DCG: An Efficient, Retargetable Dynamic Code Generation System

Dawson R. Engler*
Massachusetts Institute of Technology

Todd A. Proebsting†
University of Arizona

Abstract

Dynamic code generation allows aggressive optimization through the use of runtime information. Previous systems typically relied on ad hoc code generators that were not designed for retargetability, and did not shield the client from machine-specific details. We present a system, `dcg`, that allows clients to specify dynamically generated code in a machine-independent manner. Our one-pass code generator is easily retargeted and extremely efficient (code generation costs approximately 350 instructions per generated instruction). Experiments show that dynamic code generation increases some application speeds by over an order of magnitude.

1 Introduction

Dynamic code generation is the creation of executable code by an executing process. Unlike *self-modifying* code, dynamic code generation does not change the existing code, but rather augments it. Dynamic code generation enables programs to create specialized instruction sequences based on runtime information. For instance, the statically generated code to scale a vector by a runtime-determined value must rely on a general multiply routine or instruction. A program benefiting from dynamic code generation could, however, produce executable code optimized for the given scale factor (e.g., since it is a runtime constant, multiplication can be eliminated through strength reduction to shifts and adds). Our experiments indicate that dynamic code generation can increase the speed of dividing an integer matrix by a runtime determined constant by a factor of 10 and integer matrix multiplication by a factor of 4.

Programmers who wish to exploit dynamic code generation face many difficulties. Because binary instructions are generated, programs using dynamic code generation must be retargeted for each machine — a potentially substantial programming effort. Differing memory subsystems (e.g., split I/D caches) present additional retargeting difficulties because code is generated in data space and then executed in instruction space. Furthermore, dynamic code generation must be efficient since the code generation time will be incurred by the executing program.

Our dynamic code generation system, `dcg`, portably and efficiently generates executable code at runtime. `dcg` client programs specify dynamically generated code using the compact, machine-independent intermediate representation (IR) of the `lcc` compiler [8]. Binary code is selected using BURS tree pattern-matching technology [17, 9]. The code generator is very efficient — the creation and translation of IR to binary instructions takes approximately 350 instructions per generated instruction.

There are two contributions of this paper: (1) a demonstration of efficient dynamic machine code generation from a machine-independent specification, and (2), the use of existing compiler technology to reduce the problem of building a dynamic code generation system to that of implementing a retargetable compiler backend.

To our knowledge `dcg` is the only stand-alone retargetable dynamic code generator to emit binary instructions directly. We have automated code generator retargeting by developing simple machine spec-

*Address: M.I.T. Laboratory of Computer Science, 545 Technology Square, Cambridge, MA 02139. Internet: `engler@lcs.mit.edu`

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

ification languages and preprocessors. The retarget of the system to the MIPS R3000 from the SPARC took approximately 1 week. The current system runs on ABI compliant SPARC implementations (e.g., SPARC1, SPARC10, IPX) [11] and the MIPS R2000/R3000 series [12]. We describe the design and implementation of our system, and report preliminary tests of its efficiency.

2 Previous Work

Many people have used dynamic code generation to exploit runtime data for creating highly efficient code that could not have been produced statically. In [14], Keppel, Eggers and Henry survey many advantageous uses for dynamic code generation.

Massalin and Pu used dynamic code generation in their Synthesis Kernel to remove a layer of interpretation from operating system routines [19]. Dynamic code generation made a single byte read/write 56 times faster and a paged-sized read/write 4 to 6 times faster in the Synthesis Kernel than in SunOS even though SunOS was running on a faster machine. Unfortunately, their system was not easily retargetable and ran only on the Motorola 680x0 family.

Implementations of languages that rely on dynamic type information benefit from this technology as well. Smalltalk [5] and Self [3], for example, have both used dynamic code generation to optimize frequently executed routines.

ParcPlace sells an implementation of of Smalltalk-80 that uses a dynamic code generator for SPARC, Motorola 68k and PowerPC, Intel x86, and other architectures. Unlike `dcg`, this system is designed specifically for the compilation of Smalltalk-80, and not as a stand-alone system for dynamic code generation.

Leone and Lee describe a "lightweight" approach to dynamic code generation, called *deferred compilation*, that utilizes compile-time specialization to reduce run-time code generation costs [?]. Their approach relies on sophisticated compiler analysis of programs to create efficient, "hard-wired" code emitter routines. No mention is made of the system's retargetability.

Pike, Locanthi and Reiser exploited dynamic code generation to optimize `bitblt`, a bit-manipulation routine used in many windowing systems [16]. `bitblt` merges a source rectangle with a destination rectangle via logical bit operators. `bitblt` code to handle every possible case of bit boundaries on a word-oriented machine is slow because of its burdensome generality. Static enumeration of all cases would require over 1MB of code. Instead, they dynamically generate code for each case as needed. The dynamically generated code was up to an order of magnitude faster than the static code.

Keppel addressed some issues relevant to retargeting dynamic code generation in [13]. 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.

Many Unix systems provide utilities to dynamically *link* object files to an executing process. Thus, a retargetable dynamic code generation system could emit C code to a file, spawn a process to compile and assemble this code, and then dynamically link in the result. Preliminary tests on `gcc` indicate that the compile and assembly phases alone require approximately 30,000 cycles per instruction generated. Our system is two orders of magnitude faster than this.

Outside the context of dynamic code generation, retargetable code generation is well studied. Two competing code generation strategies dominate retargetable compilers: Register Transfer Language (RTL)-based rewriting rule systems [4], and tree pattern matching systems [1, 9, 6]. RTL rewriting is more general than tree pattern matching, but it is more complex and slower.

3 `dcg` Code Generator Design

The primary design goals of our dynamic code generation system, `dcg`, were simplicity and efficiency. `dcg` consists of a small, but complete, library of interface routines that provides extremely efficient dynamic code generation facilities to client programs. Furthermore, `dcg` is easy to retarget. (Of course, once `dcg` is retargeted to a new machine, *all* clients should then run unchanged.)

The unit of code generation for `dcg` is a single procedure. `dcg` compiles each procedure and returns a pointer to the executable code. The client invokes that code as an indirect call to a C procedure.

To make client programs portable, they specify code using a machine-independent intermediate representation (IR) that is passed to `dcg`. The logical infrastructure of `dcg` is taken directly from an existing retargetable ANSI C compiler, `lcc` [8]. `lcc`'s IR is smaller, simpler, and more easily understood than the obvious alternative, `gcc`'s. The simplicity and regularity of the IR is important because this IR must be easily generated by client programs. In essence, every client program is a small compiler front-end. `lcc`'s

IR consists of expression trees with a minimal symbol table for variables and data types. The abstract machine, while small, is complete, being sufficient for the construction of ANSI C compilers for both RISC and CISC machines. An important benefit of using an existing interface is testing. By retaining `lcc`'s interface faithfully, `dcg`'s code generator is able to link directly to `lcc`'s frontend; testing its correctness consists of simply compiling existing test-suites to assembly language, and testing the resultant output.

Code selection is done using `burg`, which uses Bottom-Up Rewrite System (BURS) technology to optimally translate an IR tree into machine instructions [9]. Instruction selection using dynamic programming and tree pattern matching is easily understood, automated, and quite fast.

`dcg` omits any significant global optimizations and pipeline scheduling. Existing pipeline schedulers would have made the code generator slower and more complex. Global optimizations are the responsibility of the client, which has access to the low-level IR specification. `dcg` is only responsible for emitting efficient code locally. `dcg` does include a machine-independent mechanism, however, to allow a small amount of global register allocation. A few registers are reserved as expression temporaries, and the rest are available for a function's local variables and arguments. The client declares an allocation priority ordering to `dcg` for all register candidates, and `dcg` allocates registers according to that ordering. Once the machine's registers are exhausted, all subsequent local variables are kept on the runtime stack. This simple, machine-independent technique is extremely fast, and still provides some register allocation control to the client.

Complete code generation includes tree construction and labeling, register allocation, instruction selection, jump resolution and binary code emission. BURS tree labeling occurs during tree construction. After a forest is passed to `dcg`, it consumes each tree in one pass. The matching tree patterns are traversed left-to-right — register allocation and instruction emission are done immediately. Currently, spills are not handled; we assume that all expressions can be evaluated with only eight temporary registers. When the forest has been consumed, unresolved jumps are backpatched.

4 Client/`dcg` Interface

Clients dynamically develop code one procedure at a time from a forest of IR trees. In addition, declarations of local variables and procedure arguments must be communicated to the code generator. Clients respect the code generation interface defined by `lcc` when invoking `dcg`. This interface is fully documented in [8].

`dcg` consists of library routines that simplify the creation of `lcc` IR nodes and typing information. Individual functions are provided for the construction of all legal IR nodes and correspond to `lcc`'s 109-operator language (36 operations with 9 potential types). Additional functions construct some common symbol types (e.g., constants, addresses and local variables). The names of interface procedures are lower-case versions of `lcc`'s operators: operators have a functional prefix (e.g., `ARG`, `MUL`, `DIV`, `CALL`) and a type suffix (e.g., `D`, `F`, `I`, `U`, `P`, `C`). For example, the function `negi` builds a tree that computes the integer negation of a subtree. Functions that return symbols instead of tree nodes have an `s` prefixed to their name (e.g., `slabelv` returns a symbol to a void label).

Trees are linked together in a forest, and the forest is passed to the function `dcg_gen` for code generation. `dcg_gen` returns a pointer to the generated code to the client program.

Figure 1 contains a simple example that builds a function of a single integer argument that returns the value of the argument plus 1. The client routine declares the single argument with `sargi`; register allocation is done using `dcg_param_alloc`. The procedure is specified by a single IR tree that is *registered* with `dcg`. When `dcg_gen` is called, code is generated. This code can then be invoked by an indirect call. Figure 2 is the code generated by `dcg` (currently, `dcg` always allocates an activation record).

Library routines are provided to make client programs simpler to write, while maintaining `lcc`'s code generation interface. The `dcg` library interface is machine independent — client programs do not need to be altered when linked with `dcg` routines on a new target machine.

5 Experimental Clients

We illustrate using `dcg` with two simple clients: a customizing matrix multiplication that synthesizes code tailored for each row of an input matrix, and an *interactive* small C-like language implementation that compiles and executes its code on the spot. Our experiments are conducted on a MIPS R3000 and a SPARC 10.

5.1 Matrix Multiplication

Fast multiplication of matrices is important to many graphics and image processing applications. Often

```
typedef int (*FPtr)(int);

FPtr example() {
        Symbol arg[2];  /* argument vec sent to gen */
        int ncalls = 0; /* number of calls made by plus1 */

        arg[0] = sargi();       /* allocate symbol for 'x' */
        dcg_param_alloc(arg, ncalls); /* associate with a virtual register (if possible) */

        /* create and register IR tree for "return x + 1;" with dcg */
        regtree( reti( addi( indiri(addrfp(arg[0])), cnsti(scnsti(1)))));

        /* generate code on heap */
        return (FPtr) dcg_gen(sfunc("plus1"), arg, ncalls);
}
```

Figure 1: Routine to Build Function "`int plus1(int x) { return x + 1; }`" Dynamically

```
        addiu $sp, -152  # allocate AR
        add $25, $4,  1  # ADDI ($4 holds argument 1)
        move $2, $25     # RETI ($2 holds return value)
        addiu $sp, 152
        j $31
```

Figure 2: The R3000 Code Emitted to Compute "`return x + 1;`"

these matrices have regular runtime determined characteristics (e.g., large numbers of zeros and small integers) that cannot be exploited by static compilation techniques. The use of dynamic code generation allows these characteristics to be exploited by allowing a client to craft locally optimized code based on the actual values. Because code for each row is specified once and then used n times (once for each column), the costs of code generation are easily recouped. In our example code, three optimizations are employed for integer matrix multiplication — one by the client, two by `dcg`. The client directly eliminates multiplication by zero. `dcg` encodes each value as an immediate value of an emitted arithmetic instruction, where possible. When profitable, `dcg` does strength reduction, replacing multiplication with shifts and adds.

The following example is provided to illustrate dynamic code generation techniques and `dcg`'s interface. (For efficiency the right-hand matrix 'b' has been transposed; matrix multiplication is done by computing the dot products of rows of each matrix.)

Consider the example 3x3 matrix:

$$\begin{matrix} 3 & 0 & 2 \\ 0 & 7 & 4 \\ 0 & 0 & 3 \end{matrix}$$

We want to emulate the following optimized C code that describes a dot-product customized for each row (for clarity, we elide the use of shifts and adds for strength reduction).

```
int dot_row0(int *b) { return 3*b[0]+2*b[2];}
int dot_row1(int *b) { return 7*b[1]+4*b[2];}
int dot_row2(int *b) { return 3*b[2]; }
```

The following code is emitted by `dcg` for the tree specifying `dot_row0`:

```
/* return 3 * b[0] + 2 * b[2]; */
addiu $sp, -152   # allocate AR
lw $24,  0($4)    # load value of b[0]
mul $25, $24, 3   # 3 * b[0]
lw $15,  8($4)    # load value of b[2]
mul $24, $15, 2   # 2 * b[2]
add $25, $25, $24 # add two results
move $2, $25      # put in return register
```

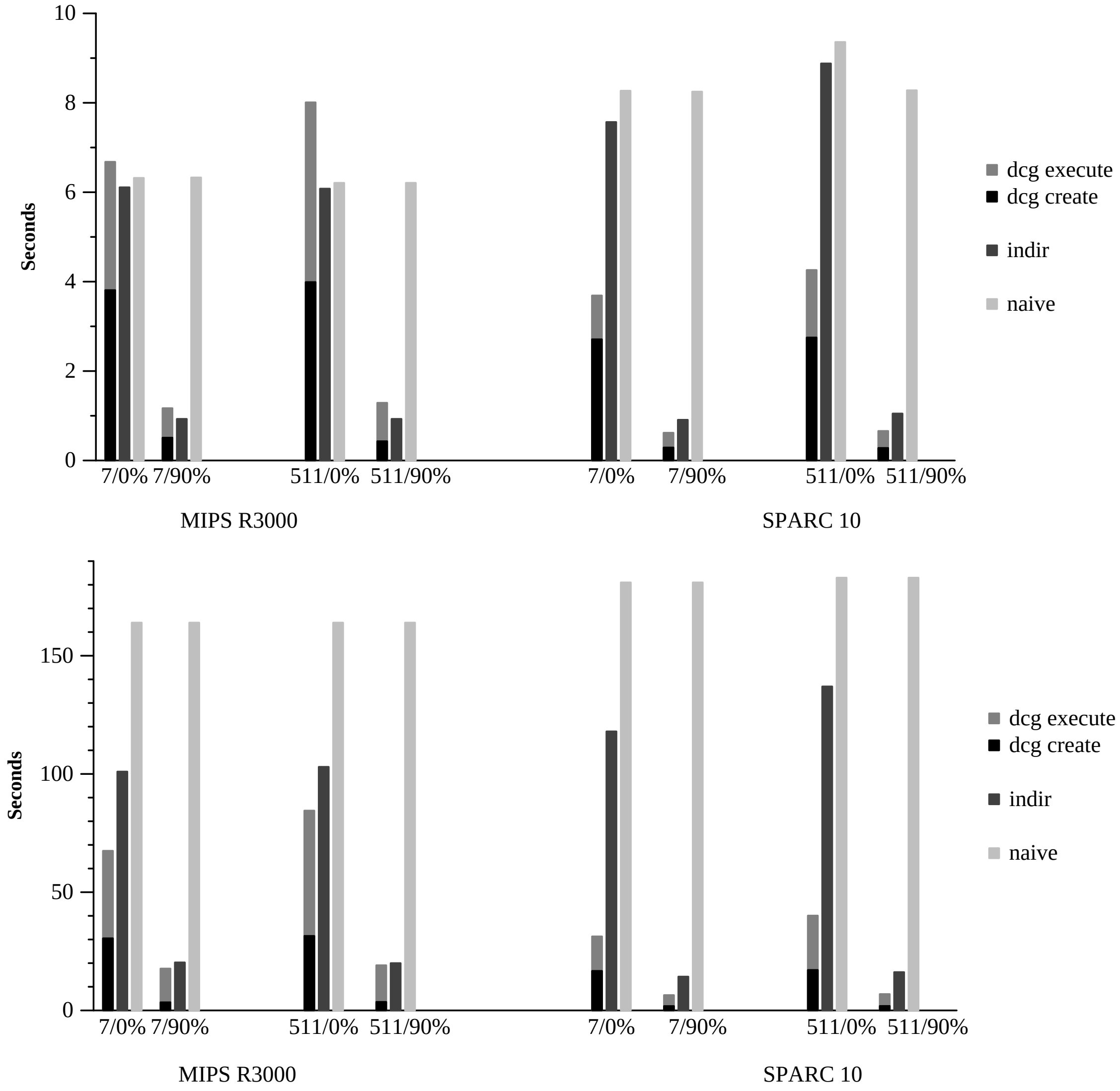

Figure 3: Matrix Multiplication times; `dcg create` refers to total code generation cost; x-axis labels are of the form: 'maximum element size/percentage of zero elements'.

```
addiu $sp, 152     # deallocate AR
j $31              # return
```

To test dynamic code generation in this setting, matrix multiplication was implemented using three algorithms. `naive`: A naive algorithm that does not take advantage of zeros. `indir`: A sophisticated scheme that uses indirection vectors to avoid multiplication by zeros. `dcg`: The dynamic code generation algorithm described above. (The C code for `indir` and `dcg` are given in Appendix A.) Figure 3 gives timings to compare the three implementations on both machines under various conditions for randomly generated matrices. We vary the maximum value held in the arrays to either 7 or 511, and choose the percentage of 0's in the arrays to be either 0% or 90%. The `dcg` timings are split into code generation times, given by "dcg create," and execution times, given by "dcg execute." As the timings indicate, dynamic code generation is almost always a win — often by a tremendous amount.

On the SPARC, `dcg`-generated code can be close to a factor of 4 faster than the optimized C implementation and almost 170 times faster than the naive one. Because the SPARC does integer multiply in software, using dynamically generated code is very profitable.

For small matrices, the MIPS implementation is slightly slower than the `indir` method because of the cost of code generation. The `dcg` generated code is always faster than the `indir` code, but there is not enough data over which to amortize the code generation costs. At larger sizes, `dcg` is profitable. Because a substantial portion of execution time for both the MIPS and SPARC dynamic code generation examples goes to code generation, overall execution time will improve dramatically with even modest increases in code generation speed.

To determine the efficiency of using `dcg`, we computed how many instructions the matrix multiply client executes for each instruction that is ultimately emitted on the R3000. Instruction counts were made using `pixie`. For matrix multiply, `dcg` routines executed approximately 350 instructions for every instruction emitted.

5.2 Interactive Tiny C Compiling Interpreter

Often, interpreters are more attractive than compilers (e.g., during debugging). Interpretation may, however, be too slow to be a practical option. We have implemented a simple, interactive compiler that reads in C-like functions, generates code for those functions dynamically, and then executes it. For recursive Fibonacci, the compiled version runs between 18-50 times faster than the interpreted version. The resultant code on a SPARC 10 executes within 9% as that of `gcc` using the highest level of optimization. Additionally, the compiling interpreter is 25% smaller than its non-compiling counterpart.

Our language, Tiny C, has only a single type (integer), supports most of C's relational and arithmetic operations on it (`/`, `-`, `<`, etc.), and provides `if` statements, `while` loops, and function calls as control constructs. Programs in Tiny C consist of global declarations, followed by function declarations, these declarations are terminated by the `begin` keyword, which starts an interactive session. Code is compiled as the user enters it.

A recursive Fibonacci program is used to measure the performance of three Tiny-C implementations.

`interp`: A simple interpreter that translates Tiny C to abstract syntax trees, which it then recursively evaluates.

`gcc`: The C code is statically compiled using gcc with optimization level "-O3". This is used to give an upper bound on the quality of local code.

`dcg`: The compiling interpreter discussed above. Tests compute the 30^{th} and 35^{th} Fibonacci numbers.

Figure 4 graphically summarizes the timings. Dynamic code generation clearly wins over interpreted code: on the SPARC, `dcg` generated code is approximately 53 times faster than interpreted code and very close to the best static code. On the MIPS it is approximately 20 times faster than an interpreted version and within a factor of two of the statically compiled code. The reason for the difference between `dcg` generated and `gcc` generated code on the MIPS is that Fibonacci is composed mostly of jumps and calls — actions which hurt `dcg` because it does no pipeline scheduling. The lack of a corresponding difference on the SPARC is a result of prolific register window dumping in response to Fibonacci's recursive nature, where `dcg`'s lack of pipeline scheduling is hidden in the overhead of bulk memory transfers.

5.3 Additional Experiments

We implemented two packet filter engines for Mogul's packet-filter language [15]. The first is an extremely efficient byte-code interpreter that uses indirect jumps (a C extension provided by the GNU C compiler) to achieve efficient interpretation. The second uses `dcg` to generate code specialized for a given filter and run it directly (eliminating interpretation overhead). Even though `dcg`'s generated code is fairly poor in this instance (packet filters utilize a number of control flow constructs, consequently `dcg`'s lack of pipeline scheduling hurts performance noticeably) the performance improvement is over a factor of 10. With straightforward compiler techniques, the generated code could be improved by 2-3 fold, yielding a relative performance improvement of 20-30 fold.

We also implemented an optimized matrix scaling library. Multiplication by a runtime constant is reduced to shifts and adds. Division is strength reduced to multiplication (and then to shifts and adds) using the techniques described in [10]. The performance of multiplying a 1024x1024 integer matrix by a runtime constant improved by a factor of 4 on a SPARC 10, and 40% on a R3000. The performance of dividing a 1024x1024 matrix of type `short` by a runtime constant improved by a factor of 10 in a SPARC 10 when the constant was a power of two and by a factor of 4 for more common values. On an R3000 the improvement was approximately a factor of 2. More dramatic improvements would be possible with a more sophisticated factorization scheme.

6 Retargeting `dcg`

Once `dcg` is retargeted, all clients will run on the new target machine. Despite the fact that each retarget

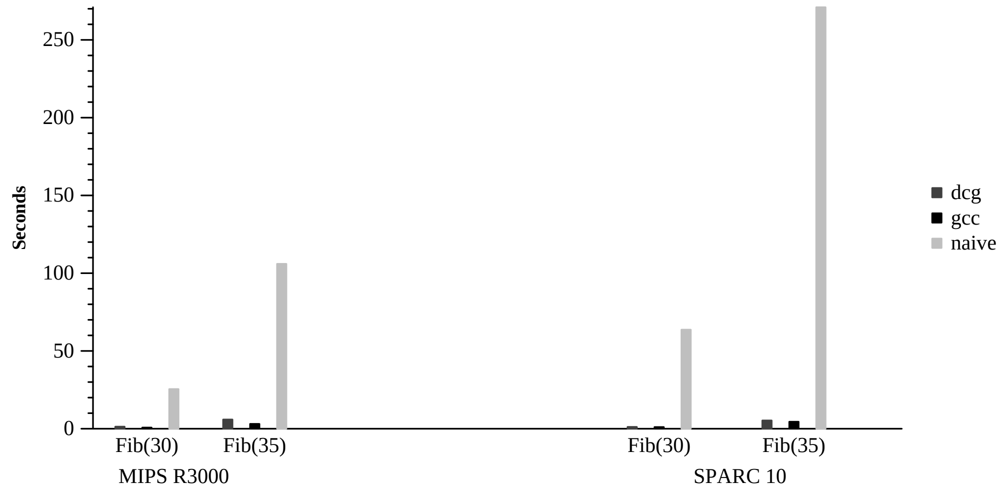

Figure 4: Tiny-C Timings on Fibonacci (timings in seconds)

is only done once, we still felt it important to make the process as easy as possible.

Retargeting `dcg` consists of three parts: creating a mapping from IR patterns to machine instructions, creating a mapping from machine instructions to binary templates, and defining auxiliary code for observing calling conventions, data layout restrictions, register allocation, etc. We developed two small languages to make retargeting easy. The first language expresses the mapping of IR patterns to machine instructions. The second language expresses the mapping of machine instructions to binary patterns that can be emitted and directly executed by the client.

`burg` automatically generates routines that efficiently map IR trees to machine instructions based on tree patterns. We use a richer pattern specification language than `burg` that is preprocessed into a `burg` specification and auxiliary routines. The language's grammar follows:

```
template: 'template' '=' type ':' burgname {
          '[' typelist ']' } '|' text
          '{' { mlist }* '}'
/* macro list */
mlist: mname '=' token { ',' token }* ';'
/* macro name */
mname: @[a-zA-Z0-9_"]?
```

`type` denotes the resulting type of a given action (e.g., `reg`, `const`, `addr`); `burgname` is the name of the `burg` rule; `typelist` is a list of child types; `text` is literal text that is emitted with each rule after macro expansion. The preprocessor expands the rules by marching down the macro lists in parallel and substituting the given token for any targets given in the template. The preprocessor has a few predefined replacements (e.g., `@r0` will give the register of the tree's leftmost child); two global state variables (`@type` and `@regpol`) control, respectively, the type of each `burg` rule and the register allocation/deallocation of each associated action. The resulting text after macro expansion is emitted into two files: a `burg` input specification, and an emitter, written in C.

For example, the specification

```
/* rule type is reg */
@type = reg;
/* deallocate children, allocate parent */
@regpol = 2;
/* binary operations */
template = @type:@burg[reg, reg] |
        asminst("@bop $r2, $r1, $r !@burg");
        @bop(@r, @r0, @r1); {
        @burg = BXORU, ADDI, ADDP, ADDU;
        @bop  =   xor,  add,  add,  add;
}
```

produces the following `burg` rule, and associated C code for `ADDI` (integer addition)

```
/*
 * Burg rule -- goes into file for burg
 * input spec.
 */
reg: ADDI(reg, reg) = 9 (1);
/*
 * Burg action -- goes into file containing
 * the emitter.
 */
```

```
case 9: /* reg: ADDI(reg, reg) = 9 (1); */
        putreg(b); putreg(a); getreg(p);
        asminst("add $r0, $r1, $r !ADDI");
        add(p->x.reg, a->x.reg, b->x.reg);
        break;
```

Our second preprocessor generates a binary emitter for machine instructions. The binary emitter is responsible for constructing the 32-bit values that encode a particular instruction. The input grammar follows:

```
template: '(' args ')' literal_text '{'
          insn_name { insn_name }*
          binary { binary }*
'}'
insn_name=[a-zA-Z0-9]
binary=[01]?
```

Two lists are specified: a list of instruction names (e.g., `add`, `addu`) and a list of their corresponding binary values (e.g., `100000`, `100001`). The instruction name is concatenated to the template and the binary value, after conversion to hexadecimal, replaces any `@bin` label. The result is emitted as a C macro that will construct the 32-bit instruction from the opcode and any operands. (In the example below, `STYPE` is a macro that builds 3-operand instructions for the SPARC.) We currently do not handle machines with variable-size instructions.

For example, the specification:

```
(dst, src1, src2)
    STYPE(@bin, dst, src1, src2); {
        add    addu   and    nor    or
        100000 100001 100100 100111 100101
}
```

Yields the following for `add`:

```
#define add(dst, src1, src2) \
            STYPE(0x20, dst, src1, src2)
```

The current specification languages are small, but not as concise as other code generator specification languages [7]. Future work will involve making the preprocessors more sophisticated.

Few architectures with separate I/D caches require that the I cache be kept coherent with memory. Consequently, dynamic code generation requires that coherence be maintained manually. The R3000 does have separate I/D caches, but a system routine is provided for flushing the caches over a given address range. The SPARC ABI documentation, after prudish warnings against self-modifying code, states that a special instruction must be used to explicitly flush each word of code. Fortunately, most SPARC implementations have unified I/D caches, obviating this requirement.

7 Future Work

While `dcg` generates good code quickly, it can be improved. `dcg` currently does not schedule instructions, and must therefore take a conservative approach to emitting instructions — too many nops are generated. In the machines we targeted, we estimate that this can degrade performance by up to 25%. Retargetable scheduling systems would help eliminate this performance penalty [2, 18]. Local code could be improved by peephole optimization and by special-casing leaf procedures.

While the interface of `dcg` is more civilized than machine code, it can be improved. We are currently investigating two approaches. The first would be to use `lcc` as a preprocessor that would accept C code as input and emit the corresponding IR. The second would be to augment ANSI C with language features that allow dynamic code generation to be controlled from within the language proper. This has the advantage that the cost of some optimizations done by `dcg` at runtime could be shifted to compile time.

Our current model assumes that all procedure calls obey C calling conventions. This is a serious restriction for dynamic code generation clients that wish to customize calling conventions for improved performance. We anticipate augmenting `lcc`'s IR to provide a lower-level view of procedure calls for such clients.

8 Conclusion

Our system, `dcg`, provides a set of routines that define a portable, efficient dynamic code generation system. The machine-independent intermediate representation specifies a small, but rich, set of operators that are sufficient to express all C language constructs at nearly a machine-level, without sacrificing portability. The code generation interface is small and easy to use — clients specify expression trees for a desired chunk of code and `dcg` returns a function pointer callable from the client program.

`dcg` generates good executable code quickly. Optimal tree pattern matching with BURS technology, and careful engineering provide a system that can generate executable instructions with at the rate of one instruction every 350 instructions.

Because the IR was taken from an existing retargetable C compiler, it is easy to retarget. This job is further simplified by preprocessors developed to simplify instruction selection and creation of binary emitters.

9 Acknowledgements

Christopher Fraser, Wilson Hsieh, Anthony Joseph, Kevin Lew, Andrew Myers, Carl Waldspurger and Deborah Wallach carefully read this paper and their insightful comments greatly improved it. Lorenz Huelsbergen brought the matrix multiplication example to the attention of the second author, and contributed valuable ideas about runtime code generation. Professor M. Frans Kaashoek of M.I.T. graciously allowed the first author to complete this "legacy work" at the beginning of his graduate career; his support is greatly appreciated.

References

[1] Alfred V. Aho, Mahedevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[3] Craig Chambers and David 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, June 1989.

[4] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):7–32, October 1984.

[5] Peter Deutsch and Alan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[6] Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG—a generator for efficient back ends. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, 1989.

[7] Christopher W. Fraser. A language for writing code generators. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 238–245, 1989.

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

[9] 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.

[10] Torbjorn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[11] SPARC International. *The SPARC Architecture Manual*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

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

[13] David 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.

[14] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington, 1991.

[15] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. of the Eleventh ACM Symposium on Operating System Principles*, pages 39–51, Nov. 1987.

[16] Rob Pike, Bart N. Locanthi, and John F. Reiser. Hardware/software trade-offs for bitmap graphics on the blit. *Software—Practice and Experience*, 15(2):131–151, February 1985.

[17] Todd A. Proebsting. Simple and efficient BURS table generation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[18] Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *Proceedings of the 21th Annual Symposium on Principles of Programming Languages*, January 1994. to appear.

[19] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

A Static Matrix Multiplication Routines

We include the code for the indir and dcg multiplication routines.

```
/* indirection vectors are used to record relevant indices -- written
 * by David Mosberger-Tang */
void matrix_mult(int *nzv, int *nzi) {
    int i, j, k, *b_j, n_nz, s;

    for(i = 0; i < n; i++) {
        for (s = n_ns = k = 0; k < n; k++) {
            if (a[i][k] != 0) {
                s += a[i][k] * b[0][k];
                nzv[n_nz] = a[i][k];
                nzi[n_nz++] = k;
            }
        }
        c[i][0] = s;
        for (j = 1; j < n; j++) {
            b_j = b[j];
            for (s = k = 0; k < n_nz; k++) {
                s += nzv[k] * *(b_j + nzi[k]);
            }
            c[i][j] = s;
        }
    }
}
/* Construct a tree representing a customized dot-product computation
 * using dcg. 'n' is the size of the matrix, A is a pointer to the
 * row being customized and arg is a pointer to the arguments symbol.  */
Node mkdot(int n,int *A, Symbol arg) {
     Node sum=NULL, mul, a, b;
     int j;
     /* march down row, checking for zeros */
     for(j=0;j<n;j++)  {
          if(A[j] != 0) {
               /* index off of a pointer passed as a parameter */
               a = index(addrfp(arg[0]), sizeof(int)*j);
               b = cnsti(scnsti(A[j]));           /* constant whose value is A[j] */
               mul = muli(a, b);                  /* multiply node */
               sum = !sum ? mul : addu(sum, mul); /* construct dot product */
          }
     }
     return !sum ? NULL : reti(sum); /* return a tree, if any was constructed */
}
```