# tcc: A Template-Based Compiler for ‘C

Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek
{maxp, engler, kaashoek}@lcs.mit.edu
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

## Abstract

Dynamic code generation is an important technique for improving the performance of software by exploiting information known only at run time. ‘C (*Tick C*) is a superset of ANSI C that, unlike most prior systems, allows high-level, efficient, and machine-independent specification of dynamically generated code. ‘C provides facilities for dynamic code generation within the context of a statically typed, imperative language closely related to the language most widely used in systems development.

This paper describes `tcc`, a compiler currently being written for ‘C. `tcc` has two objectives: (1) to deliver a complete, solid implementation of ‘C, and (2) to minimize the run-time costs of dynamic code generation. `tcc` implements dynamic code generation by emitting templates, segments of binary code which at run time can be combined and completed with the values of registers, stack offsets, and constants. `tcc` also allows some decisions about storage allocation and instruction selection to occur at run time. This provides flexibility in combining arbitrary pieces of dynamic code, while allowing run-time code generation to occur very efficiently.

## 1 Introduction

Dynamic code generation improves the performance of software by enabling the use of information unavailable at compile time. It has been widely applied by the language and systems communities to improve the performance of everything from operating systems [1, 7, 18, 19] to windowing operations [17], dynamically typed languages [2, 4, 13], simulators [21, 25, 23] and matrix manipulations [8].

‘C [5] is a superset of ANSI C that makes dynamic code generation both practical and efficient. Practicality stems from the ability to specify code in a high-level, machine independent fashion, building on the facilities already available in C. Using ‘C, it is possible to synthesize arbitrary functions at run time, obtaining the advantages of languages that use partial evaluation. In addition, since ‘C is a superset of C, it enables the user to incrementally rewrite performance-critical sections of real systems. Efficiency is the result of static typing, which greatly reduces the cost and complexity of run-time code generation (for example, by making dynamic type-checking unnecessary). ‘C is thus a tool both for improving efficiency through the use of run-time information, and for expressing programming constructs unavailable in previous imperative languages.

This paper presents `tcc`, an efficient, template-based compiler currently being developed for ‘C. In creating `tcc`, we want to provide a solid and fast implementation of ‘C, adequate for systems programming, and to minimize the run-time costs of dynamic code generation. In order to maximize the efficiency of run-time code generation, `tcc` employs VCODE [6] as a substrate for emitting code at run time. VCODE is a system of templates for dynamic code generation, which presents to its client an interface similar to that of an idealized RISC architecture. VCODE emits code in-place, consuming little space and requiring as few as 30 instructions per generated instruction. This is a significant improvement over many previous dynamic code generation systems, such as DCG [8] (used to implement an experimental version of ‘C), which manipulate relatively complex data structures at run time. By using VCODE’s template-based approach, we expect the performance of dynamic code generation to improve significantly.

The rest of the paper is structured as follows: Section 2 gives a short overview of ‘C. Section 3 describes the current design and implementation of `tcc`, illustrating the details with examples where necessary. Section 4 discusses the current status of the implementation and future work aimed at improving both the speed of code generation and the quality of emitted code. Section 5 presents related work. Section 6 concludes.

## 2 Overview of ‘C

This section gives a brief overview of the ‘C language. A complete description can be found in [5]. In the following sections, *dynamic code* refers to dynamically generated code; *static code* is all other code. In ‘C, dynamic code is *specified* at run time; the specifications can then be *composed* to build larger specifications, or *instantiated* (compiled at run time) to produce executable code. *Compile time* refers to static compile time; *specification time* denotes the specification and composition of code, *instantiation time* denotes the compilation of dynamic code, and *run time* refers to dynamic code being executed.

‘C is a clean extension of ANSI C. It does not affect the semantics of the core language, enables flexible and expressive specification of dynamically generated code (henceforth referred to as *dynamic code*), and is amenable to an efficient implementation. Three major features determine these characteristics:

- ‘C has a static type system. This results in a limited loss in flexibility, but greatly reduces the cost and complexity at instantiation time, since type checking and instruction selection can be moved to static compile time.
- Dynamic code specified in ‘C is lexically scoped. Like a static type system, this decreases the cost of dynamic code generation.

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

- ‘C currently allows for the specification of only one dynamic function at a time; a function is instantiated, and its specification ends, with a call to the compile library function. This reduces the complexity (both internal to the compiler and visible to the programmer) inherent in managing multiple code specification “contexts.”

## 2.1 ‘C Language

‘C adds two type constructors and three unary operators to ANSI C. A small library implements much of the dynamic code generation functionality, thus minimizing changes to the original language and decreasing the complexity of the compiler.

The two new type constructors, cspec and vspec, are both postfix-declared types. A cspec or vspec has an associated *evaluation type*. The evaluation type allows dynamic code to be statically typed, enabling the compiler to select instructions and perform type checking at compile time. The three new unary operators, ‘, @, and $, have the same precedence as the standard unary prefix operators.

### 2.1.1 The ‘ Operator

Dynamic code is specified using the ‘ (backquote, or “tick”) operator. ‘ applied to an expression or compound statement indicates that code corresponding to that expression or statement should be generated at run time. Since ‘C currently disallows the generation of code-generating code, ‘ does not nest.

Dynamic code is lexically scoped. As a result, free variables within a ‘ expression can capture variables in enclosing static code. The use of such a variable after its enclosing scope has been exited is undefined.

The compiler must be able to determine statically that a change in control flow is legal. As a result, break, continue, switch, case, and goto statements cannot be used to transfer control outside of their enclosing ‘ expressions. Since dynamic code is always implicitly within a function, this restriction does not apply to return statements.

### 2.1.2 cspec Types

The type of a dynamic code specification is a cspec type (for *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. Applying ‘ to a compound statement yields a result of type void cspec. The type void cspec is the type for a generic cspec type (analogous to the use of void as a generic pointer). The code generated by ‘ may include implicit casts used to reconcile the result type of ‘ with its use; the standard promotion rules of ANSI C apply.

Cspecs can be compiled using the compile function, part of the ‘C standard library (see Section 2.2). compile returns a void function pointer, which can then be cast to the appropriate type. The following code fragment dynamically constructs and instantiates a “hello world” procedure:

```
void cspec c = `{ printf("hello world\n"); };
/* Compile and call the result.
   The TC_V indicates that the return type is void. */
compile(c, TC_V)();
```

### 2.1.3 vspec Types

The type of a dynamically generated lvalue is a vspec type (for *variable specification*); the evaluation type of the vspec is the type of the lvalue. Vspecs allow dynamically generated lvalues to be statically typed. Objects of type vspec may be initialized by calling the ‘C library functions param and local. param is used to create a parameter for the function under construction; local is used to reserve space in its activation record (or allocate a register if possible). void vspec is used as a generic vspec type.

The @ operator (see Section 2.1.4) is used to incorporate vspecs into cspecs. An incorporated vspec can be used both as an lvalue and an rvalue inside a backquote expression. As a simple example, the following function returns code for a function that takes a single integer argument, adds one to it, and returns the result:

```
void cspec plus1(void) {
  /* param takes an argument index and its type. */
  int vspec i = (int vspec) param(0, TC_I);
  return `{ return @i + 1; };}
```

Vspecs allow us to construct functions that take a run-time-determined number of arguments, often in a more elegant and efficient manner than argument marshaling or the variadic function facilities provided by ANSI C.

### 2.1.4 The @ Operator

The @ operator allows dynamic code and variable specifications to be combined (or *composed*) into larger specifications. @ can only be applied inside a ‘ expression; legal operands must be cspecs or vspecs, and are evaluated at specification time. @ “dereferences” cspecs and vspecs: 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. The following example composes two cspecs into a third:

```
/* Compose c1 and c2. Evaluation of c yields "9". */
int cspec c1 = `4, cspec c2 = `5;
int cspec c = `(@c1 + @c2);
```

Applying @ to a function that returns a cspec or a vspec causes the function to be called at specification time and its result to be incorporated into the backquote expression.

### 2.1.5 The $ Operator

The $ operator allows run-time constants to be incorporated into dynamic code. Like @, $ can appear only within a ‘ expression. $ evaluates its operand at specification time; the resulting value is incorporated as a *run-time constant* into the containing cspec. $ may be applied to any expression within dynamic code that is not of type cspec or vspec. The use of $ is illustrated in the code fragment below.

```
int cspec c1, cspec c2;
void cspec c;
int x = 1;

c1 = `$x;      /* Bind x at specification time */
c2 = `x;       /* Bind x at run time */
c = `{ printf("$x = %d, x = %d\n", @c1, @c2); };
x = 14;
/* Compile and run: will print "$x = 1, x = 14". */
compile(c, TC_V)();
```

## 2.2 The ‘C Standard Library

Many of the features of ‘C are implemented through a small library. This minimizes the actual language changes necessary for ‘C, and facilitates changes in implementation and functionality. We mentioned earlier the compile, local, and param functions. These serve, respectively, to instantiate a dynamic code specification, and to dynamically add locals and incoming arguments to the function being specified.

The functions push and arg help to dynamically create function calls, whereas self returns a pointer to the function that the next invocation of compile will return, enabling the creation of recursive dynamic code. These and other elements of the ‘C standard library are described in depth in [5].

Many library functions (local, param, arg, push, and compile) expect types as arguments. The library provides enumerated types for specifying this information: types are specified by enumeration constants whose names are the the first letter(s) of the types prefixed by TC_. For example, unsigned short is specified by TC_US. Additionally, the flag TC_REGISTER can be bitwise-ored with the type to indicate that the allocated object will not have its address taken. This solution can improve the performance of generated code without adding much to run-time costs or the complexity of the code generator.

## 3 `tcc`: a Compiler for ‘C

This section describes in some detail the current implementation of `tcc`. The base on which `tcc` is being developed is `lcc` [12, 9]. We chose `lcc` because it is a simple, production-quality C compiler with a relatively large user base. Although it performs no global optimizations, `lcc` emits code of fair quality by relying on local common subexpression elimination and other peephole optimizations. Its small size, detailed documentation, and portability made it preferable to other compilers.

Section 3.1 describes the structure of `tcc`. Section 3.2 discusses issues relevant to the front end, such as parsing and intermediate code generation. Section 3.3 presents the `tcc` back end, including its use of VCODE, whereas Section 3.4 outlines the process of dynamic code generation.

### 3.1 Overview

`tcc` inherits most of its code from `lcc`, but it differs from it in important ways (see Figure 1). When compiling a ‘C program, `tcc` uses two different back ends. Static C code (everything outside a ‘ expression) is compiled by the traditional back end for the target architecture, which emits assembly code directly.

In the case of dynamic code, however, `tcc` must emit code which, at run time, will generate the specified code. To do this, `tcc` employs its “dynamic” back end. For each ‘ expression, this back end generates a C function (which we refer to as the *code-generating* function for that expression) that contains VCODE instructions.

VCODE is a set of C macros and an associated library, designed for fast dynamic code generation. The system presents to its client, in this case `tcc`, an interface which looks like that of an idealized load/store RISC architecture. VCODE generates code in-place in “zero passes”: code generation in VCODE is essentially transliteration of VCODE instructions into the instructions of the target architecture.

After emitting C code for the code-generating functions, `tcc` compiles it like any other C program, emitting assembly code which, at run time, will perform the required code generation.

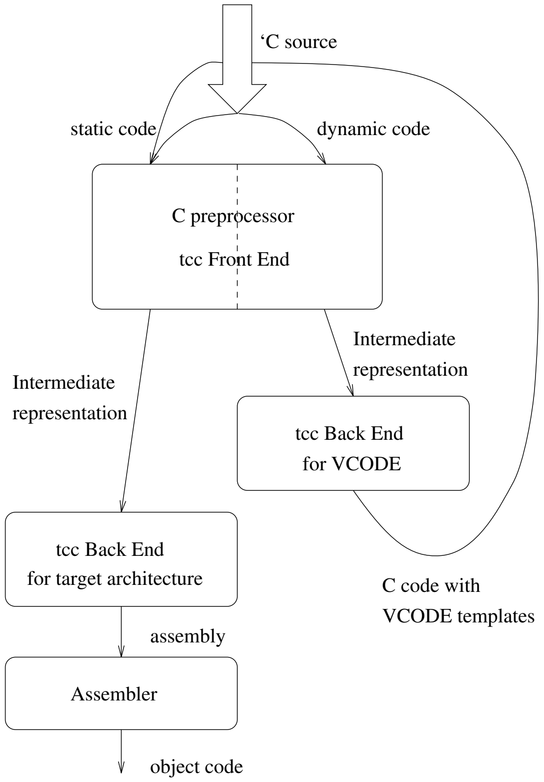


Figure 1: Overview of the `tcc` compilation process (static compile time).

### 3.2 Parsing and Intermediate Representation

`lcc`’s front end is built around a recursive descent parser. We modified this parser to accept the three new unary operators described above. To allow type checking, we extended `lcc`’s type symbol table to include cspecs and vspecs.

Intermediate code generation consists of appending IR “trees” (directed acyclic graphs that are roughly an abstract syntax tree of statements in the parsed code) to a growing “forest.” The intermediate code for an entire function consists of a “code list” of forests. Unlike `lcc`, which has only one code list, `tcc` uses two, one for static code, one for dynamic code.

When inside a ‘ expression, the parser appends most newly generated IR trees to this second queue. The parser must also take care of a few details:

- It tracks goto statements and labels to ensure that a goto does not transfer control outside of the cspec body.
- It parses the operands of $ and @ operators into `lcc` IR trees, but does not emit them immediately into a code list, simply storing them for later processing.
- It keeps a list of the free variables of a ‘ expression. These are identifiers inside the body of a cspec but which are neither declared within its scope nor the arguments of an @ or $ operator.

Having emitted the code for a ‘ expression into the dynamic code list, `tcc` generates static code to allocate and assign a closure for the cspec. The closure is a heap-allocated region which stores information regarding the free variables, run-time constants, and code and variable specifications found in the expression. It has three major parts:

- A function pointer, whose value is the code-generating function for the cspec.
- The address of each free variable in the ‘ expression. The code-generating function will emit code in which each such pointer is a constant. At run time, all operations on free variables in the ‘ expression then occur through this pointer. The dynamic code thus always has access to the most recent value of the free variable, rather than to that valid at specification time. This results in the behavior illustrated in the example in Section 2.1.5.

  Combined with the availability of run-time constants, this feature gives the programmer much flexibility. However, it means that it is illegal for a variable to be both declared register and a free variable of a cspec.
- The values of all expressions which are run-time constants or variable or code specifications. These are computed by emitting the stored IR trees mentioned above.

Internally, a ‘ expression is represented by a pointer to its closure. Variables of type cspec are implemented just like pointers (they have the same size, alignment requirements, etc.). Thus, when a ‘ expression is assigned to a cspec variable, the resulting code is essentially an assignment to a pointer.

Implementing vspecs is simple. Like a cspec, a vspec is represented by a pointer to a structure. In the case of vspecs, the structure contains information on whether the vspec is in a register or not, what its register or stack offset is, whether it has been addressed, and so forth. The structure is created by a standard library function, such as local or param, which returns a pointer to it appropriately cast as a vspec. Additionally, all local variables defined within a ‘ expression are implemented as vspecs: we describe this in the following section.

## 3.3 Code Generation at Compile Time

When `lcc` finishes parsing a function, it consumes the code list of intermediate code corresponding to that function, performing common subexpression elimination and other optimizations at the level of basic blocks, selecting instructions, allocating registers, and finally emitting code. Target-specific information necessary for instruction selection and code generation is specified in a machine description file which `lburg` [10, 11] uses to generate parts of the back end.

`tcc` follows this general pattern, but extends it by using two back ends: one is for static code, and generates assembly directly, whereas the other is for dynamic code, and generates C code containing VCODE macros. The back end for dynamic code traverses the dynamic code list, identifying the beginning and end of each ‘ expression. It then translates the IR for each expression into a separate code-generating function. Having emitted all the code-generating functions, `tcc` then compiles them like any other C program, and passes them to the assembler together with the original static code.

Each code-generating function takes a void * as an argument, which is used to reference the closure structure. A code-generating function's type depends on the type of the corresponding cspec. If this is an expression, the code-generating function's type is VCODE's reg_type, because the function must return the register in which the result of the expression is stored. Otherwise, the code-generating function is void.

The following example illustrates the structure of code-generating functions. Consider the following procedure, which dynamically constructs a function which returns the sum of its n arguments:

```
/* Construct cspec to sum n integer arguments. */
void cspec construct_sum(int n) {
  int i, cspec c = `0;
  for (i = 0; i < n; i++) {
    /* create a parameter */
    int vspec v = (int vspec) param(i, TC_I);
    /* Add param 'v' to current sum. */
    c = `(@c + @v);
  }
  return `{ return @c; };
}
```

Each ‘ expression results in a code-generating function, as well as in static code to heap-allocate a closure containing that expression's free variables, vspecs and cspecs, and a pointer to the code-generating function. An example of the code-generating function for the initial cspec, '0, appears below. It contains VCODE primitives to emit, at run time, an instruction that assigns 0 to a temporary register.

```
v_reg_type __cgf0(void *__c_arg)
{
  /* declare a register */
  v_reg_type r3;
  /* allocate temporary reg */
  v_getreg(&r3,V_I,V_TMP);
  /* move 0 into temporary reg */
  v_setii(r3,0);
  /* return the result register */
  return (r3);
}
```

In turn, code for each of the c = ‘(@c+@v) expressions will be generated at run time by the following function:

```
v_reg_type __cgf1(void *__c_arg)
{
  /*  __closure_internal_1 is the type of the closure structure
      for the current cspec (the lhs of c = ‘(@c+@v)). */
  struct __closure_internal_1 *__c = __c_arg;

  v_reg r3;
  v_reg r14;
  v_spec v1;
  /* allocate temporary reg */
  v_getreg(&r3,V_I,V_TMP);

  /* find the register in which the cspec @c returns its value:
     + __c–>c1 points to the closure structure of the cspec
       referenced by @c.
     + __c–>c1–>code is the code–generating code for @c
       (a function similar to this one, or  __cgf0, which takes
        a pointer to the closure structure for @c as an
        argument). */
  r14 = ((struct __closure_internal*)__c→c1)→code
          ((void *)__c→c1);

  /* __c–>v1 points to the v_spec structure of the vspec
     referenced by @v in (@c+@v). */
  v1 = (v_spec)__c→v1;
```

```
  if (v1→regp) {
    /* if @v is in a register do a reg–>reg addition */
    v_addi(r3,r14,v1→reg);
  } else {
    /*  first load the value from  memory, then add */
    v_ldi(r3,v_lp,v1→offset);
    v_addi(r3,r14,r3);
  }
  /* return the result register */
  return (r3);
}
```

Lastly, the expression '{ return @c; } results in the following code. Note that in this case the cspec contains a statement: it does not need to return a value in a register, hence the type of the code-generating function is void.

```
void __cgf2(void *__c_arg)
{
  struct __closure_internal_3 *__c = __c_arg;
  /* declare a register */
  v_reg_type r14;

  /* like above, find the register containing the result of @c */
  r14 = ((struct __closure_internal*)__c→c1)→code
          ((void *)__c→c1);
  /* emit a return instruction */
  v_reti(r14);
}
```

Since VCODE provides a clean RISC interface, not much different, for example, from that of the MIPS R3000 processor, most code generation and instruction selection occurs as it does in other back ends. We mention here some of the major differences.

- All variables local to the ' expression, including compiler-generated temporaries, are implemented as vspecs. In order to enable the composition of arbitrary cspecs, storage class information (whether a variable is in a register or not, what its stack offset is, etc.) cannot be fixed at compile time. Therefore, a call to an analogue of the user-visible local function is performed to allocate space for each local and temporary.
- The structure of code is dependent on the storage class of vspecs in the code. This is a very important point, and the greatest difficulty in generating code.

  At present, when performing instruction selection and code generation for the VCODE back end, `tcc` allocates registers and stack slots like it does for other back ends. When allocating vspecs for locals and temporaries, it explicitly requests a vspec with the appropriate storage class from VCODE. This works well when not many cspecs are composed together. If multiple cspecs are combined, however, and the total resulting number of vspecs with a register storage class allocated at a given time exceeds the number on the target machine, VCODE can either report an error and fail, or return a "virtual register" which is actually a location on the stack. The former solution is unacceptable because it makes the correctness of some 'C programs implementation-dependent; the latter is poor because it decreases performance. We discuss this further in Section 4.
- Composition of cspecs (including cspec b into the body of cspec a) is implemented by simply applying b's code-generating function to its closure pointer within a's code-generating function. If b returns a value, the return register is returned by its code-generating function, and can be used as an argument to another VCODE instruction. This simple scheme is sufficient, since we disallow run-time code generation of run-time code-generating code in this version of 'C.
- Labels and registers are explicitly declared. The VCODE interface requires that labels and registers, just like vspecs, be explicitly declared and defined before being used.

### 3.4 Code Generation at Run Time

At instantiation time, code for a cspec is emitted by calling the compile function with the cspec as an argument. compile uses VCODE to first allocate space (either registers or offsets from the stack) for locals and arguments dynamically specified with local and param. It then invokes the code-generating function for the cspec on the closure itself. The VCODE instructions in the code-generating function then perform most of code generation. When the function returns, compile just needs to link the resulting code, reset the information regarding dynamically generated locals and parameters, and return a pointer to the generated code. This pointer is to a function of type void; the user is responsible for casting it to the appropriate type for the given cspec, and can then call it like any other function pointer.

To make this all more concrete, consider once again the construct_sum example above. When the user invokes compile on the return value of construct_sum, compile first performs some bookkeeping to manage the dynamically generated parameter v. It subsequently invokes __cgf2 on the pointer to the closure for the expression '{ return @c; }. __cgf2 in turn invokes the code-generating function for the cspec that it contains, and so forth. In each code-generating function, the VCODE macros lay out in memory the binary code for the instructions to which they each correspond. When this is over, compile links the resulting code, and returns a pointer to it. Dynamic compilation is complete, and the user may employ this pointer to invoke the generated code.

## 4 Status and Future Work

A pre-beta implementation of `tcc` is essentially complete at the time of this writing. It implements all of 'C except for run-time constant aggregates, and generates code for the SPARC and MIPS R3000 processors. This version of the compiler, however, does not attempt inter-cspec analysis when allocating registers at runtime, and it makes only limited use of information about run-time constants. In addition, it is not yet well-tuned. We discuss each of these issues in turn.

### 4.1 Dynamic Register Allocation

Currently, the code-generating functions emitted by `tcc` do not implement any form of inter-cspec register allocation. It is thus possible that an expression generated dynamically require many more registers than its statically defined equivalent, causing spilling to the stack and an unnecessarily large number of calls to the dynamic register allocation routine. Consider the statement { s = '1; }, followed by a loop containing { s = '(x+@s); }, similarly to the code in construct_sum. The code generated by iterating through the loop $n$ times corresponds to an expression tree of depth $n$, as in Figure 2. When compiled statically, the expression can be computed using 2 registers. In the current implementation, however, a new register is allocated every time the code-generating function for the cspec in the loop is called, requiring spilling after only a few iterations. Beside making the resulting code inefficient, this approach adds to the cost of dynamic compilation, since it requires unnecessary calls to the dynamic register allocation routine. We are exploring different approaches for efficiently propagating register requirements across code-generating functions at run time.

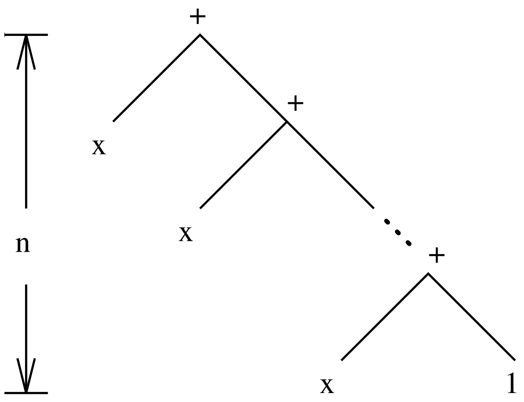


Figure 2: Register allocation problem.

### 4.2 Optimizations for Run-Time Constants

Currently, arithmetic and other expressions involving run-time constants are evaluated at instantiation time, so that the dynamically generated code contains only references to constants. For example, the expression `‘{ x = 4+$a[i]; }` compiles at run time to a single instruction, which moves the numeric value 4+a[i] available at specification time into the register corresponding to x. However, more aggressive transformations are clearly possible. For example, a branch based on a run-time constant can be evaluated only once, at instantiation time: the code for the case not corresponding to the value of the run-time constant need never be emitted.

In the limit, it is possible to implement a relaxation algorithm for determining run-time constant values based on other run-time constant values, at least within a given cspec. This is impractical, however, since it greatly increases run-time code generation costs. We plan to further explore the tradeoffs involved in performing different levels of optimizations during run-time code generation.

### 4.3 Performance Tuning

Our naive register allocation strategy, together with a poorly tuned initial implementation of code-generating functions, limits the quality of emitted code and slows down code generation. For small tests (such as the construct_sum example above), `tcc` currently generates code at run time at an end-to-end cost of about 80 instructions per generated instruction. For these benchmarks, however, a large fraction of this cost is incurred only once, at the beginning and end of compilation. Ignoring these start and end factors, the cost of code generation is approximately 50 instructions per generated instruction. This is about six times faster than the original ‘C compiler, which was based on DCG [8], but still twice as expensive as the amortized code generation cost of bare VCODE. We expect to obtain improvements by tuning the implementation and improving register allocation.

## 5 Related Work

Dynamic code generation has been used to great advantage in the systems community. By using dynamic code generation, programs can achieve speed-ups exceeding an order of magnitude. Unfortunately, to date most systems for dynamic code generation have been ad-hoc, hard to program and retarget, and sometimes paying considerable run-time overhead. ‘C promises to make dynamic code generation easier and more practical.

`tcc` will be the first mature implementation of ‘C [5]. ‘C was initially implemented to make use of DCG [8]. That implementation offered several improvements over the original DCG system, among them a high-level code specification interface, and the opportunity to perform static analysis to decrease the overhead of generating code at run time. `tcc`'s contribution is to further lower the cost of code generation. Employing VCODE [6] to generate code dynamically, `tcc` eschews the overhead of maintaining complex data structures such as DCG's intermediate representation (based on the IR of `lcc` [12]) at run time.

Many languages, such as most Lisp dialects [20, 22], Tcl [16], and Perl [24], provide an “eval” operation that allows code to be generated dynamically. This approach is flexible but has a high run-time cost, since these languages are dynamically typed.

Consel and Noël [3] describe a technique for specializing programs with respect to run-time invariants. Their system exploits user-annotated run-time constants, using templates to dynamically emit specialized code. Similarly, Leone and Lee [14, 15] use programmer-supplied hints to perform compile-time specialization in a first-order subset of ML, with good results. Like `tcc`, both of these systems use templates to achieve low code generation costs. The difference between them and `tcc` is the level at which they provide dynamic code generation to the user. Both Consel and Leone use dynamic code generation to improve performance of compiled code, largely hiding the dynamic code generation process from the user. ‘C, on the other hand, aims to expose run-time code generation to the user, exploiting dynamic code generation for expressiveness as well as performance.

## 6 Conclusion

We have presented `tcc`, the first real implementation of ‘C, a language for specifying dynamic code generation in a high-level, machine-independent, and efficient manner. `tcc` employs VCODE to generate code at run time, reducing the overhead of dynamic code generation with respect to other implementations of similar systems. Exciting work remains to be done in further reducing code generation costs and improving the quality of dynamically generated code.

## References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

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

[3] C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, FL, January 1996.

[4] P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.

[5] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, 1995.

[6] D.R. Engler. VCODE: A very fast, retargetable, and extensible dynamic code generation substrate. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. (To appear).

[7] D.R. Engler, M.F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, December 1995.

[8] D.R. Engler and T.A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–272, San Jose, CA, October 1994.

[9] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[10] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[11] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1991.

[12] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. Technical Report CS-TR-270-90, Princeton University, Dept. of Computer Science, Princeton, New Jersey, July 1990.

[13] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–335, Orlando, Florida, June 1994.

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

[15] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, may 1996. (To appear).

[16] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

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

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

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

[20] J. Rees, W. Clinger (editors), et al. Revised$^4$ report on the algorithmic language Scheme. AIM 848b, MIT AI Lab, November 1992.

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

[22] G.L. Steele Jr. *Common Lisp*. Digital Press, second edition, 1990.

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

[24] Larry Wall. *The Perl Programming Language*. Prentice Hall Software Series, 1994.

[25] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS*, Philadelphia, PA, 1996. (To appear).