‘C and tcc: A Language and Compiler for Dynamic Code Generation

Massimiliano Poletto
Laboratory for Computer Science, Massachusetts Institute of Technology
and
Wilson C. Hsieh
Department of Computer Science, University of Utah
and
Dawson R. Engler
Laboratory for Computer Science, Massachusetts Institute of Technology
and
M. Frans Kaashoek
Laboratory for Computer Science, Massachusetts Institute of Technology

Dynamic code generation allows programmers to use run-time information in order to achieve performance and expressiveness superior to those of static code. The ‘C (*Tick C*) language is a superset of ANSI C that supports efficient and high-level use of dynamic code generation. ‘C provides dynamic code generation at the level of C expressions and statements, and supports the composition of dynamic code at run time. These features enable programmers to add dynamic code generation to existing C code incrementally, and to write important applications (such as “just-in-time” compilers) easily. The paper presents many examples of how ‘C can be used to solve practical problems.

The tcc compiler is an efficient, portable, and freely available implementation of ‘C. tcc allows programmers to trade dynamic compilation speed for dynamic code quality: in some applications, it is most important to generate code quickly, while in others code quality matters more than compilation speed. The overhead of dynamic compilation is on the order of 100 to 600 cycles per generated instruction, depending on the level of dynamic optimization. Measurements show that the use of dynamic code generation can improve performance by almost an order of magnitude; two- to four-fold speedups are common. In most cases, the overhead of dynamic compilation is recovered in under 100 uses of the dynamic code; sometimes it can be recovered within one use.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*compilers; code generation; run-time environments*

General Terms: Algorithms, Languages, Performance

Additional Key Words and Phrases: compilers, dynamic code generation, dynamic code optimization, ANSI C

Email: maxp@lcs.mit.edu, wilson@cs.utah.edu, engler@lcs.mit.edu, kaashoek@lcs.mit.edu. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. The second author can be reached at: University of Utah, Computer Science, 50 S Central Campus Drive, Room 3190, Salt Lake City, UT 84112-9205.

This research was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and N66001-96-C-8522, and by a NSF National Young Investigator Award.

1. INTRODUCTION

Dynamic code generation — the generation of executable code at *run time* — enables the use of run-time information to improve code quality. Information about run-time invariants provides new opportunities for classical optimizations such as strength reduction, dead code elimination, and inlining. In addition, dynamic code generation is the key technology behind just-in-time compilers, compiling interpreters, and other components of modern mobile code and other adaptive systems.

'C is a superset of ANSI C that supports the high-level and efficient use of dynamic code generation. It extends ANSI C with a small number of constructs that allow the programmer to express dynamic code at the level of C expressions and statements, and to *compose* arbitrary dynamic code at run time. These features enable programmers to write complex imperative code manipulation programs in a style similar to Lisp [Steele Jr. 1990], and make it relatively easy to write powerful and portable dynamic code. Furthermore, since 'C is a superset of ANSI C, it is not difficult to improve performance of code incrementally by adding dynamic code generation to existing C programs.

'C's extensions to C — two type constructors, three unary operators, and a few special forms — allow dynamic code to be type-checked statically. Much of the overhead of dynamic compilation can therefore be incurred statically, which improves the efficiency of dynamic compilation. While these constructs were designed for ANSI C, it should be straightforward to add analogous constructs to other statically typed languages.

tcc is an efficient and freely available implementation of 'C, consisting of a front end, back ends that compile to C and to MIPS and SPARC assembly, and two runtime systems. tcc allows the user to trade dynamic code quality for dynamic code generation speed. If compilation speed must be maximized, dynamic code generation and register allocation can be performed in one pass; if code quality is most important, the system can construct and optimize an intermediate representation prior to code generation. The overhead of dynamic code generation is approximately 100 cycles per generated instruction when tcc only performs simple dynamic code optimization, and approximately 600 cycles per generated instruction when all of tcc's dynamic optimizations are turned on.

This paper makes the following contributions:

—It describes the 'C language, and motivates the design of the language.

—It describes tcc, with special emphasis on its two runtime systems, one tuned for code quality and the other for fast dynamic code generation.

—It presents an extensive set of 'C examples, which illustrate the utility of dynamic code generation and the ease of use of 'C in a variety of contexts.

—It analyzes the performance of tcc and tcc-generated dynamic code on several benchmarks. Measurements show that use of dynamic compilation can improve performance by almost an order of magnitude in some cases, and generally results in two- to four-fold speedups. The overhead of dynamic compilation is usually recovered in under 100 uses of the dynamic code; sometimes it can be recovered within one use.

The rest of this paper is organized as follows. Section 2 describes 'C, and Section 3 describes tcc. Section 4 illustrates several sample applications of ‘C. Section 5 presents performance measurements. Finally, we discuss related work in Section 6, and summarize our conclusions in Section 7. Appendix A describes the ‘C extensions to the ANSI C grammar.

2. THE ‘C LANGUAGE

The ‘C language was designed to support easy-to-use dynamic code generation in a systems and applications programming environment. This requirement motivated some of the key features of the language:

—‘C is a small extension of ANSI C: it adds very few constructs — two type constructors, three unary operators, and a few special forms – and leaves the rest of the language intact. As a result, it is possible to convert existing C code to ‘C incrementally.

—Dynamic code in ‘C is statically typed. This is consistent with C, and improves the performance of dynamic compilation by eliminating the need for dynamic type-checking. The same constructs used to extend C with dynamic code generation should be applicable to other statically typed languages.

—The dynamic compilation process is imperative: the ‘C programmer directs the creation and composition of dynamic code. This approach distinguishes ‘C from several recent declarative dynamic compilation systems [Auslander et al. 1996; Grant et al. 1997; Consel and Noel 1996]. We believe that the imperative approach is better suited to a systems environment, where the programmer wants tight control over dynamically generated code.

In ‘C, a programmer creates *code specifications*, which are static descriptions of dynamic code. Code specifications can capture the values of run-time constants, and they may be composed at run time to build larger specifications. They are compiled at run time to produce executable code. The process works as follows:

(1) At *static compile time*, a ‘C compiler processes the written program. For each code specification, the compiler generates code to capture its run-time environment, as well as code to generate dynamic code.

(2) At run time, code specifications are evaluated. The specification captures its run-time environment at this time, which we call *environment binding time*.

(3) At run time, code specifications are passed to the ‘C compile special form. compile invokes the specification's code generator, and returns a function pointer. We call this point in time *dynamic code generation time*.

For example, the following code fragment implements “Hello World” in ‘C:

```
void make_hello(void) {
    void (*f)() = compile(‘{ printf(“hello, world\n”); }, void);
    (*f)();
}
```

The code within the backquote and braces is a code specification for a call to printf that should be generated at run time. The code specification is evaluated (environment binding time), and the resulting object is then passed to compile, which generates executable code for the printf and returns a function pointer (dynamic code generation time). The function pointer can then be invoked directly.

The rest of this section describes in detail the dynamic code generation extensions introduced in ‘C. Section 2.1 describes the ‘ operator. Section 2.2 describes the type constructors in ‘C. Section 2.3 describes the unquoting operators, @ and $. Section 2.4 describes the ‘C special forms.

2.1 The ‘ Operator

The ‘ (backquote, or “tick”) operator is used to create dynamic code specifications in ‘C. ‘ can be applied to an expression or compound statement, and indicates that code corresponding to that expression or statement should be generated at run time. ‘C disallows the dynamic generation of code-generating code, so ‘ does not nest. In other words, a code specification cannot contain a nested code specification. Some simple usages of backquote are as follows:

```
/* Specification of dynamic code for the expression “4”: results
   in dynamic generation of code that produces 4 as a value */
‘4

/* Specification of dynamic code for a call to printf;
   j must be declared in an enclosing scope */
‘printf(“%d”, j)

/* Specification of dynamic code for a compound statement */
‘{ int i; for (i = 0; i < 10; i++) printf(“%d\n”, i); }
```

Dynamic code is lexically scoped: variables in static code can be referenced in dynamic code. Lexical scoping and static typing allow type-checking and some instruction selection to occur at static compile time, decreasing dynamic code generation overhead.

The value of a variable after its scope has been exited is undefined, just as in ANSI C. In contrast to ANSI C, however, not all uses of variables outside their scope can be detected statically. For example, one may use a local variable declared in static code from within a backquote expression, and then return the value of the code specification. When the code specification is compiled, the resulting code references a memory location that no longer contains the local variable, because the original function’s activation record has gone away. The compiler could perform a data-flow analysis to conservatively warn the user of a potential error; however, in our experience this situation arises very rarely, and is easy to avoid.

The use of several C constructs is restricted within backquote expressions. In particular, a break, continue, case, or goto statement cannot be used to transfer control outside the enclosing backquote expression. For instance, the destination label of a goto statement must be contained in the same backquote expression that contains the goto. ‘C provides other means for transferring control between backquote expressions; we discuss these methods in Section 2.4. The limitation on goto and other control-transfer statements enables a ‘C compiler to statically determine whether a control-flow change is legal. The use of return is not restricted, because dynamic code is always implicitly inside a function.

A backquote expression can be dynamically compiled using the compile special form, which is described in Section 2.4. compile returns a function pointer, which can then be invoked like any other function pointer.

2.2 Type Constructors

‘C introduces two new type constructors, cspec and vspec. cspecs are static types for dynamic code; their presence allows dynamic code to be type-checked statically. vspecs are statics types for dynamic lvalues (expressions that may be used on the left hand side of an assignment); their presence allows dynamic code to allocate lvalues as needed.

A cspec or vspec has an associated *evaluation type*, which is the type of the dynamic value of the specification. The evaluation type is analogous to the type to which a pointer points.

2.2.1 cspec *Types.* cspec (short for *code specification*) is the type of a dynamic code specification; the evaluation type of the cspec is the type of the dynamic value of the code. For example, the type of the expression ‘4 is int cspec. The type void cspec is the type of a generic cspec type (analogous to the use of void * as a generic pointer).

Applying ‘ to a statement or compound statement yields an expression of type void cspec. In particular, if the dynamic statement or compound statement contains a return statement, the type of the return value does not affect the type of the backquote expression. Since all type-checking is performed statically, it is possible to compose backquote expressions to create a function at run time with multiple (possibly incompatible) return types. This deficiency in the type system is a design choice: such errors are rare in practice, and checking for them would involve more overhead at dynamic compile time or additional linguistic extensions.

The code generated by ‘ may include implicit casts used to reconcile the result type of ‘ with its use; the standard conversion rules of ANSI C apply.

Some simple uses of cspec follow:

```
int cspec expr1 = ‘4;           /* Code specification for expression “4” */
float x;
float cspec expr2 = ‘(x + 4.);  /* Capture free variable x: its value will be
                                   bound when the dynamic code is executed */

/* All dynamic compound statements have type void cspec, regardless of
   whether the resulting code will return a value */
void cspec stmt = ‘{ printf(“hello, world\n”); return 0; };
```

2.2.2 vspec *Types.* vspec (*variable specification*) is the type of a dynamically generated lvalue, a variable whose storage class (whether it should reside in a register or on the stack, and in what location exactly) is determined dynamically. The evaluation type of the vspec is the type of the lvalue. void vspec is used as a generic vspec type. Objects of type vspec may be created by invoking the special forms param and local. param is used to create a parameter for the function currently under construction; local is used to reserve space in its activation record (or allocate a register if possible). See Section 2.4 for more details on these special forms.

In general, an object of type vspec is automatically treated as a variable of the vspec's evaluation type when it appears inside a cspec. A vspec inside a backquote expression can thus be used like a traditional C variable, both as an lvalue and an rvalue. For example, the following function creates a cspec that takes a single integer argument, adds one to it, and returns the result:

```
void cspec plus1(void) {
    /* Param takes the type and position of the argument to be generated */
    int vspec i = param(int, 0);
    return `{ return i + 1; };
}
```

vspecs allow us to construct functions that take a run-time-determined number of arguments; this functionality is necessary in applications such as the compiling interpreter described in Section 4.4.2.

2.2.3 *Discussion.* Within a quoted expression, vspecs and cspecs can be passed to functions that expect their evaluation types. The following code is legal:

```
int f(int j);
void main() {
    int vspec v;
    /* ... initialize v to a dynamic lvalue using local or param ... */
    void cspec c1 = `{ f(v); };
}
```

Within the quoted expression, f expects an integer. Since this function call is evaluated during the execution of the dynamic code, the integer lvalue to which v refers will already have been created. As a result, v can be passed to f like any other integer.

2.3 Unquoting Operators

The ` operator allows a programmer to create code at run time. In this section we describe two operators, @ and $, that are used within backquote expressions. @ is used to compose cspecs dynamically. $ is used to instantiate values as run-time constants in dynamic code. These two operators "unquote" their operands: their operands are evaluated at environment binding time.

2.3.1 *The @ Operator.* The @ operator allows code specifications to be composed into larger specifications. @ can only be applied inside a backquote expression, and its operand must be a cspec or vspec. @ "dereferences" its operand at environment binding time: it returns an object whose type is the evaluation type of @'s operand. The returned object is incorporated into the cspec in which the @ occurs. For example, in the following fragment, c is the additive composition of two cspecs:

```
/* Compose c1 and c2. Evaluation of c yields "9". */
```

```
int cspec c1 = '4, cspec c2 = '5;
int cspec c = '(@c1 + @c2);
```

Statements can be composed through concatenation:

```
/* Concatenate two null statements. */
void cspec s1 = '{}, cspec s2 = '{};
void cspec s = '{ @s1; @s2; };
```

Applying @ inside a backquote expression to a function (which must return a cspec or a vspec) causes the function to be called at environment binding time. Its result is incorporated into the backquote expression.

In order to improve the readability of code composition, 'C provides some implicit coercions of vspecs and cspecs, so that the @ operator may be omitted in several situations. An expression of type vspec or cspec that appears inside a quoted expression is coerced (with an implicit @) to an object of its corresponding evaluation type under the following conditions:

(1) the expression is not inside an unquoted expression.
(2) the expression is not being used as a statement.

The first restriction also includes implicitly unquoted expressions: that is, expressions that occur within an implicitly coerced expression are not implicitly coerced. For example, the arguments in a call to a function returning type cspec or vspec are not coerced, because the function call itself already is.

These coercions do not limit the expressiveness of 'C, because 'C supports only one "level" of dynamic code: it does not support dynamic code that generates more dynamic code. Therefore, the ability to manipulate vspecs and cspecs in dynamic code is not useful.

These implicit coercions simplify the syntax of cspec composition. Consider the following example:

```
int cspec a = '4; int cspec b = '5;
int cspec sum = '(a+b);
int cspec sumofsum = '(sum+sum);
```

This code is equivalent to the following code, due to the implicit coercion of a, b, and sum.

```
int cspec a = '4; int cspec b = '5;
int cspec sum = '(@a+@b);
int cspec sumofsum = '(@sum+@sum);
```

Compiling sumofsum results in dynamic code equivalent to (4+5)+(4+5).

Statements (and compound statements) are considered to have type void; an object of type void cspec inside a backquote expression cannot be used inside an expression, but can be composed as an expression statement:

```
void cspec mkscale(int **m, int n, int s) {
    return `{
        int i, j;
        for (i = 0; i < $n; i++) { /* Loop can be dynamically unrolled */
            int *v = ($m)[i];
            for (j = 0; j < $n; j++)
                v[j] = v[j] * $s; /* Multiplication can be strength−reduced */
        } };
}
```

Fig. 1. ‘C code to specialize multiplication of a matrix by an integer.

```
void cspec hello = `{ printf("hello "); };
void cspec world = `{ printf("world\n"); };
void cspec greeting = `{ @hello; @world; };
```

2.3.2 *The* $ *Operator.* The $ operator allows run-time values to be incorporated as *run-time constants* in dynamic code. $ evaluates its operand at environment binding time; the resulting value is used as a run-time constant in the containing cspec. $ may only appear inside a backquote expression, and it may not be unquoted. It may be applied to any object not of type cspec or vspec. The use of $ is illustrated in the code fragment below.

```
int x = 1;
void cspec c = `{ printf("$x = %d, x = %d\n", $x, x); };
x = 14;
(*compile(c, void))(); /* Compile and run: will print "$x = 1, x = 14". */
```

Use of $ enables specialization of code based on run-time constants. An example of this is the program in Figure 1, which specializes multiplication of a matrix by an integer. The pointer to the matrix, the size of the matrix, and the scale factor are all run-time constants, which enables optimizations such as dynamic loop unrolling and strength reduction of multiplication.

2.3.3 *Discussion.* Within an unquoted expression, vspecs and cspecs cannot be passed to functions that expect their evaluation types. The following code is illegal:

```
void cspec f(int j);
int g(int j);
void main() {
    int vspec v;
    void cspec c1 = `{ @f(v); }; /* error: v is the wrong type */
    void cspec c2 = `{ $g(v); }; /* error: v is the wrong type */
}
```

The storage class of a variable declared within the scope of dynamic code is determined dynamically. Therefore, a variable of type T that is local to a backquote

Category	Name	Synopsis
Management of dynamic code	compile	T (*compile(void cspec *code*, T))()
	free_code	void free_code(T)
Dynamic variables	local	T vspec local(T)
Dynamic function arguments	param	T vspec param(T, int *param-num*)
	push_init	void cspec push_init(void)
	push	void push(void cspec *args*, T cspec *next-arg*)
Dynamic control flow	label	void cspec label()
	jump	void cspec jump(void cspec *target*)
	self	T self(T, *other-args...*)

Table I. The ‘C special forms. T denotes a type.

expression must be treated as type T vspec when used in an unquoted expression. The example below illustrates this behavior.

```
void cspec f(int vspec j); /* f must take an int vspec ... */
void main() {
    void cspec c = ‘{ int v; @f(v); }; /* because v is a dynamic local */
}
```

2.4 Special Forms

‘C extends ANSI C with several special forms. Most of these special forms take types as arguments, and their result types sometimes depend on their input types. The special forms can be broken into four categories, as shown in Table I.

2.4.1 *Management of Dynamic Code.* The compile and free_code special forms are used to create executable code from code specifications and to deallocate the storage associated with a dynamic function, respectively. compile generates machine code from *code*, and returns a pointer to a function returning type T. It also automatically reclaims the storage for all existing vspecs and cspecs: as described in Section 3, vspecs and cspecs are objects that track necessary pieces of program state at environment binding and dynamic code generation time, so they are no longer needed and can be reclaimed after the dynamic code has been generated. Some other dynamic compilation systems [Leone and Lee 1996; Consel and Noel 1996] memoize dynamic code fragments: in the case of ‘C, the low overhead required to create code specifications, their generally small size, and their susceptibility to changes in the run-time environment make memoization of cspecs and vspecs unattractive.

free_code takes as argument a function pointer to a dynamic function previously created by compile, and reclaims the memory for that function. In this way, it is possible to reclaim the memory consumed by dynamic code when the code is no longer necessary. The programmer can use compile and free_code to explicitly manage memory used for dynamic code, similarly to how one normally uses malloc and free.

```
/* Construct cspec to sum n integer arguments. */
void cspec construct_sum(int n) {
    int i, cspec c = `0;
    for (i = 0; i < n; i++) {
        int vspec v = param(int, i); /* Create a parameter */
        c = `(c + v); /* Add param 'v' to current sum */
    }
    return `{ return c; };
}
int cspec construct_call(int nargs, int *arg_vec) {
    int (*sum)() = compile(construct_sum(5), int);
    void cspec args = push_init();     /* Initialize argument list */
    int i;
    for (i = 0; i < nargs; i++)        /* For each arg in arg_vec... */
        push(args, `$arg_vec[i]);      /* push it onto the args stack */
    return `sum(args);
}
```

Fig. 2. `C allows programmers to construct functions with dynamic numbers of arguments. construct_sum creates a function that takes n arguments and adds them. construct_call creates a cspec that invokes a dynamic function: it initializes an argument stack by invoking push_init, and dynamically adds arguments to this list by calling push. `C allows an argument list (an object of type void cspec) to be used as a single argument in a call: `sum(args) calls sum using the argument list denoted by args.

2.4.2 *Dynamic Variables.* The local special form is a mechanism for creating local variables in dynamic code. The objects it creates are analogous to local variables declared in the body of a backquote expression, but they can be used *across* backquote expressions, rather than being restricted to the scope of one expression. In addition, local enables dynamic code to have an arbitrary number of local variables. local returns an object of type T vspec that denotes a dynamic local variable of type T in the current dynamic function. In `C, the type T may include one of two C storage class specifiers, auto and register: the former indicates that the variable should be allocated on the stack, while the latter is a hint to the compiler that the variable should be placed in a register, if possible.

2.4.3 *Dynamic Function Arguments.* The param special form is used to create parameters of dynamic functions. param returns an object of type T vspec that denotes a formal parameter of the current dynamic function. *param-num* is the parameter's position in the function's parameter list, whereas T denotes its evaluation type. As illustrated in Figure 2, param can be used to create a function that has the number of its parameters determined at run time. Figure 3 shows how param can be used to curry functions.

Whereas param serves to create the formal parameters of a dynamic function, push_init and push are used together to dynamically build argument lists for function calls. push_init returns a cspec that corresponds to a new (initially empty) dynamic argument list. push adds the code specification for the next argument, *next-arg*, to the dynamically generated list of arguments *args*. (T, the evaluation type of *next-arg*, may not be void.) These two special forms allow the programmer to create function calls which pass a dynamically determined number of arguments to the

```
typedef int (*write_ptr)(char *, int);
/* Create a function that calls "write" with "tcb" hardwired as its first argument. */
write_ptr mkwrite(struct tcb *tcb) {
    char * vspec msg = param(char *, 0);
    int vspec nbytes = param(int, 1);
    return compile(`{ return write($tcb, msg, nbytes); }, int);
}
```

Fig. 3. 'C can be used to curry functions by creating function parameters dynamically. In this example, this functionality allows a network connection control block to be hidden from clients, but still enables operations on the connection (write, in this case) to be parameterized with per-connection data.

invoked function. Figure 2 illustrates their use.

2.4.4 *Dynamic Control Flow.* For error-checking purposes, 'C forbids goto statements from transfering control outside the enclosing backquote expression. Two special forms, label and jump, are used for inter-cspec control flow: jump returns the cspec of a jump to its argument, *target*. *Target* may be any object of type void cspec. label simply returns a void cspec that may be used as the destination of a jump. Syntactic sugar allows jump(target) to be written as jump target. Section 4.1.5 presents example 'C code that uses label and jump to implement specialized finite-state machines.

Lastly, self allows recursive calls in dynamic code without incurring the overhead of dereferencing a function pointer. T denotes the return type of the function that is being dynamically generated. Invoking self results in a call to the function that contains the invocation, with *other-args* passed as the arguments. self is just like any other function call, except that the return type of the dynamic function is unknown at environment binding time, so it must be provided as the first argument.

3. THE tcc COMPILER

The implementation of tcc was driven by two goals: high-quality dynamic code, and low dynamic compilation overhead. 'C allows the user to compose arbitrary pieces of code dynamically, which reduces the effectiveness of static analysis. As a result, many optimizations on dynamic code in 'C can only be performed at run time: improvements in code quality require more dynamic code generation time. The rest of this section discusses how tcc handles this tradeoff. Section 3.1 describes the structure of tcc, Section 3.2 gives an overview of the dynamic compilation process, and Section 3.3 discusses in detail some of the machinery that tcc uses to generate code at run time.

3.1 Architecture

The tcc compiler is based on lcc [Fraser and Hanson 1995; 1990], a portable compiler for ANSI C. lcc performs common subexpression elimination within extended basic blocks, and uses lburg [Fraser et al. 1992] to find the lowest-cost implementation of a given IR-level construct. Otherwise, it performs few optimizations.

Figure 4 illustrates the interaction of static and dynamic compilation in tcc. All parsing and semantic checking of dynamic expressions occurs at static compile time.

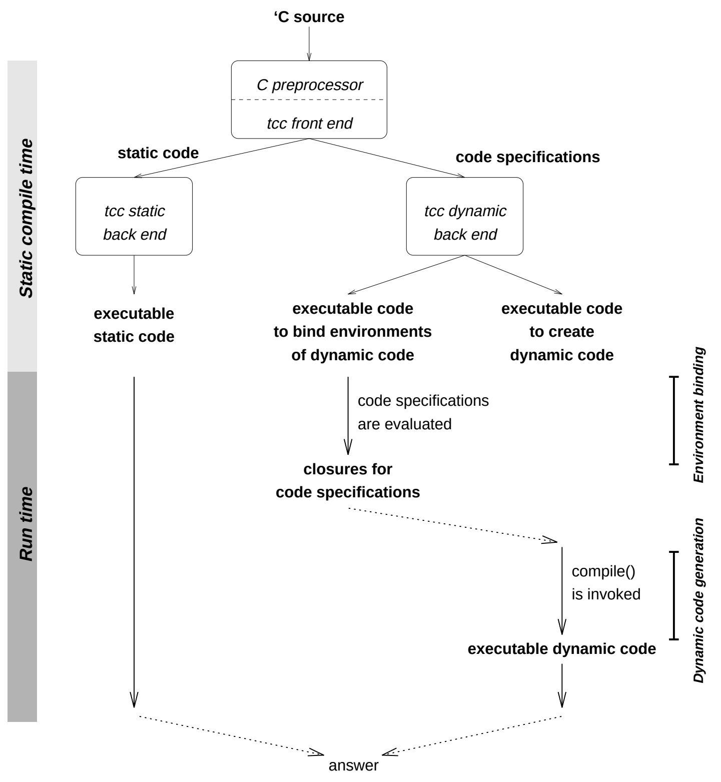

Fig. 4. Overview of the tcc compilation process.

Semantic checks are performed at the level of dynamically generated expressions. For each cspec, tcc performs internal type checking. It also tracks goto statements and labels to ensure that a goto does not transfer control outside the body of the containing cspec.

Unlike traditional static compilers, tcc uses two types of back ends to generate code. One is the static back end, which compiles the non-dynamic parts of ‘C programs, and emits either native assembly code or C code suitable for compilation by an optimizing compiler. The other, referred to as the dynamic back end, emits C code to *generate* dynamic code. Once produced by the dynamic back end, this C code is in turn compiled by the static back end.

tcc provides two dynamic code generation runtime systems so as to trade off code generation speed for dynamic code quality. The first of these runtime systems is VCODE [Engler 1996]. VCODE provides an interface resembling that of an idealized load/store RISC architecture; each instruction in this interface is a C macro which emits the corresponding instruction (or series of instructions) for the target architecture. VCODE's key feature is that it generates code with low run-time overhead: as few as ten instructions per generated instruction in the best case. While VCODE generates code quickly, it only has access to local information about backquote expressions: the quality of its code could often be improved. The second runtime system, ICODE, makes a different tradeoff, and produces better code at the expense of additional dynamic compilation overhead. Rather than emit code in one pass, it builds and optimizes an intermediate representation prior to code generation.

lcc is not an optimizing compiler. The assembly code emitted by its traditional static back ends is usually significantly slower (even three or more times slower) than that emitted by optimizing compilers such as gcc or vendor C compilers. To improve the quality of static code emitted by tcc, we have implemented a static back end that generates ANSI C from ‘C source; this code can then be compiled by any optimizing compiler. lcc’s traditional back ends can thus be used when static compilation must be fast (i.e., during development), and the C back end can be used when the performance of the code is critical.

3.2 The Dynamic Compilation Process

As described in Section 2, the creation of dynamic code can be divided into three phases: static compilation, environment binding, and dynamic code generation. This section describes how tcc implements these three phases.

3.2.1 *Static Compile Time.* During static compilation, tcc compiles the static parts of a ‘C program just like a traditional C compiler. It compiles each dynamic part – each backquote expression – to a *code-generating function* (CGF), which is invoked at run time to generate code for dynamic expressions.

In order to minimize the overhead of dynamic compilation, tcc performs as much instruction selection as possible statically. When using VCODE, both instruction selection based on operand types and cspec-local register allocation are done statically. Additionally, the intermediate representation of each backquote expression is processed by the common subexpression elimination and other local optimizations performed by the lcc front end. tcc also uses copt [Fraser 1980] to perform static peephole optimizations on the code-generating macros used by CGFs.

Not all register allocation and instruction selection can occur statically when using VCODE. For instance, it is not possible to determine statically what vspecs or cspecs will be incorporated into other cspecs when the program is executed. Hence, allocation of dynamic lvalues (vspecs) and of results of composed cspecs must be performed dynamically. The same is true of variables or temporaries that live across references to other cspecs. Each read or write to one of these dynamically determined lvalues is enclosed in a conditional in the CGF: different code is emitted at run time, depending on whether the object is dynamically allocated to a register or to memory. Since the process of instruction selection is encoded in the body of the code-generating function, it is inexpensive.

When using ICODE, tcc does not precompute as much information about dynamic code generation. Rather than emitting code directly, the ICODE macros first build up a simple intermediate representation; the ICODE runtime system then analyzes this representation to allocate registers and perform other optimizations before emitting code.

State for dynamic code generation is maintained in CGFs and in dynamically allocated closures. Closures are data structures that store five kinds of necessary information about the run-time environment of a backquote expression: (1) a function pointer to the corresponding statically generated CGF; (2) information about inter-cspec control flow (i.e., whether the backquote expression is the destination

```
cspec_t i = ((closure0 = (closure0_t)alloc_closure(4)),
             (closure0→cgf = cgf0), /* code gen func */
             (cspec_t)closure0);

cspec_t c = ((closure1 = (closure1_t)alloc_closure(16)),
             (closure1→cgf = cgf1), /* code gen func */
             (closure1→cs_i = i), /* nested cspec */
             (closure1→rc_j = j), /* run−time const */
             (closure1→fv_k = &k),/* free variable */
             (cspec_t)closure1);
```

Fig. 5. Sample closure assignments.

of a jump), (3) the values of run-time constants bound via the $ operator; (4) the addresses of free variables; (5) pointers to the run-time representations of the cspecs and vspecs used inside the backquote expression. Closures are necessary to reason about composition and out-of-order specification of dynamic code.

For each backquote expression, tcc statically generates both its code-generating function and the code to allocate and initialize closures. A new closure is initialized each time a backquote expression is evaluated. Cspecs are represented by pointers to closures.

For example, consider the following code:

```
int j, k;
int cspec i = ‘5;
void cspec c = ‘{ return i+$j*k; };
```

tcc implements the assignments to these cspecs by assignments to pointers to closures, as illustrated in Figure 5. i's closure contains only a pointer to its code-generating function. c has more dependencies on its environment, so its closure also stores other information.

Simplified code-generating functions for these cspecs appear in Figure 6. cgf0 allocates a temporary storage location, generates code to store the value 5 into it, and returns the location. cgf1 must do a little more work: the code that it generates loads the value stored at the address of free variable k into a register, multiplies it by the value of the run-time constant j, adds this to the dynamic value of i, and returns the result. Since i is a cspec, the code for "the dynamic value of i" is generated by calling i's code generating function.

3.2.2 *Run Time.* At run time, the code that initializes closures and the code-generating functions run to create dynamic code. As illustrated in Figure 4, this process consists of two parts: environment binding and dynamic code generation.

3.2.2.1 *Environment Binding.* During environment binding, code such as that in Figure 5 builds a closure that captures the environment of the corresponding backquote expression. Closures are heap-allocated, but their allocation cost is greatly reduced (down to a pointer increment, in the normal case) by using arenas [Forsythe 1977].

```
unsigned int cgf0 (closure0_t c) {
  vspec_t itmp0 = tc_local (INT); /* int temporary */
  seti (itmp0, 5); /* set it to 5 */
  return itmp0; /* return the location */
}

void cgf1 (closure1_t *c) {
  vspec_t itmp0 = tc_local (INT); /* some temporaries */
  vspec_t itmp1 = tc_local (INT);
  ldii (itmp1,zero,c→fv_k); /* addr of k */
  mulii (itmp1,itmp1,c→rc_j); /* run−time const j */
  /* now apply i’s CGF to i’s closure: cspec composition! */
  itmp0 = (*c→cs_i→cgf)(c→cs_i);
  addi (itmp1,itmp0,itmp1);
  reti (itmp1); /* emit a return (not return a value) */
}
```

Fig. 6. Sample code generating functions.

3.2.2.2 *Dynamic Code Generation.* During dynamic code generation, the ‘C run-time processes the code-generating functions. The CGFs use the information in the closures to generate code, and they perform various dynamic optimizations.

Dynamic code generation begins when the compile special form is invoked on a cspec. Compile calls the code-generating function for the cspec on the cspec’s closure, and the CGF performs most of the actual code generation. In terms of our running example, the code `int (*f)() = compile(j, int);` causes the run-time system to invoke closure1→cgf(closure1).

When the CGF returns, compile links the resulting code, resets the information regarding dynamically generated locals and parameters, and returns a pointer to the generated code. We attempt to minimize poor cache behavior by laying out the code in memory at a random offset modulo the i-cache size. It would be possible to track the placement of different dynamic functions to improve cache performance, but we do not do so currently.

Cspec composition — the inlining of code corresponding to one cspec, b, into that corresponding to another cspec, a, as described in Section 2.3.1 — occurs during dynamic code generation. This composition is implemented simply by invoking b’s CGF from within a’s CGF. If b returns a value, the value’s location is returned by its CGF, and can then be used by operations within a’s CGF.

The special forms for inter-cspec control flow, jump and label, are implemented efficiently. Each closure, including that of the empty void cspec ‘{}, contains a field that marks whether the corresponding cspec is the destination of a jump. The code-generating function checks this field, and if necessary, invokes a VCODE or ICODE macro to generate a label, which is eventually resolved when the runtime system links the code. As a result, label can simply return an empty cspec. jump marks the closure of the destination cspec appropriately, and then returns a closure that contains a pointer to the destination cspec and to a CGF that contains an ICODE or VCODE unconditional branch macro.

Generating efficient code from composed cspecs requires optimization analogous to function inlining and inter-procedural optimization. Performing some optimizations on the dynamic code after the order of composition of cspecs has been determined can significantly improve code quality. tcc's ICODE runtime system builds up an intermediate representation and performs some analyses before it generates executable code. The VCODE runtime system, by contrast, optimizes for code generation speed: it generates code in just one pass, but can make poor spill decisions when there is register pressure.

Some dynamic optimizations performed by tcc do not depend on the runtime system employed, but are encoded directly in the code-generating functions. These optimizations do not require global analysis or other expensive computation, and they can considerably improve the quality of dynamic code.

First, tcc does constant folding on run-time constants. The code-generating functions contain code to evaluate any parts of an expression that consist of static and run-time constants. The dynamically emitted instructions can then encode these values as immediates. Similarly, tcc performs simple local strength reduction based on run-time knowledge. For example, the code-generating functions can replace multiplication by a run-time constant integer with a series of shifts and adds, as described in [Briggs and Harvey 1994].

In addition, the code-generating functions automatically perform some dynamic loop unrolling and dead code elimination based on run-time constants. If the test of a loop or conditional is invariant at run time, or if a loop is bounded by run-time constants, then control flow can be determined at dynamic code generation time. In addition, run-time constant information propagates down loop nesting levels: for example, if a loop induction variable is bounded by run-time constants, and it is in turn used to bound a nested loop, then the induction variable of the nested loop is considered run-time constant too, within each unrolled iteration of the nested loop.

This style of optimization, which is hard-coded at static compile time and performed dynamically, produces good code without high dynamic compilation overhead. The code transformations are encoded in the CGF, and do not depend on run-time data structures. Furthermore, dynamic code that becomes unreachable at run time does not need to be generated, which can lead to faster code generation.

3.3 Runtime Systems

tcc provides two runtime systems for generating code. VCODE emits code locally, with no global analysis or optimization. ICODE builds up an intermediate representation in order to support more optimizations: in particular, better register allocation.

These two runtime systems allow programmers to choose the appropriate level of run-time optimization. The choice is application-specific: it depends on the number of times the code will be used and on the code's size and structure. Programmers can select which runtime system to use when they compile a 'C program.

3.3.1 VCODE.

When code generation speed is more important, the user can have tcc generate CGFs that use VCODE macros, which emit code in one pass. Register allocation with VCODE is fast and simple. VCODE provides getreg and putreg operations: the former allocates a machine register, the latter frees it. If there are no unallocated registers when getreg is invoked, it returns a spilled location designated by a negative number; VCODE macros recognize this number as a stack offset, and emit the necessary loads and stores. Clients that find these per-instruction if statements too expensive can disable them: getreg is then guaranteed to return only physical register names and, if it cannot satisfy a request, it terminates the program with a run-time error. This methodology is quite workable in situations where register usage is not data-dependent, and the improvement in code generation speed (roughly a factor of two) can make it worthwhile.

tcc statically emits getreg and putreg operations together with other VCODE macros in the code-generating functions: this ensures that the register assignments of one cspec do not conflict with those of another cspec dynamically composed with it. However, efficient inter-cspec register allocation is hard, and the placement of these register management operations can greatly affect code quality. For example, if a register is reserved (getreg'd) across a cspec composition point, it becomes unavailable for allocation in the nested cspec and in all cspecs nested within it. As a result, VCODE could run out of registers and resort to spills after only a few levels of cspec nesting. To help improve code quality, tcc follows some simple heuristics. First, expression trees are rearranged so that cspec operands of instructions are evaluated before non-cspec operands. This minimizes the number of temporaries that span cspec references, and hence the number of registers allocated by the CGF of one cspec during the execution of the code-generating function of a nested cspec. Secondly, no registers are allocated for the return value of non-void cspecs: the code-generating function for a cspec allocates the register for storing its result, and simply returns this register name to the CGF of the enclosing cspec.

To further reduce the overhead of VCODE register allocation, tcc reserves a limited number of physical registers. These registers are not allocated by getreg, but instead are managed at static compile time by tcc's dynamic back end. They can only be used for values whose live ranges do not span composition with a cspec and are typically employed for expression temporaries.

As a result of these optimizations, VCODE register allocation is quite fast. However, if the dynamic code contains large basic blocks with high register pressure, or if cspecs are dynamically combined in a way that forces many spills, code quality suffers.

3.3.2 ICODE.

When code quality is more important, the user can have tcc generate CGFs that use ICODE macros, which generate an intermediate representation on which optimizations can be performed. For example, ICODE can perform global register allocation on dynamic code more effectively than VCODE in the presence of cspec composition.

ICODE provides an interface similar to that of VCODE, with two main extensions: (1) an infinite number of registers, and (2) primitives to express changes in estimated usage frequency of code. The first extension allows ICODE clients to emit code that assumes no spills, leaving the work of global, inter-cspec register allocation to ICODE. The second allows ICODE to obtain estimates of code execution frequency at low cost. For instance, prior to invoking ICODE macros that correspond to a loop body, the ICODE client could invoke refmul(10): this tells ICODE that all variable references occurring in the subsequent macros should be weighted as occurring 10 times (an estimated average number of loop iterations) more than the surrounding code. After emitting the loop body, the ICODE client should invoke a corresponding refdiv(10) macro to correctly weight code outside of the loop. The estimates obtained in this way are useful for several optimizations; they currently provide approximate variable usage counts that help to guide register allocation.

ICODE's intermediate representation is designed to be compact (two 4-byte machine words per ICODE instruction) and easy to parse in order to reduce the overhead of subsequent passes. When compile is invoked in ICODE mode, ICODE builds a flow graph, performs register allocation, and finally generates executable code. We have attempted to minimize the cost of each of these operations. We briefly discuss each of them in turn.

3.3.2.1 *Flow Graph Construction.* ICODE builds a control-flow graph in one pass after all CGFs have been invoked. The flow graph is a single array that uses pointers for indexing. In order to allocate all required memory in a single allocation, ICODE computes an upper bound on the number of basic blocks by summing the numbers of labels and jumps emitted by ICODE macros. After allocating space for an an array of this size, it traverses the buffer of ICODE instructions and adds basic blocks to the array in the same order in which they exist in the list of instructions. Forward references are initially stored in an array of pairs of basic block addresses; when all the basic blocks are built, the forward references are resolved by traversing this array and linking the pairs of blocks listed in it. As it builds the flow graph, ICODE also collects a minimal amount of local data-flow information (def and use sets for each basic block). All memory management occurs through arenas [Forsythe 1977], which ensures low amortized cost for memory allocation and essentially free deallocation.

3.3.2.2 *Register Allocation.* Good register allocation is the main benefit that ICODE provides over VCODE. ICODE currently implements four different register allocation algorithms, which differ in overhead and in the quality of the code that they produce: graph coloring, linear scan, and a simple scheme based on estimated usage counts.

The graph coloring allocator implements a simplified version of Chaitin's algorithm [Chaitin et al. 1981]: it does not do coalescing, but employs estimates of usage counts to guide spilling. The live variable information used by this allocator is obtained by an iterative data-flow pass over the ICODE flow graph. Both the liveness analysis and the register allocation pass were carefully implemented for speed, but their actual performance is inherently limited because the algorithms were developed for static compilers, and prioritize code quality over compilation speed. The graph coloring allocator therefore serves as a benchmark: it produces the best code, but is relatively slow.

At the opposite end of the spectrum is ICODE's simple "usage count" allocator: it makes no attempt to produce particularly good code, but is fast. This allocator ignores liveness information altogether: it simply sorts all variables in order of decreasing estimated usage counts, allocates the n available registers to the n variables with the highest usage counts, and places all other variables on the stack. Most dynamic code created using 'C is relatively small: as a result, despite its simplicity, this allocation algorithm oftern performs just as well as graph coloring.

Lastly, ICODE implements linear scan register allocation [Poletto and Sarkar 1998]. This algorithm improves performance relative to graph coloring: it does not build

```
LinearScanRegisterAllocation
    $active \leftarrow \{\}$
    foreach live interval $i$, in order of increasing start point
        ExpireOldIntervals($i$)
        if length($active$) $= R$ then
            SpillAtInterval($i$)
        else
            $register[i] \leftarrow$ a register removed from pool of free registers
            add $i$ to $active$, sorted by increasing end point

ExpireOldIntervals($i$)
    foreach interval $j$ in $active$, in order of increasing end point
        if $endpoint[j] \geq startpoint[i]$ then
            return
        remove $j$ from $active$
        add $register[j]$ to pool of free registers

SpillAtInterval($i$)
    $spill \leftarrow$ last interval in $active$
    if $endpoint[spill] > endpoint[i]$ then
        $register[i] \leftarrow register[spill]$
        $location[spill] \leftarrow$ new stack location
        remove $spill$ from $active$
        add $i$ to $active$, sorted by increasing end point
    else
        $location[i] \leftarrow$ new stack location
```

Fig. 7. Linear scan register allocation.

and color a graph, but rather assigns registers to variables in one pass over a sorted list of *live intervals*. Given an ordering (for example, linear layout order, or depth-first order) of the instructions in a flow graph, a live interval of a variable v is the interval $[m, n]$ such that v is not live prior to instruction m or after instruction n. Once the list of live intervals is computed, allocating R available registers so as to minimize the number of spilled intervals requires removing the smallest number of live intervals so that no more than R live intervals overlap. The algorithm skips forward through the sorted list of live intervals from start point to start point, keeping track of the set of overlapping intervals. When more than R intervals overlap, it heuristically spills the interval that ends furthest away, and moves on to the next start point. The algorithm appears in Figure 7, and is discussed in detail in [Poletto and Sarkar 1998].

ICODE can obtain live interval information in two different ways. The first method is simply to compute live variable information by iterative analysis, as for graph coloring, and to then coarsen this information to one live interval per variable. This technique produces live intervals that are as accurate as possible, but is not fast. The second method is considerably faster, but produces slightly more conservative intervals. The algorithm finds and topologically sorts the strongly connected components (SCCs) of the flow graph. If a variable is defined or used in an SCC, it is assumed live throughout the whole SCC. The live interval of a variable therefore stretches from the (topologically) first SCC where it appears to the last. Like the linear scan algorithm, this technique is analyzed in [Poletto and Sarkar 1998].

These different algorithms allow ICODE to provide a variety of tradeoffs of compile-time overhead versus quality of code. Graph coloring is most expensive and usually produces the best code, linear scan is considerably faster but sometimes produces worse code, and the usage count allocator is faster than linear scan but can produce considerably worse code. However, given the relatively small size of most ‘C dynamic code, the algorithms perform similarly on the benchmarks presented in this paper. As a result, Section 5 presents measurements only for a representative case, the linear scan allocator using live intervals derived from full live variable information.

3.3.2.3 *Code Generation.* The final phase of code generation with ICODE is the translation of the intermediate representation into executable code. The code emitter makes one pass through the ICODE intermediate representation: it invokes the VCODE macro that corresponds to each ICODE instruction, and prepends and appends spill code as necessary.

ICODE has several hundred instructions (the Cartesian product of operation kinds and operand types), so a translator for the entire instruction set is quite large. Most ‘C programs, however, use only a small subset of all ICODE instructions. tcc therefore keeps track of the ICODE instructions used by an application. It encodes this usage information for a given ‘C source file in dummy symbol names in the corresponding object file. A pre-linking pass then scans all the files about to be linked and emits an additional object file containing an ICODE-to-binary translator tailored specifically to the ICODE macros present in the executable. This simple trick significantly reduces the size of the ICODE code generator; for example, for the benchmarks presented in this paper it usually shrank the code generators by a factor of 5 or 6.

3.3.2.4 *Other Features.* ICODE is designed to be a generic framework for dynamic code optimization: it is possible to extend it with additional optimization passes, such as copy propagation, common subexpression elimination, etc. However, preliminary measurements indicate that much dynamic optimization beyond register allocation is probably not practical: the increase in dynamic compile time is not justified by sufficient improvements in the speed of the resulting code.

4. APPLICATIONS

‘C is valuable in a number of practical settings. The language can be employed to increase performance through the use of dynamic code generation, as well as to simplify the creation of programs that cannot easily be written in ANSI C. For example, ‘C can be used to build efficient searching and sorting routines, implement dynamic integrated layer processing for high performance network subsystems, and create compiling interpreters and “just-in-time” compilers. This section presents several ways in which ‘C and dynamic code generation can help to solve practical problems. We have divided the examples into four broad categories: specialization, dynamic function call construction, dynamic inlining, and compilation. Many of the applications described below are also used for the performance evaluation in Section 5.

```
struct hte {            /* Hash table entry structure */
    int val;            /* Key that entry is associated with */
    struct hte *next;   /* Pointer to next entry */
    /* ... */
};

struct ht {             /* Hash table structure */
    int scatter;        /* Value used to scatter keys */
    int norm;           /* Value used to normalize */
    struct hte **hte;   /* Vector of pointers to hash table entries */
};

/* Hash returns a pointer to the hash table entry, if any, that matches val. */
struct hte *hash(struct ht *ht, int val) {
    struct hte *hte = ht→hte[(val * ht→scatter) / ht→norm];
    while (hte && hte→val != val) hte = hte→next;
    return hte;
}
```

Fig. 8. A hash function written in C.

4.1 Specialization

'C provides programmers with a general set of mechanisms to build code at run time. Dynamic code generation can be used to hard-wire run-time values into the instruction stream, which can enable code optimizations such as strength reduction and dead code elimination. In addition, 'C enables more unusual and complicated operations, such as specializing a piece of code to a particular input data structure (for example, a given array) or some class of data structures (for example, all arrays with elements of a given length).

4.1.1 *Hashing.* A simple example of 'C is the optimization of a generic hash function, where the table size is determined at run time, and where the function uses a run-time value to help its hash. Consider the C code in Figure 8. The C function has three values that can be treated as run-time constants: ht→hte, ht→scatter, and ht→norm. As illustrated in Figure 9, using 'C to specialize the function for these values requires only a few changes. The resulting code can be considerably faster than the equivalent C version, because tcc hard-codes the run-time constants hte, scatter, and norm in the instruction stream, and reduces the multiplication and division operations in strength. The cost of using the resulting dynamic function is an indirect jump on a function pointer.

4.1.2 *Vector Dot Product.* Matrix and vector manipulations such as dot product provide many opportunities for dynamic code generation. They often involve a large number of operations on values which change relatively infrequently. Matrices may have run-time characteristics (i.e., large numbers of zeros and small integers) that can improve performance of matrix operations, but cannot be exploited by static compilation techniques. In addition, sparse matrix techniques are only efficient for matrices with a high degree of sparseness.

In the context of matrix multiplication, dynamic code generation can remove multiplication by zeros, and strength-reduce multiplication by small integers. Be-

```
/* Type of the function generated by mk_hash: takes a value as input
   and produces a (possibly null) pointer to a hash table entry */
typedef struct hte *(*hptr)(int val);

/* Construct a hash function with the size, scatter, and hash table pointer hard-coded. */
hptr mk_hash(struct ht *ht) {
    int vspec val = param(int, 0);
    void cspec code = `{
        struct hte *hte = ($ht->hte)[(val * $ht->scatter) / $ht->norm];
        while (hte && hte->val != val) hte = hte->next;
        return hte;
    };
    return compile(code, struct hte *); /* Compile and return the result */
}
```

Fig. 9. Specialized hash function written in ‘C.

```
void dot(int *a, int *b, int n) {
    int sum, k;
    for (sum = k = 0; k < n; k++) sum += a[k]*b[k];
    return sum;
}
```

Fig. 10. A dot-product routine written in C.

cause code for each row is created once and then used once for each column, the costs of code generation can be recovered easily. Consider the C code to compute dot product in Figure 10. At run time several optimizations can be employed. For example, the programmer can directly eliminate multiplication by zero. The corresponding ‘C code appears in Figure 11.

The dot product written in ‘C can perform substantially better than its static C counterpart. The ‘C code does not emit code for multiplications by zero. In addition, the ‘C compiler can encode values as immediates in arithmetic instructions, and can reduce multiplications by the runtime constant `$row[k]` in strength.

4.1.3 *Binary Search.* Figure 12 illustrates a recursive implementation of binary search. We present a recursive version here for clarity: the static code used for measurements in Section 5 is a more efficient, iterative version. Nonetheless, even the iterative implementation incurs some overhead due to looping and because it needs to reference into the input array.

When the input array will be searched several times, however, one can use ‘C to write code like that in Figure 13. The structure of this code is very similar to that of the recursive binary search. By adding a few dynamic code generation primitives to the original algorithm, we have created a function that returns a `cspec` for binary search that is tailored to a given input set. We can create a C function pointer from this `cspec`:

```
typedef int (*ip)(int key);
ip mksearch(int n, int *x) {
```

```
void cspec mkdot(int row[], int n) {
    int k;
    int *vspec col = param(int *, 0); /* Input vector for dynamic function */
    int cspec sum = `0; /* Spec for sum of products; initally 0 */
    for (k = 0; k < n; k++) /* Only generate code for non-zero multiplications */
        if (row[k]) /* Specialize on index of col[k] and value of row[k] */
            sum = `(sum + col[$k] * $row[k]);
    return `{ return sum; };
}
```

Fig. 11. 'C code to build a specialized dot-product routine.

```
int bin(int *x, int key, int l, int u, int r) {
    int p;
    if (l > u) return −1;
    p = u − r;
    if (x[p] == key) return p;
    else if (x[p] < key) return bin(x, key, p+1, u, r/2);
    else return bin(x, key, l, p−1, r/2);
}
```

Fig. 12. A tail-recursive implementation of binary search.

```
void cspec gen(int *x, int vspec key, int l, int u, int r) {
    int p;
    if (l > u) return `{ return −1; };
    p = u − r;
    return ` {
        if ($(x[p]) == key) return $p;
        else if ($(x[p]) < key) @gen(x, key, p+1, u, r/2);
        else @gen(x, key, l, p−1, r/2);
    };
}
```

Fig. 13. 'C code to create a "self-searching" executable array.

```
typedef double (*dptr)();
dptr mkpow(int exp) {
    double vspec base = param(double, 0); /* Argument: the base */
    double vspec result = local(register double); /* Local: running product */
    void cspec squares;
    int bit = 2;

    /* Initialize the running product */
    if (1&exp) squares = `{ result=base; };
    else squares = `{ result=1.; };

    /* Multiply some more, if necessary */
    while (bit ≤ exp) {
        squares = `{ @squares; base *= base; };
        if (bit & exp) squares = `{ @squares; result *= base; };
        bit = bit ≪ 1;
    }
    /* Compile a function which returns the result */
    return compile(`{ @squares; return result; }, double);
}
```

Fig. 14. Code to create a specialized exponentiation function.

```
    int vspec key = param(int, 0); /* One argument: the key to search for */
    return (ip)compile(gen(x, key, 0, n−1, n/2), int);
}
```

In the resulting code, the values from the input array are hard-wired into the instruction stream, and the loop is unrolled into a binary tree of nested if statements that compare the value to be found to constants. As a result, the search involves neither loads from memory nor looping overhead, so the dynamically constructed code is considerably more efficient than its static counterpart. For small input vectors (on the order of 30 elements), this results in lookup performance superior even to that of a hash table.

4.1.4 *Exponentiation.* Another example of tailoring code to an input set comes from computer graphics [Draves 1995], where it is sometimes necessary to apply an exponentiation function to a large data set. Traditionally, exponentiation is computed in a loop which performs repeated multiplication and squaring. Given a fixed exponent, we can unroll this loop and obtain straight-line code that contains the minimum number of multiplications. The ‘C code to perform this optimization appears in Figure 14.

4.1.5 *Finite State Machines.* It is possible to use ‘C to specialize code for more complex data than just arrays and primitive values. For example, tcc can compile a DFA description into specialized code, as shown in Figure 15. The function mk_dfa accepts a data structure that describes a DFA with a unique start state and some number of accept states: at each state, the DFA transitions to the next state and produces one character of output based on the next character in its input. mk_dfa uses ‘C’s inter-cspec control flow primitives, jump and label (Section 2.4),

```
typedef struct {
    int n;                  /* State number (start state has n=0) */
    int acceptp;            /* Non−zero if this is an accept state */
    char *in;               /* I/O and next state info: on input in[k] */
    char *out;              /*   produce output out[k] and go to state */
    int *next;              /*   number next[k] */
} *state_t;
typedef struct {
    int size;               /* Number of states */
    state_t *states;        /* Description of each state */
} *dfa_t;

int (*mk_dfa(dfa_t dfa))(char *in, char *out) {
    char * vspec in = param(char *, 0); /* Input to dfa */
    char * vspec out = param(char *, 1); /* Output buffer */
    char vspec t = local(char);
    void cspec *labels = (void cspec *)malloc(dfa→size * sizeof(void cspec));
    void cspec code = `{};     /* Initially dynamic code is empty */
    int i;
    for (i = 0; i < dfa→n_states; i++)
        labels[i] = label();  /* Create labels to mark each state */
    for (i = 0; i < dfa→n_states; i++) { /* For each state ... */
        state_t cur = dfa→states[i];
        int j = 0;
        code = `{ @code;      /* ... prepend the code so far */
                  @labels[i]; /* ... add the label to mark this state */
                  t = *in; }; /* ... read current input */
        while (cur→in[j]) {   /* ... add code to do the right thing if */
            code = `{ @code;  /*      this is an input we expect */
                      if (t == $cur→in[j]) {
                          in++; *out++ = $cur→out[j];
                          jump labels[cur→next[j]];
                      } };
            j++;
        }
        code = `{ @code;      /* ... add code to return 0 if we're at end */
                  if (t == 0) { /*   of input in an accept state, or */
                      if ($cur→acceptp) return 0;
                      else return −2; /* −2 if we're in another state */
                  } /* or −1 if no transition and not end of input */
                  else return −1; };
    }
    return compile(code, int);
}
```

Fig. 15. Code to create a hard-coded finite state machine.

to create code that directly implements the given DFA: each state is implemented by a separate piece of dynamic code, and state transitions are simply conditional branches. The dynamic code contains no references into the original data structure that describes the DFA.

4.1.6 *Swap.* The examples so far have illustrated specialization to specific values. Value specialization can improve performance, but it may be impractical if

```
typedef void (*fp)(void *, void *);
fp mk_swap(int size) {
    long * vspec src = param(long *, 0); /* Arg 0: source */
    long * vspec dst = param(long *, 1); /* Arg 1: destination */
    long vspec tmp = local(long); /* Temporary for swaps */
    void cspec s = `{}; /* Code to be built up, initially empty */
    int i;

    for (i = 0; i < size/sizeof(long); i++) /* Build swap code */
        s = `{ @s; tmp = src[$i]; src[$i] = dst[$i]; dst[$i] = tmp; };
    return (fp)compile(s, void);
}
```

Fig. 16. ‘C code to generate a specialized swap routine.

the values change too frequently. A related approach that does not suffer from this drawback is specialization based on properties, such as size, of data types. For instance, it is often necessary to swap the contents of two regions of memory: in-place sorting algorithms are one such example. As long as the data being manipulated is no larger than a machine word, this process is quite efficient. However, when manipulating larger regions (for example, C structs), the code is often inefficient. One way to copy the regions is to invoke the C library memory copy routine, memcpy, repeatedly. Using memcpy incurs function call overhead, as well overhead within memcpy itself. Another way is to iteratively swap one word at a time, but this method incurs loop overhead.

‘C allows us to create a swap routine that is specialized to the size of the region being swapped. The code in Figure 16 is an example, simplified to handle only the case where the size of the region is a multiple of sizeof(long). This routine returns a pointer to a function that contains only assignments, and swaps the region of the given size without resorting to looping or multiple calls to memcpy. The size of the generated code will usually be rather small, which makes this a profitable optimization. Section 5 evaluates a ‘C heapsort implementation in which the swap routine is customized to the size of the objects being sorted and dynamically inlined into the main sorting code.

4.1.7 *Copy.* Copying a memory region of arbitrary size is another common operation. An important application of this is computer graphics [Pike et al. 1985]. Similar to the previous code for swapping regions of memory, the example code in Figure 17 returns a function customized for copying a memory region of a given size.

The procedure mk_copy takes two arguments: the size of the regions to be copied, and the number of times that the inner copying loop should be unrolled. It creates a cspec for a function that takes two arguments, pointers to source and destination regions. It then creates prologue code to copy regions which would not be copied by the unrolled loop (if $n \bmod \mathit{unrollx} \neq 0$), and generates the body of the unrolled loop. Finally, it composes these two cspecs, invokes compile, and returns a pointer to the resulting customized copy routine.

```
typedef void (*fp)(void *, void *);
fp mk_copy(int n, int unrollx) {
    int i, j;
    unsigned * vspec src = param(unsigned *, 0); /* Arg 0: source */
    unsigned * vspec dst = param(unsigned *, 1); /* Arg 1: destination */
    int vspec k = local(int); /* Local: loop counter */
    cspec void copy = `{}, unrollbody = `{}; /* Code to build, initially empty */

    for (i = 0; i < n % unrollx; i++) /* Unroll the remainder copy */
        copy = `{ @copy; dst[$i] = src[$i]; };

    if (n ≥ unrollx) /* Unroll copy loop unrollx times */
        for (j = 0; j < unrollx; j ++)
            unrollbody = `{ @unrollbody; dst[k+$j] = src[k+$j]; };

    copy = `{ /* Compose remainder copy with the main unrolled loop */
            @copy;
            for (k = $i; k < $n; k += $unrollx) @unrollbody;
    };
    /* Compile and return a function pointer */
    return (fp)compile(copy, void);
}
```

Fig. 17. Generating specialized copy code in ‘C.

4.2 Dynamic Function Call Construction

‘C allows programmers to generate functions (and calls to functions) that have arguments whose number and types are not known at compile time. This functionality distinguishes ‘C: neither ANSI C nor any of the dynamic compilation systems discussed in Section 6 provides mechanisms for constructing function calls dynamically.

A useful application of dynamic function call construction is the generation of code to marshal and unmarshal arguments stored in a byte vector. These operations are frequently performed to support remote procedure call [Birrell and Nelson 1984]. Generating specialized code for the most active functions results in substantial performance benefits [Thekkath and Levy 1993].

We present two functions, marshal and unmarshal, that dynamically construct marshaling and unmarshaling code, respectively, given a “format vector,” types, that specifies the types of arguments. The sample code in Figure 18 generates a marshaling function for arguments with a particular set of types (in this example, int, void *, and double). First, it specifies code to allocate storage for a byte vector large enough to hold the arguments described by the type format vector. Then, for every type in the type vector, it creates a vspec that refers to the corresponding parameter, and constructs code to store the parameter's value into the byte vector at a distinct run-time constant offset. Finally, it specifies code that returns a pointer to the byte vector of marshaled arguments. After dynamic code generation, the function that has been constructed will store all of its parameters at fixed, non-overlapping offsets in the result vector. Since all type and offset computations

```
typedef union { int i; double d; void *p; } type;
typedef enum { INTEGER, DOUBLE, POINTER } type_t; /* Types we expect to marshal */
extern void *alloc();
void cspec mk_marshal(type_t *types, int nargs) {
    int i;
    type *vspec m = local(type *); /* Spec of pointer to result vector */
    void cspec s = `{ m = (type *)alloc(nargs * sizeof(type)); };

    for (i = 0; i < nargs; i++) { /* Add code to marshal each param */
        switch(types[i]) {
        case INTEGER: s = `{ @s; m[$i].i = param($i, int); };        break;
        case DOUBLE:  s = `{ @s; m[$i].d = param($i, double); };  break;
        case POINTER: s = `{ @s; m[$i].p = param($i, void *); };  break;
        }
    }
    /* Return code spec to marshal parameters and return result vector */
    return `{ @s; return m; };
}
```

Fig. 18. Sample marshaling code in ‘C.

```
typedef int (*fptr)(); /* Type of the function we will be calling */
void cspec mk_unmarshal(type_t *types, int nargs) {
    int i;
    fptr vspec fp = param(fptr, 0); /* Arg 0: the function to invoke */
    type *vspec m = param(type *, 1); /* Arg 1: the vector to unmarshal */
    void cspec args = push_init(); /* Initialize the dynamic argument list */

    for (i = 0; i < nargs; i++) { /* Build up the dynamic argument list */
        switch(types[i]) {
        case INTEGER: push(args, `m[$i].i); break;
        case DOUBLE:  push(args, `m[$i].d); break;
        case POINTER: push(args, `m[$i].p); break;
        }
    }
    /* Return code spec to call the given function with unmarshaled args */
    return `{ fp(args); };
}
```

Fig. 19. Unmarshaling code in ‘C.

have been done during environment binding, the generated code will be efficient. Further performance gains could be achieved if the code were to manage details such as endianness and alignment.

Dynamic generation of unmarshaling code is equally useful. The process relies on ‘C's mechanism for constructing calls to arbitrary functions at run time. It not only improves efficiency, but also provides valuable functionality. For example, in Tcl [Ousterhout 1994] the runtime system can make upcalls into an application. However, because Tcl cannot dynamically create code to call an arbitrary function, it marshals all of the upcall arguments into a single byte vector, and forces applications to explicitly unmarshal them. If systems such as Tcl used ‘C to construct upcalls, clients would be able to write their code as normal C routines, which would increase the ease of expression and decrease the chance for errors.

The code in Figure 19 generates an unmarshaling function that works with the marshaling code in Figure 18. The generated code takes a function pointer as its first argument and a byte vector of marshaled arguments as its second. It unmarshals the values in the byte vector into their appropriate parameter positions, and then invokes the function pointer. mk_unmarshal works as follows: it creates the specifications for the generated function's two incoming arguments, and initializes the argument list. Then, for every type in the type vector, it creates a cspec to index into the byte vector at a fixed offset and pushes this cspec into its correct parameter position. Finally, it creates the call to the function pointed to by the first dynamic argument.

4.3 Dynamic Inlining

'C makes it easy to inline and compose functions dynamically. This feature is analogous to dynamic inlining through indirect function calls. It improves performance by eliminating function call overhead and by creating the opportunity for optimization across function boundaries.

4.3.1 *Parameterized Library Functions.* Dynamic function composition is useful when writing and using library functions that would normally be parameterized with function pointers, such as many mathematical and standard C library routines. The 'C code for Newton's method [Press et al. 1992] in Figure 20 illustrates its use. The function newton takes as arguments the maximum allowed number of iterations, a tolerance, an initial estimate, and two pointers to functions that return cspecs to evaluate a function and its derivative. In the calls f(p0) and fprime(p0), p0 is passed as a vspec argument. The cspecs returned by these functions are incorporated directly into the dynamically generated code. As a result, there is no function call overhead, and inter-cspec optimization can occur during dynamic code generation.

4.3.2 *Network Protocol Layers.* Another important application of dynamic inlining is the optimization of networking code. The modular composition of different protocol layers has long been a goal in the networking community [Clark and Tennenhouse 1990]. Each protocol layer frequently involves data manipulation operations, such as checksumming and byte-swapping. Since performing multiple data manipulation passes is expensive, it is desirable to compose the layers so that all the data handling occurs in one phase [Clark and Tennenhouse 1990].

'C can be used to construct a network subsystem that dynamically integrates protocol data operations into a single pass over memory (e.g., by incorporating encryption and compression into a single copy operation). A simple design for such a system divides each data-manipulation stage into *pipes* that each consume a single input and produce a single output. These pipes can then be composed and incorporated into a data-copying loop. The design includes the ability to specify prologue and epilogue code that is executed before and after the data-copying loop, respectively. As a result, pipes can manipulate state and make end-to-end checks, such as ensuring that a checksum is valid after it has been computed.

The pipe in Figure 21 can be used to do byte-swapping. Since a byte swapper

```
typedef double cspec (*dptr)(double vspec);

/* Dynamically create a Newton−Raphson routine. n: max number of iterations;
   tol: maximum tolerance; p0: initial estimate; f: function to solve; fprime: derivative of f. */
double newton(int n, double tol, double usr_p0, dptr f, dptr fprime) {
    void cspec cs = `{
        int i; double p, p0 = usr_p0;
        for (i = 0; i < $n; i++) {
            p = p0 − f(p0) / fprime(p0); /* Compose cspecs returned by f and fprime */
            if (abs(p − p0) < tol) return p; /* Return result if we've converged enough */
            p0 = p; /* Seed the next iteration */
        }
        error("method failed after %d iterations\n", i);
    };
    return (*compile(cs,double))(); /* Compile, call, and return the result. */
}

/* Function that constructs a cspec to compute f(x) = (x+1)^2 */
double cspec f(double vspec x) { return `((x + 1.0) * (x + 1.0)); }
/* Function that constructs a cspec to calculate f'(x) = 2(x+1) */
double cspec fprime(double vspec x) { return `(2.0 * (x + 1.0)); }
/* Call newton to solve an equation */
void use_newton(void) { printf("Root is %f\n", newton(100, .000001, 10., f, fprime)); }
```

Fig. 20. ‘C code to create and use routines to use Newton's method for solving polynomials. This example computes the root of the function $f(x) = (x + 1)^2$.

```
unsigned cspec byteswap(unsigned vspec input) {
    return `((input << 24) | ((input & 0xff00) << 8) |
              ((input >> 8) & 0xff00) | ((input  >> 24) & 0xff));
}
/* "Byteswap" maintains no state and so needs no initial or final code */
void cspec byteswap_initial(void) { return `{}; }
void cspec byteswap_final(void) { return `{}; }
```

Fig. 21. A sample pipe: byteswap returns a cspec for code that byte-swaps input.

does not need to maintain any state, there is no need to specify initial and final code. The byte swapper simply consists of the "consumer" routine that manipulates the data.

To construct the integrated data-copying routine, the initial, consumer, and final cspecs of each pipe are composed with the corresponding cspecs of the pipe's neighbors. The composed initial code is placed at the beginning of the resulting routine; the consumer code is inserted in a loop, and composed with code which provides it with input and stores its output; and the final code is placed at the end of the routine. A simplified version would look like the code fragment in Figure 22. In a mature implementation of this code, we could further improve performance by unrolling the data-copying loop. Additionally, pipes would take inputs and outputs of different sizes that the composition function would reconcile.

```
typedef void cspec (*vptr)();
typedef unsigned cspec (*uptr)(unsigned cspec);
/* Pipe structure: contains pointers to functions that return cspecs
   for the initialization, pipe, and finalization code of each pipe */
struct pipe {
    vptr initial, final;        /* initial and final code */
    uptr pipe;                  /* pipe */
};

/* Return cspec that results from composing the given vector of pipes */
void cspec compose(struct pipe *plist, int n) {
    struct pipe *p;
    int vspec nwords = param(int, 0);                 /* Arg 0: input size   */
    unsigned * vspec input = param(unsigned *, 1);    /* Arg 1: pipe input   */
    unsigned * vspec output = param(unsigned *, 2);   /* Arg 2: pipe output  */
    void cspec initial = `{}, cspec final = `{};  /* Prologue and epilogue code */
    unsigned cspec pipes = `input[i];/* Base pipe input */

    for (p = &plist[0]; p < &plist[n]; p++) { /* Compose all stages together */
        initial = `{ @initial; @p→initial(); }; /* Compose initial statements */
        pipes =`p→pipe(pipes); /* Compose pipes: one pipe's output is the next one's input */
        final = `{ @final; @p→final(); }; /* Compose final statements */
    }
    /* Create a function with initial statements first, consumer statements
       second, and final statements last. */
    return `{ int i;
            @initial;
            for (i = 0; i < nwords; i++) output[i] = pipes;
            @final;
    };
}
```

Fig. 22. ‘C code for composing data pipes.

4.4 Compilation

‘C’s imperative approach to dynamic code generation makes it well-suited for writing compilers and compiling interpreters. ‘C helps to make such programs both efficient and easy to write: the programmer can focus on parsing, and leave the task of code generation to ‘C.

4.4.1 *Domain-Specific Languages.* Small, domain-specific languages can benefit from dynamic compilation. The small query languages used to search databases are one class of such languages [Keppel et al. 1993]. Since databases are large, dynamically compiled queries will usually be applied many times, which can easily pay for the cost of dynamic code generation.

We provide a toy example in Figure 23. The function mk_query takes a vector of queries. Each query contains the following elements: a database record field (i.e., CHILDREN or INCOME); a value to compare this field to; and the operation to use in the comparison (i.e., $<, >$, etc.). Given a query vector, mk_query dynamically creates a query function which takes a database record as an argument and checks whether that record satisfies all of the constraints in the query vector. This check

```
typedef enum { INCOME, CHILDREN /* ... */ } query; /* Query type */
typedef enum { LT, LE, GT, GE, NE, EQ } bool_op; /*  Comparison operation */

struct query {
    query record_field;          /* Field to use */
    unsigned val;                /* Value to compare to */
    bool_op bool_op;             /* Comparison operation */
};
struct record { int income; int children; /* ... */ }; /* Simple database record */

/* Function that takes a pointer to a database record and returns 0 or 1,
   depending on whether the record matches the query */
typedef int (*iptr)(struct record *r);

iptr mk_query(struct query *q, int n) {
    int i, cspec field, cspec expr = `1; /* Initialize the boolean expression */
    struct record * vspec r = param(struct record *, 0); /* Record to examine */

    for (i = 0; i < n; i++) { /* Build the rest of the boolean expression */
        switch (q[i].record_field) { /* Load the appropriate field value */
        case INCOME:     field = `(r→income); break;
        case CHILDREN: field = `(r→children); break;
        /* ... */
        }
        switch (q[i].bool_op) { /* Compare the field value to runtime constant q[i] */
        case LT: expr = `(expr && field < $q[i].val); break;
        case EQ: expr = `(expr && field == $q[i].val); break;
        case LE: expr = `(expr && field ≤ $q[i].val); break;
        /* ... */
        }
    }
    return (iptr)compile(`{ return expr; }, int);
}
```

Fig. 23. Compilation of a small query language.

```
enum { WHILE, IF, ELSE, ID, CONST, LE, GE, NE, EQ };  /* Multi−character tokens */
int expect();              /* Consume the given token from the input stream, or fail if not found */
int cspec expr();          /* Parse unary expressions */
int gettok();              /* Consume a token from the input stream */
int look();                /* Peek at the next token without consuming it */
int cur_tok;               /* Current token */
int vspec lookup_sym(); /* Given a token, return corresponding vspec */

void cspec stmt() {
    int cspec e = `0;  void cspec s = `{}, s1 = `{}, s2 = `{};
    switch (gettok()) {
    case WHILE: /* 'while' '(' expr ')' stmt */
        expect(`('); e = expr();
        expect(`)'); s = stmt();
        return `{ while(e) @s; };
    case IF: /* 'if' '(' expr ')' stmt { 'else' stmt } */
        expect(`('); e = expr();
        expect(`)'); s1 = stmt();
        if (look(ELSE)) {
            gettok(); s2 = stmt();
            return `{ if (e) @s1; else @s2; };
        } else return `{ if (e) @s1; };
    case `{': /* `{' stmt* `}' */
        while (!look(`}')) s = `{ @s; @stmt(); };
        return s;
    case `;': return `{};
    case ID: { /* ID `=' expr `;' */
        int vspec lvalue = lookup_sym(cur_tok);
        expect(`='); e = expr(); expect(`;');
        return `{ lvalue = e; };
    }
    default: parse_err("expecting statement");
    }
}
```

Fig. 24. A sample statement parser from a compiling interpreter written in 'C.

is implemented simply as an expression which computes the conjunction of the given constraints. The query function never references the query vector, since all the values and comparison operations in the vector have been hard-coded into the dynamic code's instruction stream.

The dynamically generated code expects one incoming argument, the database record to be compared. It then "seeds" the boolean expression: since we are building a conjunction, the initial value is 1. The loop then traverses the query vector, and builds up the dynamic code for the conjunction according to the fields, values, and comparison operations described in the vector. When the cspec for the boolean expression is constructed, mk_query compiles it and returns a function pointer. That optimized function can be applied to database entries to determine whether they match the given constraints.

4.4.2 *Compiling Interpreters.* Compiling interpreters (also known as JIT compilers) are important pieces of technology: they combine the flexibility of an interpreted programming environment with the performance of compiled code. For a given piece of code, the user of a compiling interpreter pays a one-time cost of compilation, which can be roughly comparable to that of interpretation. Every subsequent use of that code employs the compiled version, which can be much faster than the interpreted version. Compiling interpreters are also useful in systems such as Java, in which "just-in-time" compilers are commonly used.

Figure 24 contains a fragment of code for a simple compiling interpreter written in 'C. This interpreter translates a simple subset of C: as it parses the program, it builds up a cspec that represents it.

5. EVALUATION

'C and tcc support efficient dynamic code generation. In particular, the measurements in this section demonstrate the following results:

—By using 'C and tcc, we can achieve good speedups relative to static C. Speedups by a factor of two to four are common for the programs that we have described.

—'C and tcc do not impose excessive overhead on performance. The cost of dynamic compilation is usually recovered in under 100 runs of a benchmark; sometimes, this cost can be recovered in one run of a benchmark.

—The tradeoff between dynamic code quality and dynamic code generation speed must be made on a per-application basis. For some applications, it is better to generate code faster; for others, it is better to generate better code.

—Dynamic code generation can result in large speedups when it enables large-scale optimization: when interpretation can be eliminated, or when dynamic inlining enables further optimization. It provides smaller speedups if only local optimizations, such as strength reduction, are performed dynamically. In such cases, the cost of dynamic code generation may outweigh its benefits.

5.1 Experimental Methodology

The benchmarks that we measure have been described in previous sections. Table II briefly summarizes each benchmark, and lists the section in which it appears.

The performance improvements of dynamic code generation hinge on customizing code to data. As a result, the performance of all of the benchmarks in this section is data-dependent to some degree. In particular, the amount of code generated by the benchmarks is in some cases dependent on the input data. For example, since dp generates code to compute the dot product of an input vector with a run-time constant vector, the size of the dynamic code (and hence its i-cache performance) is dependent on the size of the run-time constant vector. Its performance relative to static code also depends on the density of 0s in the run-time constant vector, since those elements are optimized out when generating the dynamic code. Similarly, binary, and dfa generate more code for larger inputs, which generally improves their performance relative to equivalent static code until negative i-cache effects come into play.

Some other benchmarks – ntn, ilp, and query – involve dynamic function inlining that is affected by input data. For example, the code inlined in ntn depends on the

Benchmark	Description	Section	Page
ms	Scale a 100x100 matrix by the integers in $[10, 100]$	2.3.2	8
hash	Hash table, constant table size, scatter value, and hash table pointer: one hit and one miss	4.1.1	22
dp	Dot product with a run-time constant vector: length 40, one-third zeroes	4.1.2	23
binary	Binary search on a 16-element constant array; one hit and one miss	4.1.3	23
pow	Exponentiation of 2 by the integers in $[10, 40]$	4.1.4	24
dfa	Finite state machine computation 6 states, 13 transitions, input length 16	4.1.5	25
heap	Heapsort, parameterized with a specialized swap: 500-entry array of 12-byte structures	4.1.6	26
mshl	Marshal five arguments into a byte vector	4.2	28
unmshl	Unmarshal a byte vector, and call a function of five arguments	4.2	28
ntn	Root of $f(x) = (x+1)^2$ to a tolerance of 10^{-9}	4.3.1	30
ilp	Integrated copy, checksum, byteswap of a 16KB buffer	4.3.2	31
query	Query 2000 records with seven binary comparisons	4.4.1	32

Table II. Descriptions of benchmarks.

function to be computed, that in ilp on the nature of the protocol stack, and that in query on the type of query submitted by the user. The advantage of dynamic code over static code increases with the opportunity for inlining and cross-function optimization. For example, an ilp protocol stack composed from many small passes will perform relatively better in dynamic code that one composed from a few larger passes.

Lastly, a few benchmarks are relatively data-independent. pow, heap, mshl, and unmshl generate varying amounts of code depending, respectively, on the exponent used, or the type and size of the objects being sorted, marshaled or unmarshaled, but the differences are small for most reasonable inputs. ms obtains performance improvements by hard-wiring loop bounds and strength-reducing multiplication by the scale factor. hash makes similar optimizations when computing a hash function. The values of run-time constants may affect performance to some degree (for example, excessively large constants are not useful for this sort of optimization), but such effects are much smaller than those of more large-scale dynamic optimizations.

Each benchmark was written both in 'C and in static C. The 'C programs were compiled with both the VCODE and the ICODE-based tcc back ends. When measuring the performance of the ICODE runtime system, we always employed linear scan register allocation with live intervals derived from live variable information. The static C programs were compiled both with the lcc compiler and with the GNU C compiler, gcc. The code-generating functions used for dynamic code generation are created from the lcc intermediate representation, using that compiler's code generation strategies. As a result, the performance of lcc-generated code should be used as the baseline to measure the impact of dynamic code generation. Measurements collected using gcc serve to compare tcc to an optimizing compiler of reasonable quality.

The machine used for measurements is a Sun Ultra 2 Model 2170 workstation with

384MB of main memory and two 168 MHz UltraSPARC-I CPUs. The UltraSPARC-I can issue up to 2 integer and 2 floating point instructions per cycle, and has a write-through, non-allocating, direct-mapped, on-chip 16KB cache. It implements the SPARC version 9 architecture [SPARC International 1994]. tcc also generates code for the MIPS family of processors; we report only SPARC measurements for clarity, since results on the two architectures are similar.

Times were obtained by measuring a large number of trials — enough to provide several seconds of granularity, with negligible standard deviations — using the Unix getrusage system call. The number of trials varied from 100 to 100000, depending on the benchmark. The resulting times were then divided by the number of iterations to obtain the average overhead of a single run. This form of measurement ignores the effects of cache refill misses, but is representative of how these applications would likely be used (for example, in tight inner loops).

Section 5.2 discusses the performance effects of using dynamic code generation: specifically, the speedup of dynamic code relative to static code, and the overhead of dynamic code generation relative to speedup. Section 5.3 presents break downs of the dynamic compilation overhead of both VCODE and ICODE in units of processor cycles per generated instruction.

5.2 Performance

This section shows that tcc provides low-overhead dynamic code generation, and that it can be used to speed up a number of benchmarks. We describe results for the benchmarks in Table II and for xv, a freely available image manipulation package.

We compute the speedup due to dynamic code generation by dividing the time required to run the static code by the time required to run the corresponding dynamic code. We measure overhead by calculating each benchmark's "cross-over" point, if one exists. This point is the number of times that dynamic code must be used so that the overhead of dynamic code generation equals the time gained by running the dynamic code.

The performance of dynamic code is up to an order of magnitude better than that of unoptimized static code. In many cases, the performance improvement of using dynamic code generation can be amortized over fewer than ten runs of the dynamic code. The benchmarks that achieve the highest speedups are those in which dynamic information allows the most effective restructuring of code relative to the static version. The main classes of such benchmarks are numerical code in which particular values allow large amounts of work to be optimized away (for example, dp), code in which an expensive layer of data structure interpretation can be removed at run time (for example, query), and code in which inlining can be performed dynamically but not statically (for example, ilp).

VCODE generates code approximately three to eight times more quickly than ICODE. Nevertheless, the code generated by ICODE can be considerably faster than that generated by VCODE. A programmer can choose between the two systems to trade code quality for code generation speed, depending on the needs of the application.

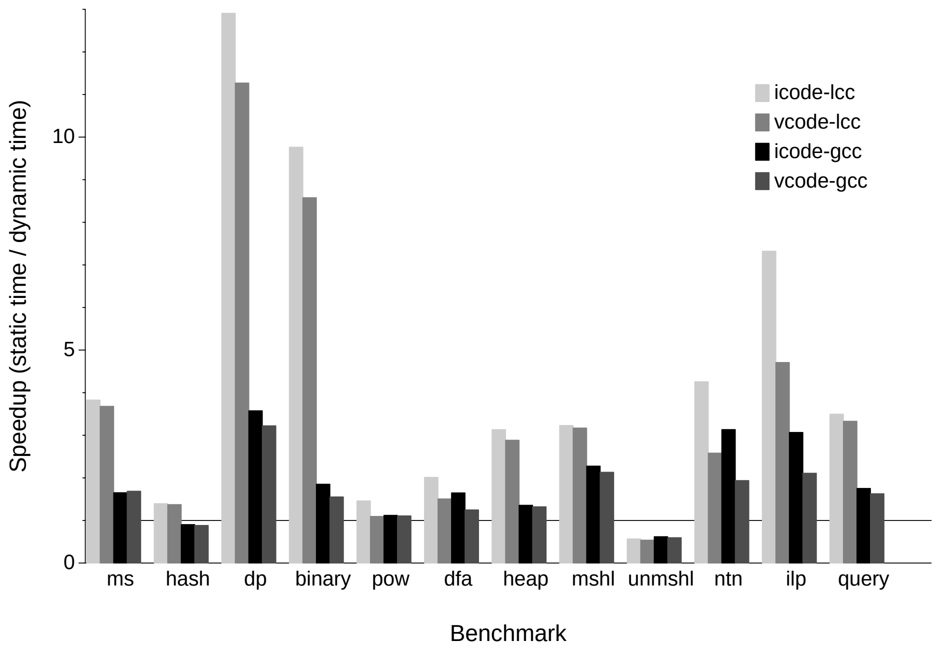

Fig. 25. Speedup of dynamic code over static code.

5.2.1 *Speedup.* Figure 25 shows that using 'C and tcc improves the performance of almost all of our benchmarks. Both in this figure and in Figure 26, the legend indicates which static and dynamic compilers are being compared. icode-lcc compares dynamic code created with ICODE to static code compiled with lcc; vcode-lcc compares dynamic code created with VCODE to static code compiled with lcc. Similarly, icode-gcc compares ICODE to static code compiled with gcc, and vcode-gcc compares VCODE to static code compiled with gcc.

In general, dynamic code is significantly faster than static code: speedups by a factor of two relative to the best code emitted by gcc are common. Unsurprisingly, the code produced by ICODE is faster than that produced by VCODE, by up to 50% in some cases. Also, the GNU compiler generates better code than lcc, so the speedups relative to gcc are almost always smaller than those relative to lcc. As mentioned earlier, however, the basis for comparison should be lcc, since the code-generating functions are generated by an lcc-style back end, which does not perform static optimizations.

Dynamic code generation does not pay off in only one benchmark, unmshl. In this benchmark, 'C provides functionality that does not exist in C. The static code used for comparison implements a special case of the general functionality provided by the 'C code, and it is very well tuned.

5.2.2 *Cross-over.* Figure 26 indicates that the cost of dynamic code generation in tcc is reasonably low. The cross-over point on the vertical axis is the number of times that the dynamic code must be used in order for the total overhead of its

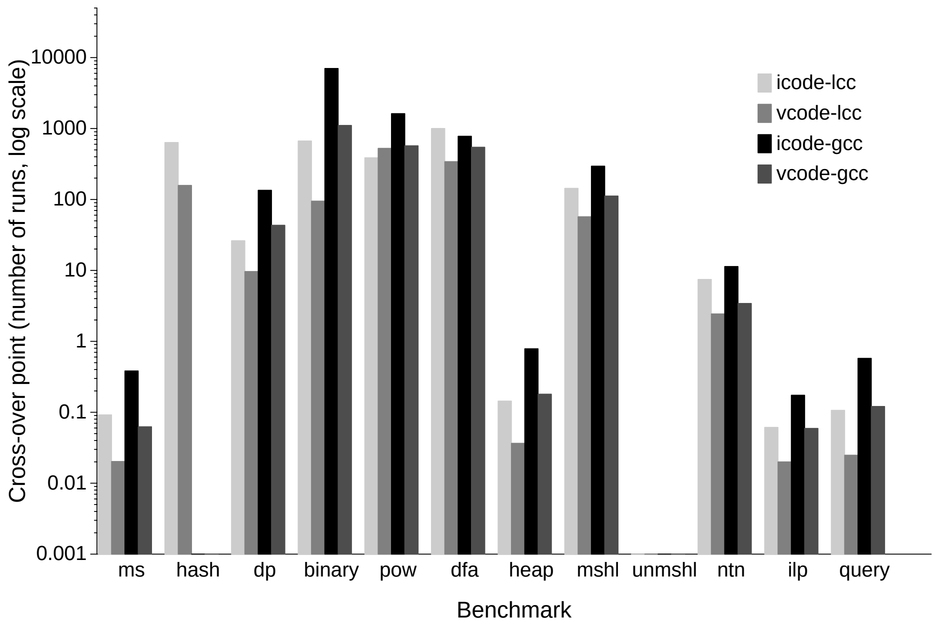

Fig. 26. Cross-over points, in number of runs. If the cross-over point does not exist, the bar is omitted.

compilation and uses to be equal to the overhead of the same number of uses of static code. This number is a measure of how quickly dynamic code "pays for itself." For all benchmarks except query, one use of dynamic code corresponds to one run of the dynamically created function. In query, however, the dynamic code is used as a small part of the overall algorithm: it is the test function used to determine whether a record in the database matches a particular query. As a result, in that case we define one use of the dynamic code to be one run of the search algorithm, which corresponds to many invocations (one per database entry) of the dynamic code. This methodology realistically measures how specialization is used in these cases.

In the case of unmshl, the dynamic code is slower than the static one, so the cross-over point never occurs. Usually, however, the performance benefit of dynamic code generation occurs after a few hundred or fewer runs. In some cases (ms, heap, ilp, and query), the dynamic code pays for itself after only one run of the benchmark. In ms and heap, this occurs because a reasonable problem size is large relative to the overhead of dynamic compilation, so even small improvements in run time (from strength reduction, loop unrolling, and hard-wiring pointers) outweigh the code generation overhead. In addition, ilp and query exemplify the types of applications in which dynamic code generation can be most useful: ilp benefits from extensive dynamic function inlining that cannot be performed statically, and query dynamically removes a layer of interpretation inherent in a database query language.

Figures 25 and 26 show how dynamic compilation speed can be exchanged for

Convolution mask (pixels)	**Times (seconds)**			
	lcc	gcc	tcc (ICODE)	DCG overhead
3×3	5.79	2.44	1.91	2.5×10^{-3}
7×7	17.57	6.86	5.78	3.5×10^{-3}

Table III. Performance of convolution on an 1152x900 image in xv.

dynamic code quality. VCODE can be used to perform fast, one-pass dynamic code generation when the dynamic code will not used very much. However, the code generated by ICODE is often considerably faster than that generated by VCODE: hence, ICODE is useful when the dynamic code is run more times, so that the code's performance is more important than the cost of generating it.

5.2.3 xv. To test the performance of tcc on a relatively large application, we modified xv to use 'C. xv is a popular image manipulation package that consists of approximately 60000 lines of code. We picked one of its image processing algorithms and changed it to make use of dynamic code generation. One algorithm is sufficient, since most of the algorithms are implemented similarly. The algorithm, *Blur*, applies a convolution matrix of user-defined size that consists of all 1's to the source image. The original algorithm was implemented efficiently: the values in the convolution matrix are known statically to be all 1's, so convolution at a point is simply the average of the image values of neighboring points. Nonetheless, the inner loop contains image-boundary checks based on run-time constants, and is bounded by a run-time constant, the size of the convolution matrix.

Results from this experiment appear in Table III. For both a 3×3 and 7×7 convolution mask, the dynamic code obtained using tcc and ICODE is approximately 3 times as fast as the static code created by lcc, and approximately 20% faster than the static code generated by gcc with all optimizations turned on. Importantly, the overhead of dynamic code generation is almost 3 orders of magnitude less than the performance benefit it provides.

xv is an example of the usefulness of dynamic code generation in the context of a well-known application program. Two factors make this result significant. First, the original static code was quite well-tuned. In addition, the tcc code generator that emits the code-generating code is derived from an lcc code generator: as a result, the default dynamic code, barring any dynamic optimizations, is considerably less well-tuned than equivalent code generated by the GNU compiler. Despite all this, the dynamic code is faster than even aggressively optimized static code, and the cost of dynamic code generation is insignificant compared to the benefit obtained.

5.3 Analysis

This section analyzes the code generation overhead of VCODE and ICODE. VCODE generates code at a cost of approximately 100 cycles per generated instruction. Most of this time is taken up by register management; just laying out the instructions in memory requires much less overhead. ICODE generates code roughly three to eight times more slowly than VCODE. Again, much of this overhead is due to register allocation: the choice of register allocator can significantly influence ICODE performance.

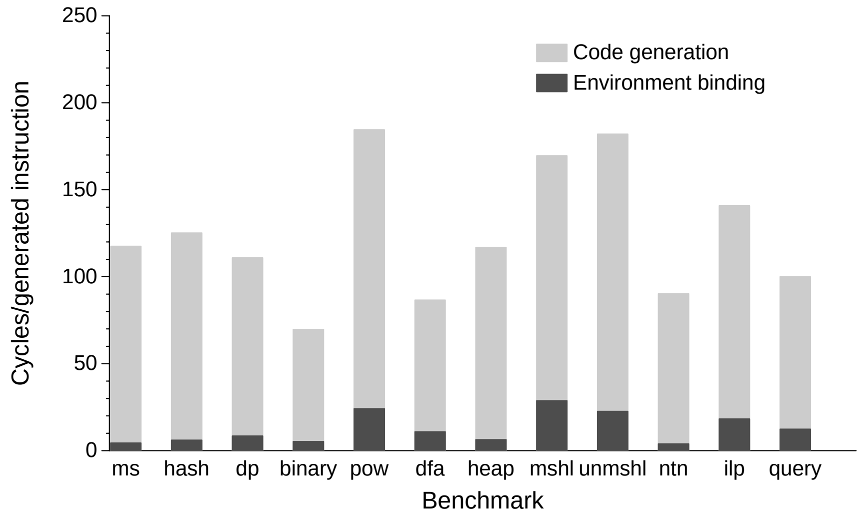

Fig. 27. Dynamic code generation overhead using VCODE.

5.3.1 VCODE *Overhead.* Figure 27 breaks down the code generation overhead of VCODE for each of the benchmarks. The VCODE back end generates code at approximately 100 cycles per generated instruction: the geometric mean of the overheads for the benchmarks in this paper is 119 cycles per instruction. The cost of environment binding is small — almost all the time is spent in code generation.

The code generation overhead has several components. The breakdown is difficult to measure precisely and varies slightly from benchmark to benchmark, but there are some broad patterns:

—Laying instructions out in memory (bitwise operations to construct instructions, and stores to write them to memory) accounts for roughly 15% of the overhead.

—Dynamically allocating memory for the code, linking, delay slot optimizations, and prologue and epilogue code add approximately another 25%.

—Register management (VCODE's putreg/getreg operations) accounts for about 50% of the overhead.

—Approximately 10% of the overhead is due to other artifacts, such as checks on the storage class of dynamic variables, the overhead of calling code generating functions, etc.

These results indicate that dynamic register allocation, even in the minimal VCODE implementation, is a major source of overhead. This cost is unavoidable in 'C's dynamic code composition model; systems that can statically allocate registers for dynamic code should therefore have a considerable advantage over 'C in terms of dynamic compile-time performance.

5.3.2 ICODE *Overhead.* Figure 28 breaks down the code generation overhead of ICODE for each of the benchmarks. For each benchmark we report two costs. The columns labeled L represent the overhead of using ICODE with linear scan register

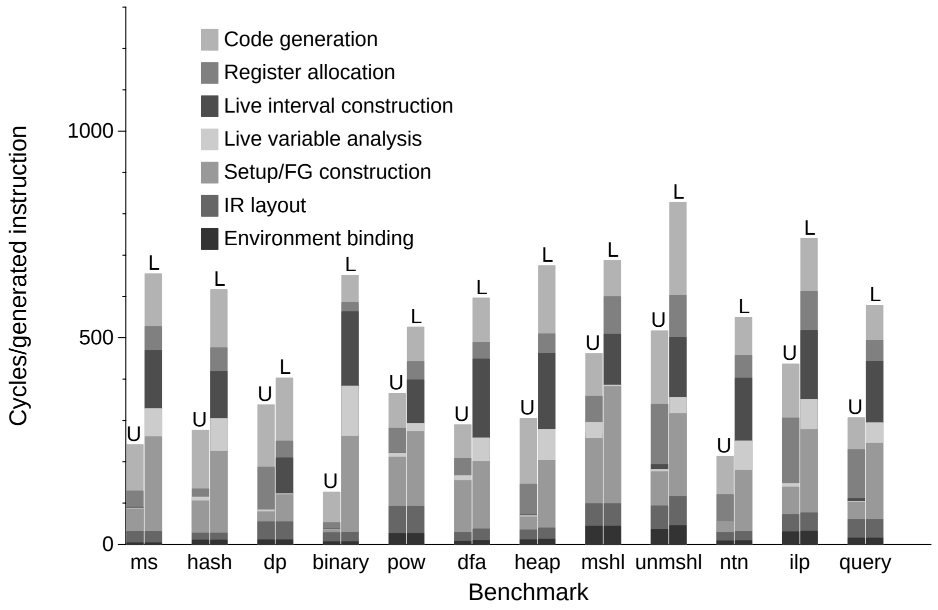

Fig. 28. Dynamic code generation overhead using ICODE. Columns labeled L denote ICODE with linear scan register allocation; those labeled U denote ICODE with a simple allocator based on usage counts.

allocation based on precise live variable information. The columns labeled U represent the overhead of using ICODE with the simple allocator that places the variables with the highest usage counts in registers. Both algorithms are described in Section 3.3.2.2. For each benchmark and type of register allocation, we report the overhead due to environment binding, laying out the ICODE IR, creating the flow graph and doing some setup (allocating memory for the code, initializing VCODE, etc.), performing various phases of register allocation, and finally generating code.

ICODE's code generation speed ranges from about 200 to 800 cycles per instruction, depending on the benchmark and the type of register allocation. The geometric mean of the overheads for the benchmarks in this paper, when using linear scan register allocation, is 615 cycles per instruction.

The allocator based on usage counts is considerably faster than linear scan, because it does not have to compute live variables and live intervals, and does not need to build a complete flow graph. By contrast, the traditional graph coloring register allocator (not shown in the figure) is generally over twice as slow as the linear scan allocator. Graph coloring is a useful reference algorithm, but it is not practical for dynamic code generation: linear scan is faster and produces code that is usually just as good. At the other extreme, the usage count allocator is faster than linear scan and often makes good allocation decisions on small benchmarks; however, it sometimes produces very poor code (for example, dfa and heap). As a result, linear scan is the default allocator for ICODE, and the one for which we show performance results in Figures 25 and 26.

In addition to register allocation and related liveness analyses, the main sources of overhead are flow graph construction and code generation. The latter roughly corresponds to the VCODE code generation overhead. Environment binding and laying out the ICODE intermediate representation are relatively inexpensive operations.

6. RELATED WORK

Dynamic code generation has a long history [Keppel et al. 1991]. It has been used to increase the performance of operating systems [Bershad et al. 1995; Engler et al. 1995; Pu et al. 1995; Pu et al. 1988], windowing operations [Pike et al. 1985], dynamically typed languages [Chambers and Ungar 1989; Deutsch and Schiffman 1984; Hölzle and Ungar 1994], and simulators [Witchel and Rosenblum 1996; Veenstra and Fowler 1994]. Research on 'C and tcc grew out of work on DCG [Engler and Proebsting 1994], a low-level dynamic code generation system. Earlier descriptions of the 'C language and tcc have been published elsewhere [Engler et al. 1995; Poletto et al. 1997].

Other languages also provide the ability to create code at run time. For example, most LISP dialects [Kelsey et al. 1998; Steele Jr. 1990], Tcl [Ousterhout 1994], and Perl [Wall et al. 1996], provide an "eval" operation that allows code to be generated dynamically. This approach is extremely flexible but, unfortunately, comes at a high price: since these languages are dynamically typed, little code generation cost can be pushed to compile time.

Keppel addressed some of the issues in retargeting dynamic code generation [Keppel 1991]. He developed a portable system for modifying instruction spaces on a variety of machines. His system dealt with the difficulties presented by caches and operating system restrictions, but it did not address how to select and emit actual binary instructions. Keppel, Eggers, and Henry [Keppel et al. 1993] demonstrated that dynamic code generation can be effective for several different applications.

There has been much recent work on specialization and run-time compilation in C. Unlike 'C, which takes an imperative approach to expressing dynamic code generation, and requires the programmer to explicitly manipulate dynamic code objects, most of these systems adopt a declarative approach. In this model, the programmer annotates the C source code with directives that identify run-time constants and (possibly) specify various code generation policies, such as the aggressiveness of specialization and the extent to which dynamic code is cached and reused. Dynamic code generation happens automatically in such systems.

One such system has been developed at the University of Washington [Auslander et al. 1996; Grant et al. 1997]. The first UW compiler [Auslander et al. 1996] provided a limited set of annotations and exhibited relatively poor performance. That system performs data-flow analysis to discover all derived run-time constants, given the run-time constants specified by the programmer. The second system, DyC [Grant et al. 1997], provides a more expressive annotation language and support for several features, including polyvariant division (which allows the same program point to be analyzed for different combinations of run-time invariants), polyvariant specialization (which allows the same program point to be dynamically compiled multiple times, each specialized to different values of a set of run-time invariants), lazy specialization, and interprocedural specialization. These features allow the system to achieve levels of functionality similar to 'C, but in a completely different style of programming. DyC does not provide mechanisms for creating functions and function calls with dynamically determined numbers of arguments. For simple forms of specialization, DyC sometimes generates code more quickly than tcc using VCODE. For more complex forms of specialization (such as generating a compiling interpreter from an interpreter), DyC is approximately as fast as tcc using ICODE.

Another automatic dynamic code generation system driven by user annotations is Tempo [Consel and Noel 1996]. Tempo is a template-based dynamic compiler derived from GNU CC. It is similar to DyC, but provides support for only function-level polyvariant division and specialization, and does not provide means of setting policies for division, specialization, caching, and speculative specialization. In addition, it does not support specialization across separate source files. Unlike DyC, however, it performs conservative alias and side-effect analysis to identify partially static data structures. The performance data indicates that Tempo's cross-over points tend to be slightly worse than DyC, but the speedups are comparable, which indicates that Tempo generates code of comparable quality, but more slowly. Since Tempo does not support complex specialization mechanisms, though, its expressiveness is weaker than that of DyC and 'C. The Tempo project has targeted 'C as a back end for its run-time specializer.

Fabius [Leone and Lee 1996] is a dynamic compilation system based on partial evaluation that was developed in the context of a purely functional subset of ML. It uses a syntactic form of currying to allow the programmer to express run-time invariants. Given the hints regarding run-time invariants, Fabius performs dynamic compilation and optimization automatically. Fabius achieves fast code generation speeds, but 'C is more flexible than Fabius. In Fabius, the user cannot directly manipulate dynamic code, and unlike in Tempo and DyC, the user has no recourse to additional annotations for controlling the code generation process. In essence, Fabius uses dynamic compilation solely for its performance advantages, extending to run time the applicability of traditional optimizations such as copy propagation and dead code elimination.

The Dynamo project [Leone and Dybvig 1997] is a successor to Fabius. Leone and Dybvig are designing a staged compiler architecture that supports different levels of dynamic optimization: emitting a high-level intermediate representation enables "heavyweight" optimizations to be performed at run time, whereas emitting a low-level intermediate representation enables only "lightweight" optimizations. The eventual goal of the Dynamo project is to build a system that will automatically perform dynamic optimization.

From a linguistic perspective, the declarative systems have the advantage that most annotations preserve the semantics of the original code, so it is possible to compile and debug a program without them. Knowing exactly where to insert the annotations, however, can still be a challenge. Also, importantly, only DyC seems to provide dynamic code generation flexibility comparable to that of 'C. Furthermore, even with DyC, many common dynamic code programming tasks, such as the various lightweight compiling interpreters presented in this paper, involve writing interpreter functions no less complicated than those one would write for 'C. In the end, the choice of system is probably a matter of individual taste.

From a performance perspective, declarative systems can often allow better static optimization than 'C, because the control flow within dynamic code can be determined statically. Nonetheless, complicated control flow, such as loops containing conditionals, can limit this advantage. For example, in DyC, the full extent of dynamic code cannot in general be determined statically unless one performs full multi-way loop unrolling, which can cause prohibitive code growth. Finally, only Leone and Lee [Leone and Lee 1996] consistently generate code significantly more quickly than tcc; as we described above, their system provides less functionality and flexibility than 'C.

7. CONCLUSION

This paper has described the design and implementation of 'C, a high-level language for dynamic code generation. 'C is a superset of ANSI C that provides dynamic code generation to programmers at the level of C expressions and statements. Not unlike LISP, 'C allows programmers to create and compose pieces of code at run time. It enables programmers to add dynamic code generation to existing C programs in a simple, portable, and incremental manner. Finally, the mechanisms that it provides for dynamic code generation can be mapped onto statically typed languages other than ANSI C.

tcc is a portable and freely available implementation of 'C. Implementing 'C demonstrated that there is an important trade-off between the speed of dynamic code generation and the quality of the generated code. As a result, tcc supports two runtime systems for dynamic code generation. The first of these, VCODE, emits code in one pass and only performs local optimizations. The second, ICODE, builds an intermediate representation at run time and performs other optimizations, such as global register allocation, before it emits code.

We have presented several example programs that demonstrate the utility and expressiveness of 'C in different contexts. 'C can be used to improve the performance of database query languages, network data manipulation routines, math libraries, and many other applications. It is also well-suited for writing compiling interpreters and "just-in-time" compilers. For some applications, dynamic code generation can improve performance by almost an order of magnitude over traditional C code; speedups by a factor of two to four are not uncommon.

Dynamic code generation with tcc is quite fast. VCODE dynamically generates one instruction in approximately 100 cycles; ICODE dynamically generates one instruction in approximately 600 cycles. In most of our examples, the overhead of dynamic code generation is recovered in under 100 uses of the dynamic code; sometimes it can be recovered within one run.

'C and tcc are practical tools for using dynamic code generation in day-to-day programming. They also provide a framework for exploring the trade-offs in the use and implementation of dynamic compilation. A release of tcc, which currently runs on MIPS and SPARC processors, is available at `http://pdos.lcs.mit.edu/tickc`.

ACKNOWLEDGMENTS

Vivek Sarkar was instrumental in the development of the linear scan register allocation algorithm, which grew out of his work on spill-code minimization within a single basic block. Eddie Kohler provided valuable feedback on the language and on VCODE register allocation. Jonathan Litt patiently rewrote parts of xv in 'C while tcc was still immature, and thus helped us find several bugs.

A. 'C GRAMMAR

The grammar for 'C consists of the C grammar given in Harbison and Steele's C reference manual [Harbison and Steele Jr. 1991] with the additions listed below, and the following restrictions:

—An *unquoted-expression* can only appear inside a *backquote-expression*, and cannot appear within another *unquoted-expression*.

—A *backquote-expression* cannot appear within another *backquote-expression*.

—cspecs and vspecs cannot be declared within a *backquote-expression*.

unary-expression: backquote-expression | *unquoted-expression*
unquoted-expression: at-expression | *dollar-expression*
backquote-expression: ' *unary-expression* | ' *compound-statement*
at-expression: @ *unary-expression*
dollar-expression: $ *unary-expression*
pointer: cspec type-qualifier-list$_{opt}$ | *vspec type-qualifier-list*$_{opt}$
| *cspec type-qualifier-list*$_{opt}$ *pointer* | *vspec type-qualifier-list*$_{opt}$ *pointer*

REFERENCES

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S., AND BERSHAD, B. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.* Philadelphia, PA, 149–159.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* Copper Mountain, CO, 267–284.

BIRRELL, A. D. AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2,* 1 (Feb.), 39–59.

BRIGGS, P. AND HARVEY, T. 1994. Multiplication by integer constants. `http://softlib.rice.edu/MSCP`.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Computer Languages 6*, 47–57.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of PLDI '89.* Portland, OR, 146–160.

CLARK, D. D. AND TENNENHOUSE, D. L. 1990. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990.* Philadelphia, PA.

CONSEL, C. AND NOEL, F. 1996. 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, 145–156.

DEUTSCH, P. AND SCHIFFMAN, A. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages.* Salt Lake City, UT, 297–302.

DRAVES, S. 1995. Lightweight languages for interactive graphics. Technical Report CMU-CS-95-148, Carnegie Mellon University. May.

ENGLER, D. AND PROEBSTING, T. 1994. DCG: An efficient, retargetable dynamic code generation system. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 263–272.

ENGLER, D. R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.* Philadelphia, PA, 160–170. `http://www.pdos.lcs.mit.edu/~engler/ vcode.html`.

ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1995. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages.* St. Petersburg, FL, 131–144.

ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. 1995. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* Copper Mountain Resort, Colorado, 251–266.

FORSYTHE, G. E. 1977. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ.

FRASER, C. 1980. copt. `ftp://ftp.cs.princeton.edu/pub/ lcc/contrib/copt.shar`.

FRASER, C. W. AND HANSON, D. R. 1990. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Department of Computer Science, Princeton University.

FRASER, C. W. AND HANSON, D. R. 1995. *A retargetable C compiler: design and implementation.* Benjamin/Cummings Publishing Co., Redwood City, CA.

FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. 1992. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices 27,* 4 (April), 68–76.

GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. 1997. Annotation-directed run-time specialization in C. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Amsterdam, The Netherlands.

HARBISON, S. AND STEELE JR., G. 1991. *C, A Reference Manual*, Third ed. Prentice Hall, Englewood Cliffs, NJ.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation.* Orlando, Florida, 326–335.

KELSEY, R., CLINGER, W., REES, J., (EDITORS), ET AL. 1998. Revised^5 *Report on the Algorithmic Language Scheme.* `http://www-swiss.ai.mit.edu/~jaffer/r5rs_toc.html`.

KEPPEL, D. 1991. A portable interface for on-the-fly instruction space modification. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* Santa Clara, CA, 86–95.

KEPPEL, D., EGGERS, S., AND HENRY, R. 1991. A case for runtime code generation. TR 91-11-04, University of Washington.

KEPPEL, D., EGGERS, S., AND HENRY, R. 1993. Evaluating runtime-compiled value-specific optimizations. TR 93-11-02, Department of Computer Science and Engineering, University of Washington.

LEONE, M. AND DYBVIG, R. K. 1997. Dynamo: A staged compiler architecture for dynamic program optimization. Tech. Rep. 490, Indiana University Computer Science Department. Sept.

LEONE, M. AND LEE, P. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.* Philadelphia, PA, 137–148.

OUSTERHOUT, J. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA.

PIKE, R., LOCANTHI, B., AND REISER, J. 1985. Hardware/software trade-offs for bitmap graphics on the Blit. *Software—Practice and Experience 15,* 2 (Feb.), 131–151.

POLETTO, M., ENGLER, D. R., AND KAASHOEK, M. F. 1997. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.* Las Vegas, NV, 109–121.

POLETTO, M. AND SARKAR, V. 1998. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems.* (To appear).

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C*, Second ed. Cambridge University Press, Cambridge, UK.

Pu, C., Autry, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. 1995. Optimistic incremental specialization: streamlining a commerical operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles.* Copper Mountain, CO.

Pu, C., Massalin, H., and Ioannidis, J. 1988. The Synthesis kernel. *Computing Systems 1,* 1, 11–32.

SPARC International 1994. *SPARC Architecture Manual Version 9.* SPARC International, Englewood Cliffs, New Jersey.

Steele Jr., G. 1990. *Common Lisp*, Second ed. Digital Press, Burlington, MA.

Thekkath, C. A. and Levy, H. M. 1993. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems 11,* 2 (May), 179–203.

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

Wall, L., Christiansen, T., and Schwartz, R. 1996. *Programming Perl.* O'Reilly & Associates, Sebastopol, CA.

Witchel, E. and Rosenblum, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems.* Philadelphia, PA, 68–79.

Received October 1997, Revised May 1998, Accepted June 1998.