodes for the function body. Object files are linked together with appropriate C libraries (e.g., `libc.a`) and the interpreter. The executable may be compared to natively compiled code for both size and speed. The code size includes the function prologues, bytecodes, and one copy of the interpreter. Interpreter comparisons will depend on available superoperators.

Comparisons were made for three programs:

- `burg`: A ~5,000-line tree pattern matcher generator, processing a 136-rule specification.
- `hti`: The ~13,000-line translator and subject of this paper, translating a 1117-line C file.
- `loop`: An empty `for` loop that executes 10,000,000 times.

On the 33MHz R3000, `hti` is compared to a production quality `lcc` compiler. Because `lcc`'s SPARC code generator is not available, `hti` is compared to `acc`, Sun's ANSI C compiler, on the 33MHz Sun 4/490. Because `lcc` does little global optimization, `acc` is also run without optimizations. `hti` is run both with and without enabling the superoperator optimization. Superoperators are inferred based on a static count of how many times each tree is emitted from the front end for that benchmark — `hti+so` represents these tests. The columns labelled `hti` represent the interpreter built with exactly one VM operator for each IR operator.

Figures 2 and 3 summarize the sizes of the code segments for each benchmark. "code" is the total of bytecodes, function prologues, and wasted

[1] I understand that the latest release of the R3000 assembler and linker supports unaligned initialized data, and that the R3000 has instructions for reading unaligned data. Unfortunately, I do not have access to these new tools.

space. "waste" is the portion wasted due to alignment restrictions. "interp" is the size of the interpreter. (The sizes do not include linked system library routines since all executables would use the same routines.)

The interpreted executables are slightly larger than the corresponding native code. The interpreted executables are large for a few reasons besides the wasteful alignment restrictions already mentioned. First, no changes were made to the `lcc`'s IR except the addition of `ADDRb`, and `lcc` creates wasteful IR nodes. For instance, `lcc` produces a `CVPU` node to convert a pointer to an unsigned integer, yet this a nop on both the R3000 and SPARC. Removing this node from IR trees would reduce the number of emitted bytecodes. Additionally, `lcc` produces IR nodes that require the same code sequences on most machines, like pointer and integer addition. Distinguishing these nodes hampers superoperator inference, and superoperators save space. Unfortunately, much of the space taken up by executables is for immediate values, not operator bytecodes. To reduce this space would require either encoding the sizes of the immediate data in new operators (like `ADDRb`) or tagging the data with size information, which would complicate fetching the data.

Fortunately, `hti` produces extremely fast interpreters. Figures 4 and 5 summarize the execution times for each benchmark.

`lcc` does much better than `acc` relative to interpretation because it does modest global register allocation, which `acc` and `hti` do not do. `lcc`'s code is 28.2 times faster than the interpreted code on `loop` because of register allocation. Excluding the biased `loop` results, interpreted code without superoperators is less than 16 times slower than native code — sometimes significantly. Furthermore, superoperators consistently increase the speed of the interpreted code by 2-3 times.

These results can be improved with more engineering and better software. Support for unaligned data would make all immediate data reads faster. Inferring superoperators based on profile information rather than static counts would make them have a greater effect on execution efficiency.

If space were not a consideration, the interpreter could be implemented in a directly threaded fashion to decrease operator decode time [Kli81]. The implementation of each VM operator is unrelated to the encoding of the operators, so changing from the current indirect table lookup to threading would not be difficult.

9 Limitations and Extensions

Almost certainly, each additional superoperator contributes decreasing marginal returns. I made no attempt to determine what the time and space tradeoffs would be if the number of superoperators were limited to some threshold like 10 or 20. I would conjecture that the returns *for a given application* diminish very quickly and that 20 superoperators realize the bulk of the potential optimization. The valuable superoperators for numerically intensive programs probably differ from those for pointer intensive programs. To create a single VM capable of executing many classes of programs efficiently, the 147 additional bytecodes could be partitioned into superoperators targeted to different representative classes of applications.

This system's effectiveness is limited by `lcc`'s trees. The common C expression, `x ? y`, cannot be expressed as a single tree by `lcc`. Therefore, `hti` cannot infer superoperators to optimize its evaluation based on the IR trees generated by the front end. Of course, any scheme based on looking for common tree patterns will be limited by the operators in the given intermediate language.

`hti` generates bytecodes as `.data` assembler directives and function prologues as assembly language instructions. Nothing about the techniques described above is limited to such an implementation. The bytecodes could have been emitted into a simple array that would be immediately interpreted, much like in a traditional interpreter. This would require an additional bytecode to represent the *prologue* of a function — to mimic the currently executed assembly instructions. To make this work, the system would have to resolve references within the bytecode,

R3000 Code Size Summary (in bytes)							
Benchmark	Translator						
	lcc	hti			hti+so		
	code	code	interp	waste	code	interp	waste
burg	56576	92448	4564	15895	72616	12388	13862
hti	230160	315040	4564	51868	289516	11296	64299
loop	48	52	4564	4	44	600	6

Figure 2: R3000 Benchmark Code Sizes

SPARC Code Size Summary (in bytes)							
Benchmark	Translator						
	acc	hti			hti+so		
	code	code	interp	waste	code	interp	waste
burg	75248	84720	4080	13560	63992	11840	10862
hti	271736	292568	4080	41423	254808	10512	37507
loop	80	56	4080	2	48	312	4

Figure 3: SPARC Benchmark Code Sizes

R3000 Execution Summary						
Benchmark	Times (in seconds)			Ratios		
	lcc	hti	hti+so	hti/lcc	hti+so/lcc	hti/hti+so
burg	1.65	14.04	7.07	8.5	4.3	2.0
hti	2.69	42.83	23.81	15.9	8.8	1.8
loop	1.53	43.12	13.88	28.2	9.1	3.1

Figure 4: R3000 Benchmark Code Speeds

SPARC Execution Summary						
Benchmark	Times (in seconds)			Ratios		
	acc	hti	hti+so	hti/acc	hti+so/acc	hti/hti+so
burg	1.78	18.52	8.37	10.4	4.7	2.2
hti	4.39	58.24	28.73	13.3	6.5	2.0
loop	6.62	61.26	20.05	9.3	3.0	3.1

Figure 5: SPARC Benchmark Code Speeds

which would require some additional machine-independent effort. (The system linker/loader resolves references in the currently generated assembler.) Not emitting function prologues of machine instructions would make seamless calls between interpreted and compiled functions very tricky, however.

10 Related Work

Many researchers have studied interpreters for high-level languages. Some were concerned with interpretation efficiency, and others with the diagnostic capabilities of interpretation.

Supercombinators optimize combinator-based functional-language interpreters in a way similar to how superoperators optimize `hti`. Supercombinators are combinators that encompass the functionality of many smaller combinators [FH88]. By combining functionality into a single combinator, the number of combinators to describe an expression is reduced and the number of function applications necessary to evaluate an expression is decreased. This is analogous to reducing the number of bytecodes emitted and fetched through superoperator optimization.

Pittman developed a hybrid interpreter and native code system to balance the space/time tradeoff between the two techniques [Pit87]. His system provided hooks for escaping interpreted code to execute time-critical code in assembly language. Programmers coded directly in both interpreted operations, or assembly.

Davidson and Gresch developed a C interpreter, Cint, that, like `hti`, maintained C calling conventions in order to link with native code routines [DG87]. Cint was written entirely in C for easy retargetability. Cint's VM is similar to `hti`'s — it includes a small stack-based operator set. On a set of small benchmarks the interpreted code was 12.4-42.6 times slower than native code on a VAX-11/780, and 20.9-42.5 times slower on a Sun-3/75. Executable sizes were not compared.

Kaufer, *et. al.*, developed a diagnostic C interpreter environment, Saber-C, that performs approximately 70 run-time error checks [KLP88]. Saber-C's interpreted code is roughly 200 times slower than native code, which the authors attribute to the run-time checks. The interpreter implements a stack-based machine, and maintains calling conventions between native and interpreted code. Unlike `hti` interpreted functions have two entry points: one for being called from other interpreted functions, and another for native calls, with a machine-code prologue.

Similarly, Feuer developed a diagnostic C interpreter, `si`, for debugging and diagnostic output [Feu85]. `si`'s primary design goals were quick translation and flexible diagnostics — time and space efficiency were not reported.

Klint compares three ways to encode a program for interpretation [Kli81]. The methods are "Classical," "Direct Threaded" [Bel73], and "Indirect Threaded." Classical — employed by `hti` and Cint — encodes operators as values such that address of the corresponding interpreter code must be looked up in a table. Direct Threaded encodes operations with the addresses of the corresponding interpreter code. Indirect Threaded encodes operations with pointers to locations that hold the actual code addresses. Klint concludes that the Classical method gives the greatest compaction because it is possible to use bytes to encode values (or even to use Huffmann encoding) to save space. However, the Classical method requires more time for the table lookup.

11 Discussion

`hti` translates ANSI C into tight, efficient code that includes a small amount of native code with interpreted code. This hybrid approach allows the object files to maintain all C calling conventions so that they may be freely mixed with natively compiled object files. The interpreted object code is approximately the same size as equivalent native code, and runs only 3-16 times slower.

Much of the interpreter's speed comes from being implemented in assembly language. Retargeting the interpreter is simplified using compiler-writing tools like `burg` and special-purpose machine specifications. For the MIPS R3000 and the SPARC, each machine required fewer than 800 lines of machine-specific code to

be retargeted.

Superoperators, which are VM operations that represent the aggregate functioning of many connected simple operators, make the interpreted code both smaller and faster. Tests indicate superoperators can double or triple the speed of interpreted code. Once specified by the interpreter developer, new superoperators are automatically incorporated into both the translator and the interpreter. Furthermore, heuristics can automatically isolate beneficial superoperators from static or dynamic feedback information for a specific program or for an entire suite of programs.

12 Acknowledgements

Chris Fraser provided useful input on this work.

References

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.

[DG87] J. W. Davidson and J. V. Gresch. Cint: A RISC interpreter for the C programming language. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 189–198, June 1987.

[Feu85] Alan R. Feuer. si — an interpreter for the C language. In *Proceedings of the 1985 Usenix Summer Conference, Portland, OR*, June 1985.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[FH91a] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, September 1991.

[FH91b] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10), October 1991.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[GG90] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice Hall, 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11(10):963–973, October 1981.

[KLP88] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C: An interpreter-based programming environment for the C language. In *Proceedings of the 1988 Usenix Summer Conference, San Francisco, CA*, June 1988.

[Pit87] T. Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 150–152, June 1987.

[Sun91] Sun Microsystems, Inc. *The SPARC Architecture Manual (Version 8)*, 1991.