tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation

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

tcc is a compiler that provides efficient and high-level access to dynamic code generation. It implements the ‘C (“Tick-C”) programming language, an extension of ANSI C that supports dynamic code generation [15]. ‘C gives power and flexibility in specifying dynamically generated code: whereas most other systems use annotations to denote run-time invariants, ‘C allows the programmer to specify and compose arbitrary expressions and statements at run time. This degree of control is needed to efficiently implement some of the most important applications of dynamic code generation, such as “just in time” compilers [17] and efficient simulators [10, 48, 46].

The paper focuses on the techniques that allow tcc to provide ‘C’s flexibility and expressiveness without sacrificing run-time code generation efficiency. These techniques include fast register allocation, efficient creation and composition of dynamic code specifications, and link-time analysis to reduce the size of dynamic code generators. tcc also implements two different dynamic code generation strategies, designed to address the tradeoff of dynamic compilation speed versus generated code quality. To characterize the effects of dynamic compilation, we present performance measurements for eleven programs compiled using tcc. On these applications, we measured performance improvements of up to one order of magnitude.

To encourage further experimentation and use of dynamic code generation, we are making the tcc compiler available in the public domain. This is, to our knowledge, the first high-level dynamic compilation system to be made available.

1 Introduction

Dynamic code generation has recently attracted considerable interest. Unlike tcc, current systems generally do not provide an interface to dynamic code generation that is both flexible and easy to use. On one side, annotation-driven approaches allow the programmer to communicate high-level hints about run-time invariants to the compiler, but provide relatively limited code specification flexibility [1, 11, 32]. On the other, library-based approaches allow flexible specification of code, but require programmers to work with a low-level representation [14, 19].

This code specification flexibility is necessary to efficiently implement some of the most profitable applications of dynamic code generation, such as “just in time” compilers [17] and efficient simulators [10, 48, 46], which require the composition of arbitrary statements and expressions and the creation of routines with statically-unknown type signatures. tcc aims to provide the ease of specifying dynamic code at a high level while retaining the flexibility and expressiveness of low-level systems.

tcc compiles the ‘C programming language, an extension of ANSI C that provides mechanisms for specifying and dynamically composing arbitrary ANSI C expressions and statements. Although code composition is powerful and relatively easy to use, it truncates control flow information, reducing the opportunity for static analysis. Hence, implementing ‘C efficiently is challenging. tcc addresses this problem with several techniques, including algorithms for fast dynamic register allocation, efficient creation and analysis of dynamic code specifications, and a link-time analysis to reduce the size of dynamic code generators. In addition, tcc implements two different dynamic compilation strategies, designed to accommodate the tradeoff between the speed of code generation and the quality of generated code. When compilation time must be minimized, dynamic code generation and register allocation are performed in one pass; when code quality is most important, the system dynamically constructs and optimizes an intermediate representation prior to generating code.

These solutions are interesting in the context of dynamic compilation. Furthermore, some, such as a fast global register allocation algorithm, are also applicable to traditional, static compilers.

To characterize the effects of these techniques, we present performance results for eleven programs compiled using tcc. Our measurements show that ‘C programs employing dynamic code generation are up to an order of magnitude faster than their static ANSI C counterparts.

Dynamic code generation has existed for a long time [7, 10, 12, 16, 26, 29, 30, 36, 40, 48, 45, 46]. Unfortunately, despite its utility, programmers have never had portable, high-level access to it, so it has remained a curiosity rather than a popular technique. One goal of the ‘C project is to provide a real system that can be used in day-to-day development and for further research on dynamic code generation. This objective has determined many of our tradeoffs. In terms of language design, it led us to base our work on ANSI C and to keep our extensions within “the spirit of C.” In terms of implementation, it required that the compiler be publicly available and that it run on a variety of different architectures. Attention to these constraints has allowed us to produce a robust, efficient, and high-level dynamic code generation system available for public use.

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.

The following section describes how our compiler relates to other systems. Section 3 gives a concise overview of tcc's input language, ‘C. We discuss the basic architecture of tcc in Section 4, details of its code generation strategies in Section 5, and experimental results in Section 6.

2 Related Work

Dynamic code generation has a long history [29]. It has been used to increase the performance of operating systems [2, 16, 39, 40], windowing operations [36], dynamically typed languages [7, 12, 26], and simulators [48, 46].

Many languages, such as most Lisp dialects [41, 43], Tcl [35], and Perl [47], 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 issues relevant to retargeting dynamic code generation in [28]. 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 [30] demonstrated that dynamic code generation can be effective for several different applications.

Leone and Lee [31] use programmer-supplied hints to perform compile-time specialization in a simple functional language. Recently, they have extended their compiler, FABIUS, to accept a functional subset of ML [32]. They achieve low code generation costs using some of the techniques we independently derived for tcc. ‘C, however, provides the programmer with a wider range of mechanisms for dynamic code generation. Additionally, ‘C provides support for dynamic code generation in the context of ANSI C, a complex imperative language.

Another interesting automatic code generation system is Tempo [11], a general-purpose dynamic specializer for C. Like other automatic systems, it can be less flexible than ‘C. For example, the scope of Tempo's optimizations is limited by the usual challenges C presents to optimizing compilers (*e.g.*, unrestricted aliasing). No performance data is provided for Tempo, so it is difficult to compare the two implementations.

The dynamic compilation project at the University of Washington also uses automatic compiler support for detecting run-time constants [1]. Their compiler employs programmer annotations to indicate some run-time constants; the compiler computes what variables are derived run-time constants. Published results seem to indicate that it dynamically generates code up to an order of magnitude more slowly than tcc but, since it is integrated with the Multiflow optimizing compiler [33], we assume that the code it generates is superior in quality.

In theory, the three systems described above can generate good code with relatively low overhead by moving almost all the work to static compilation. In such systems, the static compiler emits templates that at run time are filled with appropriate values, optimized in relatively simple but effective ways, and emitted directly. In practice, however, only Leone and Lee [32] appear to generate code more quickly than tcc. Furthermore, improving performance in this way has a usability cost: no existing template-based system gives the programmer as much freedom and expressive power as ‘C. ‘C allows the user to dynamically compose pieces of code in arbitrary ways, and can thus provide functionality not present in plain C. Existing template-based systems, by contrast, use dynamic compilation solely for its performance advantages, extending to run-time the applicability of traditional optimizations such as copy propagation and dead code elimination.

We build on work done by Engler, Hsieh, and Kaashoek [15]. They describe the ‘C language and a prototype compiler for it. We provide the first real, high-performance implementation of ‘C. Our implementation uses an extension of VCODE [14], a fast, portable dynamic code generation system, as its target machine.

3 Language Overview

This section briefly describes ‘C in enough detail to provide context for the tcc compiler; a more complete discussion can be found in [15, 18]. ‘C extends ANSI C to support dynamic code generation. It introduces two unary operators, ‘ and $, and two postfix-declared type constructors. Programmers use these extensions to denote code that should be generated at run time. The units of code specification are ANSI C expressions, statements and variables. These specifications, created dynamically at *specification time*, can be *composed* to build larger specifications or *instantiated*, compiled at run time, to produce executable code. Just like ANSI C, ‘C is lexically scoped and statically typed.

Dynamic code is specified using the ‘ (backquote, or "tick") operator. Backquote applied to an expression or compound statement indicates that code corresponding to that expression or statement should be generated at run time. We refer to the resulting expression as a "tick-expression." For example, the tick-expression ‘4 specifies code to generate the integer constant 4. The type of a specified expression or statement 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 produces an object of type void cspec. An evaluation type allows dynamic code to be statically typed, enabling the compiler to do all type checking and some instruction selection at static compile time. Dynamically generated lvalues are of vspec type (*variable specification*); their evaluation type is the type of the lvalue.

When flow of control passes a tick-expression, that expression is said to be specified. However, before the code can be executed, it must be compiled, or instantiated. This operation is performed by the compile special form, which takes a cspec and a type as an argument and returns a function pointer having the corresponding return type. The following code fragment dynamically specifies and then instantiates a "hello world" procedure:

```
void cspec hello = ‘{ printf("hello world\n"); };
/* Compile and call the result */
(*compile(hello, void))();
```

Dynamic code composition is central to ‘C. References to objects of cspec or vspec type appearing in the body of a tick-expression are automatically converted to their corresponding evaluation types and incorporated into the code of the cspec in which they occur. For example, in the code below, compiling c has the same effect as compiling the cspec ‘(4+5).

```
/* Compose c1 and c2. Compiling c yields code '4+5' */
int cspec c1 = ‘4, cspec c2 = ‘5;
int cspec c = ‘(c1 + c2);
```

The $ operator allows run-time constants to be incorporated into dynamic code. $ can appear only within a tick-expression; it evaluates its operand at specification time, and 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.

```
void (*fp)(void);
int x = 1;

/* Compile simple function to demonstrate binding via $ */
fp = compile(`{ printf("$x = %d, x = %d\n", $x, x); }, void);
x = 14;
(*fp)(); /* Invoke function: will print "$x = 1, x = 14". */
```

'C has many other features, including facilities to create local variables and parameters at run time, generate function calls with run-time determined numbers of arguments, and dynamically create labels and jumps. These are implemented as special forms which the compiler translates to calls to a small library, reducing the number of actual language changes necessary for 'C, and facilitating changes in implementation and functionality. These and other features of 'C are described in depth in [15, 18].

4 Architecture

The design of tcc has been driven by three goals: providing flexibility to the programmer, minimizing the overhead of dynamic compilation, and generating high-quality code. Since many optimizations on dynamic code can be profitably performed only at run time, improvements in code quality are generally obtained at the expense of greater code generation time. One way to generate good code with low overhead is to structure the system so that most optimizations can be performed statically [1, 11, 32], but in all existing systems this ability comes at the expense of language flexibility and expressiveness (furthermore, in practice, only Leone and Lee [32] appear to generate code more quickly than tcc). 'C, by contrast, allows the user to dynamically compose arbitrary pieces of code, which makes static analysis of the dynamic code problematic.

The following subsections give an overview of tcc and discuss how tcc achieves its goal of performance and flexibility. We describe three important code generation phases: static compile time, dynamic specification time, and dynamic compile time. The subsequent section looks in detail at several of the optimizations and techniques we have developed to improve the performance of both dynamic code and the dynamic code generation process.

4.1 Overview

The tcc compiler is based on lcc [23, 22], a terse and portable compiler for ANSI C. lcc performs common subexpression elimination within extended basic blocks and uses lburg [38] to find the lowest-cost implementation of a certain IR-level construct, but it performs few other optimizations.

All parsing and semantic checking of dynamic expressions occurs at static compile time. Semantic checks are performed at the level of dynamically generated expressions. For each cspec, tcc performs type checking similarly to a traditional C compiler. 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 a highly 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 code is in turn compiled by the static back end. This process is illustrated in Figure 1.

Since the tradeoff between code generation speed and code quality is so important, tcc provides two dynamic back ends and corresponding run-time systems: the first emphasizes code generation speed over code quality, while the second inverts this tradeoff.

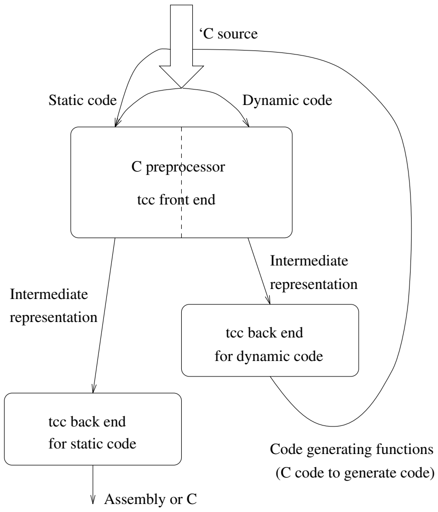

Figure 1: Overview of the tcc static compilation process.

4.2 Static compilation

During static compilation, every tick-expression is compiled to a *code generating function* (CGF), which is invoked at run time to generate code for dynamic expressions. This subsection briefly describes the static compilation process and the abstract machine that tcc uses for dynamic code generation.

Generating CGFs and Closures. All information necessary for dynamic compilation is maintained in CGFs and in dynamically-created closures. For each tick-expression, tcc statically generates both its code generating function and the code to allocate and initialize the corresponding closure.

CGFs and closures are used to support tcc's two-step approach to code generation. First, at specification time, the state of each tick-expression (*i.e.*, addresses of free variables and values of run-time constants created using $) is captured in a closure. Second, at dynamic compilation time, the code generating functions for each tick-expression process the closures and produce executable code. Closures are necessary to reason about composition and out-of-order specification of dynamic code. Eliminating closures would require generating code as soon as a dynamic expression is specified. This is a poor solution, since it does not solve the problem of propagating information across cspecs, and it may require that code fragments be copied to contiguous memory.

Cspecs are implemented by pointers to closures corresponding to tick-expressions. Objects of type cspec are therefore implemented just like pointers: they have the same size, alignment requirements, etc.. Closures are heap-allocated, but their allocation cost is greatly reduced (down to a pointer increment, in the normal case) by using arenas [20].

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 corresponding pointers to closures:

```
_tc_cspec_t i = ((_closure0 = (_closure0_t)_alloc_closure(4)),
                 (_closure0->cgf = _cgf0), /* code gen func */
```

```
                (_tc_cspec_t)_closure0);

_tc_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 ∗/
                (_tc_cspec_t)_closure1);
```

i's closure contains only a pointer to its code generating function. c, on the other hand, has more dependencies on its environment: its closure also stores run-time constants, pointers to free variables, and other information.

tcc's abstract machines. tcc compiles dynamic code to two abstract machines based on the VCODE dynamic code generation system [14]. Using abstract machines rather than machine-specific back ends significantly simplifies the structure of tcc, allowing it to ignore many machine-specific details of dynamic code generation and to compile dynamic code in essentially the same way for all the machines to which VCODE is ported (currently MIPS, SPARC, Alpha, and x86). While abstraction at this level implies some degradation in code quality, it has allowed us to build, with relatively little effort, a system that works on two different architectures (SPARC and MIPS) and is being ported to the x86 (a complete port to the Alpha would require more significant changes to the internals of the static compilation portion of tcc). As tcc matures, we may add machine-specific back ends. Since tcc preserves the features that make lcc retargetable, this process should not be difficult.

The first of tcc's abstract machines is the VCODE system itself [14]. 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 (*i.e.*, just information about one tick-expression), so the quality of the resulting code can frequently be improved. The second abstract machine, ICODE, addresses this problem: it provides an extended instruction set and, prior to emitting code, builds up and then optimizes an intermediate representation of all code for a dynamically specified function.

Section 5 discusses these abstract machines in detail. The following example, however, may help to visualize the contents of the CGFs and the interface to the abstract machines. The two functions below are the CGFs for the cspecs introduced in the previous section, for the case of the ICODE back end:

```
unsigned int _cgf0 (closure0_t c) {
  _tc_vspec_t itmp0 = _tc_local (I_I); /∗ int temporary ∗/
  i_seti(itmp0,5);  /∗ set it to 5 ∗/
  return itmp0;    /∗ return the location ∗/
}

void _cgf1 (closure1_t ∗c) {
  _tc_vspec_t itmp0 = _tc_local (I_I); /∗ some temporaries ∗/
  _tc_vspec_t itmp1 = _tc_local (I_I);
  i_ldii (itmp1,zero,c−>fv_k); /∗ addr of k ∗/
  i_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);
  i_addi (itmp1,itmp0,itmp1);
  i_reti (itmp1);   /∗ emit a return (not return a value) ∗/
}
```

_cgf0 is very simple: it allocates a temporary storage location, generates code to store the value 5 into it, and returns the location. This is exactly the meaning of '5. On the other hand, _cgf1 must do a little more work: the code that it generates (1) loads the value stored at the address of free variable k into a register, (2) multiplies it by the value of the run-time constant j, (3) adds this to the dynamic value of i, and (4) returns the result. Note that since i is a cspec, the code for "the dynamic value of i" is generated by calling i's code generating function.

Static phases of dynamic code generation. Fast dynamic code generation requires that most compilation work be done at static compile time. Thus, when optimizing code generation speed over quality of generated code, tcc performs as much instruction selection as possible statically. Both instruction selection based on operand types and register allocation for temporaries and variables not live across the reference to a cspec are done statically. Additionally, the intermediate representation of each tick-expression is processed by the common subexpression elimination and other local optimizations performed by the lcc front end. tcc also uses copt [21] to statically perform peephole optimizations on the code generating macros used by CGFs.

However, not all register allocation and instruction selection can occur statically. For instance, it is not possible in general to statically determine which vspecs or cspecs will be incorporated into another cspec when the program is executed. Hence, allocation of user-defined 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 quite inexpensive.

Optimizing for code quality, on the other hand, involves a more significant dynamic code generation cost. In this case, the CGFs contain ICODE, instead of VCODE, macros, and tcc does not precompute much statically. Rather than emitting code directly, the ICODE macros first build up a simple intermediate representation; the ICODE run-time system then makes two passes over this representation to allocate registers and perform other optimizations before emitting code. Both ICODE and VCODE are discussed further in subsequent sections.

Engineering: tcc's two static back ends. 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.

4.3 Dynamic specification time

At dynamic specification time, the 'C run-time system collects information about the environment of each tick-expression. This is a simple process, in which relevant portions of a tick-expression's environment are captured in a closure. An example appears in Section 4.2. The closure is heap-allocated and used to store four main types of information: (1) a function pointer to the statically generated CGF for the tick-expression; (2) the values of run-time constants bound via the $ operator; (3) the addresses of free variables; (4) pointers to objects representing the code and variable specifications composed inside the tick-expression. This information is used to create code during dynamic compilation.

4.4 Dynamic compilation

Dynamic compilation, or instantiation, involves processing the programmer's dynamic code specifications and producing executable code. We outline the mechanics of this process, and pay special attention to some partial evaluation mechanisms employed by the code generating functions.

Generating dynamic code. Dynamic compilation (or instantiation) for 'C is initiated by invoking the compile special form on a cspec. Compile then invokes 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 the previous 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 choosing the address of the beginning of the dynamic code randomly modulo the cache size. In the case when multiple dynamic functions are generated and used together, it would not be very expensive to track the placement of dynamic code to attempt to improve cache performance.

Dynamic compilation performs the composition of cspecs (inclusion of cspec b into the body of cspec a) into one final piece of code. This 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 the calling CGF.

The problem of generating efficient code from composed cspecs is analogous to function inlining and inter-procedural optimization when all function calls occur through pointers. 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 back end addresses this issue by building up an intermediate representation and performing some analyses prior to generating executable code. The VCODE back end, by contrast, optimizes for code generation speed: it generates code in just one pass, but can make poor spill decisions when there is significant register pressure. We describe both these systems in detail in Section 5.

Automatic dynamic partial evaluation. Partial evaluation is the key optimization that makes dynamic compilation profitable. tcc applies partial evaluation in three main ways. First, it folds run-time constants at instantiation time. The code generating functions contain code to evaluate any parts of an expression consisting of static and run-time constants. The dynamically emitted instructions can then encode these values as immediates.

Similarly, tcc performs strength reduction based on run-time constants: if an operand of an expensive operation (*e.g.*, multiplication or division) is a run-time constant, the CGF contains a fancier code-generation macro than usual: rather than emitting a fixed sequence of instructions, it first checks the value of its immediate operand, emitting different machine instructions at instantiation time depending on the value of this argument.

Lastly, the code generating functions automatically perform some dynamic loop unrolling and dead code elimination based on run-time constants. For example, if the test of a loop or conditional is run-time invariant, or if a loop is bounded by run-time invariants, then the actual control flow can be performed only once, at instantiation time, producing straight-line code and often leading to dead code elimination. 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 (for any given iteration of the nested loop). Our current implementation does not propagate run-time constant information to discover additional run-time constants in a fully general manner, as done in [1]. This restriction is mostly an engineering problem, since there is little framework for performing relaxation analyses at static compile time within lcc.

As an example of these optimizations, consider writing code that computes the dot-product of a vector col with a run-time constant vector row. In this case, it is possible to generate straight-line code in which the contents of row are hard-wired into the instruction stream. One way to do this is to use 'C's facilities for code composition:

```
void cspec code;
int cspec sum = '0;
for (k = 0; k < n; k++)
  if (row[k])
     sum = '(sum + col[$k]*$row[k]);
code = '{ return sum; }; /* sum is "a*b+c*d+..." */
```

Alternatively, one can leverage the dynamic loop unrolling provided by tcc:

```
'{ int k, sum = 0;
   for (k = 0; k < $n; k++)
      if ($row[k])
         sum = sum + col[k]*$row[k];
   return sum;
};
```

k is bounded by run-time constants and is never assigned outside of the control expressions of the for-statement, so it becomes a *derived* run-time constant. The resulting optimized code generating function appears below:

```
cgf(closure_t c) {
  int k;
  _tc_vspec_t itmp0 = _tc_local (I_I); /* int temporary */
  for (k = 0; k < c->rtc1; k++) {
     if ((*((int*)c->rtc2+k))) { /* skip if row[k]==0 */
        addpi(tmp0, c->col, 4*k);
        ldi(tmp0, zero, tmp0);
        mulii(tmp0, tmp0, (*((int*)c->rtc3+k)));
        addi(c->sum, c->sum, tmp0);
     }}}
```

In this case, the loop overhead is paid only once, at dynamic compile time. The resulting dynamic code has fewer instructions than the original, no branches, and no loop induction variable. Unless it is made too large, and hence acquires poor memory locality and incurs a high code generation cost, it will easily outperform its static counterpart.

This style of optimization, hard-coded at static compile time and then performed dynamically, helps to produce better code without incurring high dynamic compilation overhead. The code transformations are encoded in the CGF and depend on no run-time data structures. Furthermore, dynamic code that is unreachable due to a run-time constant need never be generated, sometimes even resulting in faster code generation than in the unoptimized case.

5 Dynamic Back Ends

The tension between dynamic code generation speed and code quality has led us to build two dynamic back ends for tcc: ICODE builds up an intermediate representation at run time before emitting executable code, whereas VCODE emits code directly, without any form of analysis. The main motivation behind this difference is inter-cspec optimization, particularly register allocation. Dynamic register allocation in 'C is a complex problem, equivalent to efficient inter-procedural register allocation when function calls are via pointers. Without an intermediate representation, register allocation lacks any notion of how tick-expressions are dynamically composed and so can only use information in a single tick-expression; using an intermediate representation, tcc is able to perform effective inter-cspec register allocation.

These two methods are provided because the appropriate level of run-time optimization is application-specific, since it depends on the number of times the code will be used (*i.e.*, it must be used enough to amortize the cost of run-time optimization) and on the size and structure of the dynamic code (*e.g.*, loop nesting depth, number of nested cspecs, etc.). To account for this, tcc allows the user to select the dynamic back end. This section explores the implementation details of both back ends.

5.1 VCODE-based code generation

When optimizing for code generation speed, the code generating functions use VCODE macros to emit code in one pass. Register allocation in this case 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 pressure is not data dependent (*i.e.*, it is known in advance), 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. As mentioned, however, efficient inter-cspec register allocation is hard, and unsurprisingly, the placement of these register management operations can greatly affect code quality. Consider the statement { s = '1; }, followed by a loop containing {s = '(x+s); } (where x is some free or dynamic variable). 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 two registers. With a naive dynamic register allocation strategy, 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.

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 which span cspec references, and hence the number of registers allocated by the CGF of one

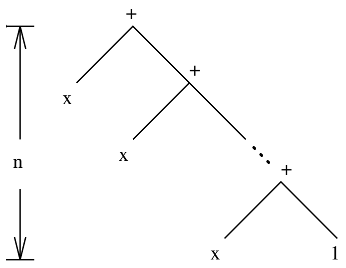

Figure 2: Register allocation problem.

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 referenced cspec allocates the register for storing its result, and simply returns this register name to the CGF for the enclosing cspec.

Obtaining and freeing a register are relatively inexpensive operations. Furthermore, tcc reduces the number of run-time register allocations that occur by "reserving" 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, dynamic register allocation using the VCODE dynamic back end is only slightly slower than in template-based systems, where it can occur entirely statically. Since most VCODE macros simply perform bit manipulations (shifts, ors, ands) on their arguments (constants and physical register names) and write the resulting machine instruction to memory, VCODE achieves an amortized code generation cost of ten instructions per generated instruction.

Clearly, however, the emitted code is good only in cases when few variables need to be spilled. 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.

The quality of emitted code can be improved without sacrificing the performance of VCODE very much by extending the code generating functions to perform two passes. On the first pass, no code is emitted: each CGF simply returns to its calling CGF (if any) the number of registers that it and its children (if any) require to avoid poor spilling. On the second pass, the CGFs and the VCODE macros they contain are executed almost as usual, generating code in one pass. However, if immediately prior to a call to a nested CGF the number of available registers is less than that previously requested by the nested CGF, the calling CGF spills and reloads registers as necessary. This is a promising technique, since it gives the CGFs a more global view of register allocation (*e.g.*, potentially pulling spills out of loop bodies) without incurring much cost (one traversal of the CGF call tree). However, at the time of this writing we have not finished implementing and evaluating it.

5.2 ICODE-based code generation

In cases where dynamically generated code is used frequently or runs for a long time, it may be preferable to trade additional dynamic compilation time for improved code quality. In this situation, tcc's dynamic back end emits code-generating functions which contain ICODE (rather than VCODE) macros.

ICODE was inspired by VCODE and the difficulty of using the getreg/putreg mechanism to effectively allocate registers in the presence of multiple code-generating functions. 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 variable use without expensive analysis: changes in expected usage frequency due to conditionals or loops can be expressed explicitly by the ICODE client.

Functionally, ICODE differs from VCODE in that it builds and consumes an intermediate representation at run time rather than immediately translating instructions to machine code. This 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. By having access to the entire body of dynamic code resulting from cspec-composition, ICODE can perform good register allocation and global optimization. After calling the last code macro, the ICODE client can invoke ICODE run-time library functions to perform various forms of register allocation and code optimizations prior to generating executable code.

When compile is invoked in ICODE mode, ICODE builds a flow graph, identifies live ranges, employs a linear-time algorithm to perform register allocation, and performs some peephole optimizations. Finally, it translates the intermediate representation to the target machine's binary format. We have attempted to minimize the cost of each of these operations. We briefly discuss each of these functions in turn.

Building a flow graph. ICODE builds a flow graph in one pass after all CGFs have been invoked, and hence after all the ICODE macros have been executed to lay out the intermediate representation in memory. 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. ICODE has full information about control flow at indirect jumps. In addition to constructing control flow information, ICODE collects a minimal amount of local data flow information (def and use sets for each basic block). All memory management occurs through arenas [20], ensuring low amortized cost for memory allocation and essentially free deallocation.

Finding live intervals. In the interest of fast code generation, ICODE does not compute precise live range information, but instead uses a coarse approximation that we call *live intervals*. An *interval* $[i, j]$ of instructions is simply all the instructions between the ith and jth instructions in the instruction stream, inclusive. Then a *live interval* of a variable v is the interval $[m, n]$, where m is the first instruction at which v is ever live, and n is the last instruction at which it is ever live. This interval information is only an approximation of the real live range information (in which ranges may be split): there may be large portions of $[m, n]$ in which v is not live, but we simply ignore them. In practice this has not been a problem: the quality of register allocation is quite good. Importantly, this scheme is quite efficient: given live variable information, creating a list of live intervals sorted by start or end point is accomplished in one pass over the code, and register allocation (described below) simply requires one pass over the interval list. There may also be room for improvement. We currently use a traditional relaxation algorithm for computing exact live variable information; since much of this information is lost when using live intervals, it may be possible to perform a much less expensive approximate live variable analysis. We are currently studying ways of doing this.

```
GREEDYREGISTERALLOCATION
    active ← {}
    foreach live interval i, from last to first
        EXPIREOLDINTERVALS(i)
        % R is the available number of registers
        if length(active) = R then
            r ← SPILLLONGESTINTERVAL(i)
        else
            r ← a register from pool of free registers
        if r is a valid register then
            register[i] ← r
            add i to active, sorted by start point
        else
            location[i] ← new stack location

EXPIREOLDINTERVALS(i)
    foreach interval j in active, from last to first
        if startpoint[j] ≤ endpoint[i] then
            return
        remove j from active
        put register[j] into pool of free registers

SPILLLONGESTINTERVAL(i)
    foreach interval j in active, from first to last
        if endpoint[j] ≥ endpoint[i] then
            break
    if startpoint[j] < startpoint[i] then
        r ← register[j]
        location[j] ← new stack location
        remove j from active
        return r
    else
        return null
```

Figure 3: Register allocation in one scan.

Fast "linear-scan" register allocation. Given a set of live intervals, our global register allocation algorithm is simple and fast. Variants have been considered in the literature [22, 24, 27] in the context of local register allocation and for spill-code minimization within a single basic block [42].

Given R available registers and a list of live intervals, allocating registers so as to minimize the number of spilled intervals involves removing the smallest number of live intervals so that no more than R live intervals overlap any one instruction. Since the number of overlapping intervals changes only at the start and end points of intervals, and the intervals appear in a list sorted in order of increasing end point, the algorithm traverses the list of intervals in reverse order, "jumping" from end point to end point while maintaining a list, *active*, of intervals live at the current point. When the number of these intervals exceeds R, the longest interval (the one with the earliest start point) is spilled. The active list is maintained in order of increasing start point. As a result, spilling the longest interval simply means removing the first element, and expiring intervals that are no longer active just involves a short search backwards from the end of the list. The details of the algorithm appear in Figure 3.

The length of *active* is bounded by R. As a result, given the live interval information from the previous section, the asymptotic running time of the algorithm is $O(I \cdot R)$, where I is the total number of live intervals (in general, one per variable).

In addition to this register allocator, we also provide a Chaitin-style graph-coloring register allocator [6]. This register allocation technique is not new: it has been studied and optimized extensively [5, 8, 25, 34], performs well in many cases, and is simple to implement. As a result, it is a good means of evaluating our simpler and faster register allocation algorithm.

Emitting code. The final phase of code generation involves translating the register-allocated, optimized ICODE instructions to the host machine's binary format. The code emitter simply makes one pass through the buffer of ICODE instructions. For each ICODE instruction, it invokes the VCODE macro corresponding to the given instruction, prepending and appending spill code as necessary, and performing some peephole optimizations and strength reduction.

The main problem with this scheme is that the code emitter can be quite large. ICODE has several hundred instructions (the cross product of operation kinds and operand types), and the code to translate and peephole-optimize each instruction is on the order of 100 instructions, about half of which are executed on a typical run. Always emitting a code generator for the full ICODE instruction set therefore generates unduly large executables, especially considering that most 'C programs use a small subset of all ICODE instructions. tcc therefore keeps track of the ICODE instructions used by an application, and automatically creates a customized ICODE back end containing code to only translate the required instructions. The compiler encodes the ICODE usage information for a given 'C source file in dummy symbol names in the corresponding object file. A prelinking 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 cuts the size of the ICODE library by up to an order of magnitude for most programs, reducing them to approximately the size of equivalent C programs.

6 Benchmarks

This section evaluates the tcc compiler. We outline our experimental methodology, describe the benchmarks we have used, and then present and discuss our performance data.

6.1 Experimental Methodology

Each of our benchmarks was written both in 'C and in static C. The 'C programs were compiled both with the VCODE and the ICODE-based tcc backends. The static C programs were compiled both with the lcc compiler and with the GNU C compiler. 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 considered as the baseline to measure the impact of dynamic code generation. Measurements collected using the GNU C compiler serve to compare tcc to an optimizing compiler of reasonable quality. We are working on improving lcc's static code generation, since optimizing the IR for dynamic code before emitting the code-generating functions will clearly result in dynamic code of significantly superior quality.

Times were derived by measuring a large number of trials (enough to provide several seconds worth of granularity, with negligible standard deviations) on a lightly-loaded SparcStation 5 using the Unix getrusage system call. These 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 (*i.e.*, in tight inner loops).

Benchmark	VCODE	ICODE
One large cspec, dynamic locals	96.8	1019.7
One large cspec, free variables	134.0	1142.6
Many small cspecs, dynamic locals	178.1	579.8
Many small cspecs, free variables	260.1	1261.9

Table 1: Code generation overhead, cycles per generated instruction.

For the 'C versions of the benchmarks, we separate the cost of dynamic code generation from the run time of the dynamic code. This separation allows us to calculate the "cross-over" point at which dynamic code generation becomes profitable. We further break down the cost of dynamic compilation, in units of processor cycles (our test machine runs at 70MHz) per generated instruction. In the case of VCODE, this cost consists of manipulating closures and other meta-data, and actually generating binary code. For ICODE, we also measure two additional phases: building the intermediate representation, and allocating registers. We present register allocation costs for both the linear scan algorithm described in this paper and for the Chaitin-style register allocator which we used as a baseline.

Several of the benchmarks are data-dependent (usually increasing in run time and, sometimes, size of dynamic code, as the size of the input data increases). Due to space constraints, we picked a single reasonable input size. Specifics about each benchmark appear in Section 6.2.

We also compare the cost of our two different dynamic code generation systems (ICODE and VCODE) in two situations which we consider significant extremes of dynamic code style: a very large tick-expression (approximately 1000 instructions) compiled alone, and a very small tick-expression (one cspec composition and one addition) composed many times with other tick-expressions (in our measurements, it is composed 100 times with itself). For both of these cases, we wrote two versions of code, one accessing free variables in the containing function's scope, and the other making use of dynamic locals. Much cspec composition and many free variables both exacerbate the cost of manipulating closures. The results appear in Table 1. Predictably, ICODE is approximately an order of magnitude slower than VCODE, due to the overhead of manipulating an intermediate representation and then translating this to binary code. Section 6.3, however, illustrates that this extra cost is often amortized by superior dynamic code quality.

Our evaluation has not been "SPECmark-driven": we have not tweaked tcc in any way to make individual benchmarks run faster. Complete data from these experiments appears in Section 6.3, below.

6.2 Benchmarks

This subsection describes the benchmarks we used to evaluate the tcc compiler. They have been chosen to highlight different styles of dynamic code generation use. Many of them are fully described (but not measured) in [15]. We also describe modifications made to xv [4], a relatively sophisticated share-ware image manipulation package, to exploit dynamic code generation, and the resulting performance benefits.

Run-time constants. Dynamic code generation can be used to hardwire infrequently changing values into the instruction stream. This optimization is beneficial because the values need not be loaded from memory, and expensive operations (such as multiplication and division) that use these values can be strength-reduced. It is increasingly profitable on modern architectures, where cache misses are very expensive and division and multiplication are frequently provided only in software.

An example is 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. Such a hash function can be faster than an equivalent C version, since a ‘C compiler can exploit run-time constants to scatter and normalize key values both by hard-coding them into the instruction stream and by strength-reducing the multiplication and division. The hash experiment measures the time to repeatedly look up two values in a hash table; the first value is in the table, the second is not. No bucket has more than one element. If the hash table were rarely modified, ‘C could also be used to dynamically construct a perfect (or near-perfect) hash function at run time, possibly further improving performance.

A second benchmark we provide for this case is ms, in which we repeatedly scale a 100x100 matrix of integers by a run-time constant.

Parameterized functions. Many library routines are parameterized via function pointers. Examples include the standard C library quicksort and heapsort routines, and many mathematical library routines. Unfortunately, indirect function calls eliminate many potential optimizations, since the function cannot be integrated with the library code. Since cspecs can be composed with each other arbitrarily, ‘C could be used to parameterize library functions with cspecs rather than function pointers, potentially leading to large gains in efficiency.

We provide two benchmarks that fit in this class. The first is a simple heapsort function, heap, that is parameterized with a code fragment to swap the contents of two memory regions of arbitrary size. It considerably outperforms static versions by specializing itself with respect to the size of the array elements that it sorts. Our experiment measures the time to heapsort a 500-entry array of 12-byte structures. The static code copies the structures using memcpy.

The second example is a Newton-Raphson root solver, ntn. The function and its derivative are provided by code fragments that are, again, incorporated into the function at run time. The experiment computes the root of the function $f(x) = (x+1)^2$ to a tolerance of 10^{-9}. The static code uses Newton's method parameterized via two functions: one to compute f, the other to compute f'.

Function composition. Similarly, ‘C allows modular function *composition*: composed code specifications can be integrated together by tcc into straight-line code. This functionality is analogous to being able to dynamically inline the code referenced by arbitrary function pointers.

Networking code is one important application of this type of composition. The networking community has long aimed to modularly compose protocol layers [9]. Each protocol layer frequently involves data manipulation operations (*e.g.*, checksumming, byte-swapping, etc.). 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 [9]. This modular composition of data operations is an active research area. The two main limitations of current approaches to this problem are that they use specialized languages and, with the exception of work using VCODE [14], they are static, in that passes cannot be built at run time. A more powerful approach is to use ‘C to compose functions dynamically; programmers can use a language they are accustomed to, and data manipulation steps can be flexibly composed at run time.

This benchmark is called cmp. The experiment measures the time to copy a 4096-byte message buffer while computing both a checksum and a byteswap operation. The static code performs these operations using function pointers while the ‘C code represents checksum and byteswap as code specifications that are dynamically incorporated into the data-copying loop.

Small language compilation. Many small, primitive languages are both time-critical and amenable to dynamic compilation. The query languages used to interrogate data bases are well-known targets for dynamic code generation [30]; since databases are large, dynamically compiled queries will be applied many times.

We have developed a small query language and benchmarked a dynamic query compiler for it, query. Each query is a boolean expression formed by accessing record fields and comparing them to other fields or to constant values. The experiment performs a query on a database with 2000 entries and selects those entries matching a query expression composed of five binary comparisons. The static code interprets queries using a pair of switch statements, while the ‘C version dynamically compiles the query to machine code.

Dynamic function call construction. ‘C allows programmers to generate functions and calls with statically unknown numbers and types of arguments. This is a powerful feature. For instance, it allows the construction of code to marshal and unmarshal arguments stored in a byte vector, operations frequently performed to support remote procedure call [3]. By generating specialized code for the most active functions it is possible to gain substantial performance benefits [44]. Our two benchmarks, mshl and umshl, dynamically generate marshaling and unmarshaling code, respectively, given a printf-style format string specifying the types of arguments. This ability goes beyond mere performance: ANSI C simply does not provide mechanisms for dynamically constructing function calls with varying numbers of arguments.

The mshl experiment measures the time required to construct and run a function that takes five arguments and marshals them into a byte vector. It is not possible to write equivalent C code in ANSI C. If the interface is exposed to the client, it is possible to crudely emulate this functionality using C's varargs facilities, but not without requiring that the caller always provide information about the number and types of arguments.

The umshl experiment measures the time to unmarshal a byte vector and call a function taking five arguments. It is impossible to write code that performs an equivalent function in ANSI C, since doing so requires the ability to generate function calls with varying number of arguments. Therefore, to give some feel for the speed of the generated code, we compare to statically compiled C code that handles the specific case of five arguments.

Dynamic partial evaluation. Partial evaluation specializes a function with respect to some number of arguments. ‘C can be used to implement partial evaluation in the context of C. An example from computer graphics ([13]) is partial evaluation of the exponentiation function with respect to a given exponent. This is very useful, because it reduces the exponentiation algorithm to a minimum number of multiplication and squaring operations. The benchmark pow dynamically generates a specialized function that raises its argument to the power 13. The static version uses a general integer power function.

Code construction. ‘C can be used to perform unusual code construction tasks, such as creating "executable data structures." The benchmark binary takes a sorted integer array as input and creates code that implements a binary search on that array. The values from the array are hardwired into the instruction stream, and the minimum number of conditionals and jumps are performed during the search. Thus, lookup into the array involves neither memory loads nor looping overhead: the code is a series of nested if statements that compare the value to be found to constants. As a result, the dynamically constructed code is an order of magnitude more efficient that its static counterpart. The benchmark measures the time to repeatedly look up two entries (one present, one not) in a 16-entry array.

Putting it all together. To validate our system on a larger example, we have modified xv, a large and popular image manipulation package. Specifically, we picked one of its image processing algorithms (we deemed this sufficient, since most of the algorithms are implemented very similarly) and changed it to make use of run-time information. The algorithm, *Blur*, applies a convolution matrix of user-specified size and consisting of all 1's to the source image. The original algorithm was implemented very efficiently: since the values in the convolution matrix are known statically to be all 1's, 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. By unrolling this loop and exploiting the run-time constant checks, tcc produces code that runs in 1.08 seconds (on a 640x480 image and 3x3 convolution matrix), with a dynamic compilation time of only 0.01 seconds (using our more expensive dynamic back end, ICODE). By contrast, code generated by lcc runs in 1.96 seconds, almost two times more slowly. Code emitted by GNU CC with all optimizations turned on runs in 1.04 seconds. All results are best of 5 runs, on an unloaded SparcStation 5. As mentioned earlier, the basis for comparison of the performance of dynamic code should be lcc, since the dynamic back ends are generated by an lcc-style back end, without further static optimizations. xv is an example of the profitability of dynamic code generation in the context of a well-known application program. We are confident that additional work on static optimization of the code-generating functions will make tcc's dynamic code superior even to that of aggressive optimizing compilers, without penalizing dynamic compilation times.

Other uses. The benefits of dynamic code generation extend beyond efficiency. For instance, 'C-enabled currying can be used to associate functions with state that is not visible to the caller. This technique is implemented by dynamically generating a wrapper function that calls the original function with internally bound state, thereby providing information hiding while allowing functions to be parameterized with data.

6.3 Results

This section discusses the performance of dynamic code versus static code measured for our benchmarks, as well as the break down of dynamic compilation costs.

In both Figure 4 and Figure 5, 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-gcc compares dynamic code created with VCODE to static code compiled with GNU CC, and so forth.

Figure 4 illustrates the ratio of run time of static code to run time of dynamic code: a ratio greater than one means that dynamic code generation is profitable. In general, this ratio is considerably greater than one: in some cases, dynamically generated code is up to an order of magnitude faster than static code. In three cases, however, dynamic code generation does not pay off. In the case of umshl, as mentioned, we are providing dynamically a functionality that does not really exist in C: the static code we created specifically for this example is very well tuned. In the case of hash and ms, ICODE provides some advantage over static code (specifically, in the case of ms, we see a six-fold speed-up), but the code generated with VCODE is slower. We attribute this to the fact that VCODE does not perform any significant optimization other than strength-reduction of run-time constants.

Figure 5 gives an indication of the relative costs of dynamic code generation. The "cross-over" point on the vertical axis is the number of times that a piece of dynamic code must be executed in order for the sum of the cost of its invocations and its compilation to be less than or equal to the cost of the same number of invocations of static code. This is a measure of how quickly dynamic code "pays for itself." In 3 cases (umshl, and hash and ms using VCODE) there are no vertical bars: this is because 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 invocations. In some cases (ms using ICODE, cmp, and query) the dynamic code even pays for itself after only one run.

These graphs confirm the central (and predictable) tradeoff inherent to the design of tcc. Improvements in dynamic code quality come at the cost of additional dynamic compilation time. Using VCODE to perform fast, in-place dynamic code generation is very cheap, and can thus often be effective when the dynamic code is not used very much. But there are some cases when the code quality is not sufficiently good; ICODE then provides a good alternative, even though its code-generating costs can be up to an order of magnitude larger than those of VCODE. Interestingly, incurring the extra cost of code optimization can sometimes improve code quality sufficiently to make dynamic compilation pay off sooner than in the unoptimized case. This occurs for the ntn benchmark, in the icode-gcc and vcode-gcc cases. Although the VCODE back end generates code six times more quickly than ICODE, the dynamic code generated by ICODE pays for itself in approximately two thirds the number of runs required for the VCODE-generated code to pay off (see Figure 5).

Figures 6 and 7 analyze these code generation costs in more detail. Figure 6 shows that the VCODE back end generates code at between 100 and 500 cycles per generated instruction. The cost of manipulating closures and other meta-data is negligible: almost all the time is spent actually emitting binary code. Figure 7 presents similar breakdowns in the case of ICODE. For each benchmark, the left column displays code generation costs when using the linear scan register allocation, and the right column displays costs when using graph coloring register allocation. The ICODE back end generates code at a speed between approximately 1000 and 2500 cycles per generated instruction. The costs of manipulating closures and building the intermediate representation are small, as is the time spent translating the ICODE opcodes to machine code. Approximately 70-80% of the ICODE code generation cost is due to register allocation and related operations, such as computing live variables and building live ranges. The linear scan register allocation algorithm outperforms the graph coloring allocator in all cases but one, sometimes by up to a factor of two (in the case of dp). The performance of the two allocators depends very much on the structure of the code. When the code contains many variables (as is the case, for example with dp), scanning live ranges is superior to graph coloring. By contrast, when there is a lot of code but very few variables (as in binary, the only benchmark where the linear scan performs more poorly than graph coloring), it is cheaper to color the (small) interference graph than to set up the live ranges for the linear scan.

7 Conclusion

Dynamic code generation is a powerful and useful technique which has not been widely exploited so far because of inadequate language and compiler support.

This paper has described and evaluated tcc, the first full implementation of 'C. 'C is a superset of ANSI C, designed to expose the dynamic code generation process to the programmer at the level of C expressions and statements. Unlike previous systems for dynamic code generation, it gives the programmer full control of the code creation process while remaining expressive and portable.

tcc is a solid and valuable tool for exploiting dynamic code generation in day-to-day programming and exploring techniques and tradeoffs of dynamic code generation itself. A release of the software, which currently runs on MIPS and SPARC processors, is available.

Measurements on sample code from a variety of application areas reveal that an efficient and easy-to-use implementation of a dynamic code generation system can provide significant performance improvements. We have reported measurements of running times for both dynamically generated code and equivalent static C code, and have compared these running times to those of code compiled by a widely-used optimizing compiler, GNU CC. Performance generally improved, increasing by up to an order of magnitude in the best cases. For several applications, the cost of dynamic code generation was amortized after just one execution of the dynamically generated code.

Acknowledgments

We thank Vivek Sarkar and Eddie Kohler for their help and insights regarding register allocation, and Jonathan Litt for his patience in rewriting parts of xv in 'C, helping us find several compiler bugs along the way.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, PA, May 1996.

[2] 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 Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] J. Bradley. xv-3.10. `ftp://ftp.cis.upenn.edu`.

[5] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991.

[6] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):201–207, June 1982.

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

[8] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984.

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

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

[11] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, FL, January 1996.

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

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

[14] D. R. Engler. 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, May 1996. `http://www.pdos.lcs.mit.edu/~engler/vcode.html`.

[15] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg, FL, 1995.

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

[17] D. R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. *Proceedings of ACM SIGCOMM'96 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, pages 53–59, August 1996.

[18] D. R. Engler and M. Poletto. A 'C tutorial. Technical Memo MIT-LCS-TM-564, MIT Laboratory for Computer Science, March 1997.

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

[20] G. E. Forsythe. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[21] C. Fraser. copt. `ftp://ftp.cs.princeton.edu/pub/lcc/contrib/copt.shar`.

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

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

[24] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, November 1974.

[25] R. Gupta, Mary Lou Soffa, and T. Steele. Register allocation via clique separators. *SIGPLAN Notices*, 24(7):264–274, June 1989.

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

[27] W-C. Hsu, Charles N. Fischer, and J. R. Goodman. On the minimization of loads and stores in local register allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[28] D. Keppel. A portable interface for on-the-fly instruction space modification. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–95, April 1991.

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

[30] D. Keppel, S.J. Eggers, and R.R. Henry. Evaluating runtime-compiled value-specific optimizations. TR 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[31] 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.

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

[33] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. In *The Journal of Supercomputing*, volume 7, pages 51–142, 1993.

[34] G. Lueh, T. Gross, and A. Adl-Tabatabai. Global register allocation based on graph fusion. Technical Report CMU-CS-96-106, Carnegie Mellon University, March 1996.

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

[36] 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.

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

[38] T. A. Proebsting. Simple and efficient BURS table generation. *SIGPLAN Notices*, 27(7):331–340, June 1992.

[39] 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.

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

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

[42] V. Sarkar. Personal communcation, September 1996.

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

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

[45] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6), June 1968.

[46] 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.

[47] L. Wall. *The Perl Programming Language*. Prentice Hall Software Series, 1994.

[48] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. *Proceedings of ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.