Finite-State Code Generation

Christopher W. Fraser
Todd A. Proebsting

Microsoft Research*

Abstract

This paper describes GBURG, which generates tiny, fast code generators based on finite-state machine pattern matching. The code generators translate postfix intermediate code into machine instructions in one pass (except, of course, for backpatching addresses). A stack-based virtual machine—known as the *Lean Virtual Machine* (LVM)—tuned for fast code generation is also described. GBURG translates the two-page LVM-to-x86 specification into a code generator that fits entirely in an 8 KB I-cache and that emits x86 code at 3.6 MB/sec on a 266-MHz P6. Our just-in-time code generator translates and executes small benchmarks at speeds within a factor of two of executables derived from the conventional compile-time code generator on which it is based.

1 Introduction

To execute virtual machine (VM) code on a client processor typically requires either a VM interpreter or a *just-in-time* (JIT) translator. Conventional wisdom dictates that the space/time tradeoff favors the interpreter approach where space is scarce because interpreters are small and "simple," but it favors the presumably larger JIT translators when speed is more important than size or simplicity. Interpreters typically give up a factor of 10 in execution speed compared to JIT-translated code. In this paper we will describe tiny, fast and simple code generators that produce native code whose speed *including JIT translation* is within 2-4X of typical compiler-generated code. The factor is less than two when compared with the conventional compile-time code generator on which our system is based. Given that our x86 code generator fits in 8 KB and generates code at 3.6 MB/sec., this technology represents a desirable alternative to interpretation on even extremely memory-limited machines (e.g., cellular phones, personal digital assistants, etc.). Furthermore, our system generates these code generators from concise machine specifications, which greatly simplifies retargeting.

Pattern-matching code generators map patterns of intermediate representation (IR) operators to equivalent target instructions. Tree-pattern matching code generators often use target-machine instruction costs to guide the selection towards least-cost (i.e., optimal) mappings via dynamic programming, which require two tree-walk passes: one bottom-up pass for dynamic programming, and one top-down pass for selecting the least-cost match. Other systems, like `gcc`'s IR-rewriting technology, heuristically search for better target instructions to emit. Both techniques do an excellent job of instruction selection but new applications for code generators—such as load-time translation of mobile code in embedded computers or on-the-fly code generators for interpreters, emulators, or program specializers—might benefit from alternatives that are smaller or faster or both.

This paper explores the other end of the spectrum of code generation technology, where code generator size and speed are as important as code quality. The extreme end of the spectrum is a *macro-expanding* code generator that reads every IR instruction and immediately emits the appropriate target instructions. This approach, while simple and fast, misses opportunities to emit more efficient target instructions that do the work of multiple IR operations. For instance, macro expansion of an `Add` followed by a `Load` could not exploit a target machine's addressing

Microsoft Research, One Microsoft Way, Redmond, WA 98052. Email: {cwfraser,toddpro}@microsoft.com.

SIGPLAN '99 5/99 Atlanta, GA, USA

modes.

We have systematically studied the costs of previous pattern-matching schemes and eliminated as much inefficiency as possible while still being able to generate good local code. Our system, "GBURG," automatically generates code generators whose underlying pattern-matching technology is a common finite state machine. This simplicity results in small, simple and efficient code generators that are able to generate many complex target instructions.

GBURG reads tree grammars and automatically generates code generators. GBURG does not operate on trees, but rather their linearized postfix notation (i.e., stack code). Furthermore, the code generators require only one pass over—and thus no buffering of—the stack code to select instructions. These one-pass matchers do greedy pattern matching and cannot guarantee least-cost matches. Based on the emitted assembly language we've examined, however, the code quality is suitable for many purposes where fast code generation is important.

GBURG cannot handle arbitrary tree grammars—it processes grammars that are equivalent in descriptive power to regular expressions, which allows GBURG to generate simple finite-state machine pattern matchers.

We have designed a new *Lean Virtual Machine* (LVM) for our intermediate representation. The LVM's instruction set design helps GBURG to generate good code very quickly.

Our two-page LVM-to-x86 code generator specification generates a code generator that fits entirely in 8 KB. Run on a 266 MHz P6, the code generator is capable of generating x86 code at 3.6 MB/sec.

2 Background

Generating fast code generators from machine specifications is not new [GG78, FHP92b, FHP92a, AGT89, PW96]. Graham-Glanville code generators parse prefix tree IR to identify large instructions (and addressing modes) from simpler IR operators. Graham-Glanville code generators suffer from a "left-bias" when translating binary operators. The prefix code for a binary operator's right operand can be arbitrarily many instructions away from the operator—the left operand separates the two. The Graham-Glanville code generator cannot defer parsing decisions arbitrarily far because it is based on LR(1) parsing technology, and, therefore, must make code generation decisions about a left operand prior to knowing its sibling, the right operand. Having to make decisions without complete information can lead to sub-optimal code. GBURG suffers from a similar left bias—although it matches a postfix notation—but the design of the LVM instruction set compensates for this problem.

Engler's VCODE system represents a VM instruction set and a code generation technology for efficient dynamic code generation [Eng96]. VCODE includes complex instructions that anticipate most modern architectures in terms of addressing modes, which means that macro-expansion of VCODE into target code will often use those addressing modes. Like GBURG's code generators, VCODE's code generators are generated from specifications and are extremely efficient, but VCODE relies entirely on the design of the instruction set to allow macro-expansion to reasonable code.

Omniware is a system for distribution of safely-executable mobile programs [ATLLW96]. Omniware relies on a machine-independent VM code that can be efficiently translated into native code via macro-expansion. Like VCODE, Omniware's instruction set design anticipates target machine instruction sets by including complex addressing modes.

The Free Software Foundation's GCC compiler uses a very general tuple-rewriting system for instruction selection. Based on the PO system for rewriting Register Transfer Language developed by Davidson and Fraser, the system is extremely flexible and powerful, but not known for speed [DF80, DF84].

Tree-pattern matching technologies combined with dynamic programming yield efficient, optimal local code generation for tree-based IRs. Previous tree-pattern matching schemes have used dynamic programming combined with sophisticated matching algorithms [HO82, PLG88, Pro95] to produce least-cost tree matches in time proportional to the size of the input tree. (The problem is NP-complete for DAG-based IRs.) All dynamic programming systems require two passes over the IR: the first pass annotates the tree with dynamic programming information, and the second pass selects the optimal match. While generating provably optimal code from trees, the code generators do so at the cost of two passes over the IR. Furthermore, the second pass requires a top-down tree walk of the IR, which implies a format more complex than a simple postfix stack code. GBURG does a one-pass pattern match that trades optimality for instruction selection speed.

Like GBURG, `wburg` generates "one-pass" tree-pattern matchers from machine specifications [PW96]. `Wburg`'s matchers are based on `burg`-generated automata, which means they are relatively large [Pro95]. Furthermore, `wburg`'s matchers operate in one-pass by buffering a small fixed-size stack

of previously seen operations for deferred matches, which is an overhead stripped from GBURG's matchers.

Typically, tree-pattern matching code generators are automatically generated from concise machine specifications that map IR patterns to target machine instructions. While the code generators often differ in pattern-matching algorithms and disambiguation techniques (e.g., dynamic programming, greedy selection, etc.), the specifications are quite similar. The grammars consist of cost-augmented tree-rewriting rules that associate patterns with nonterminals they derive and with actions that the code generator must execute if this rule is chosen. A sample grammar appears in Figure 1.

Nonterminals appear on the left-hand side of rules. *Terminals* are IR language operators. *Base rules* include a *terminal* symbol on the right-hand side of rules. *Chain rules* simply derive one nonterminal from another. Costs appear in parentheses.

3 Lean Virtual Machine

This paper bases IR examples on the Lean Virtual Machine (LVM), which we are developing as a vehicle for research in code generation and compression. The LVM is a simple stack-based machine designed to enable, among other things, efficient translation to target-machine code. The LVM is "lean" in the sense that the instruction set eliminates all redundant operations. For instance, the only addressing mode in the LVM is indirect (i.e., loads and stores find their target addresses on the evaluation stack). The LVM includes only two primitive operations for accessing literal values: one pushes a compile-time constant on the stack, and the other pushes a link-time constant on the stack. (These operations come in different sizes and types, of course.)

The stack-based LVM includes 256 registers. By convention, these LVM registers may, or may not, map to specific hardware registers.

The LVM instruction set does not include any instructions that assume particular source languages, calling conventions, or object models. For instance, the LVM does not include any instructions for passing arguments, checking types, entering monitors, etc. Such operations must be built from the primitive LVM operations. By including all the necessary primitive operations to map *any* such operations onto a particular target machine, and by avoiding hardwiring a particular language bias into the LVM, it can function as a universal target for all source languages.

The LVM borrows heavily from the `lcc` IR [FH95]. It is, by and large, `lcc` trees in postfix, with C-specific operators expanded into a few more primitive operators and with the operators renamed to be somewhat more self-explanatory in their new incarnation as instruction names.

4 Greedy Pattern Matching

Our code-generator generator, GBURG (Greedy Bottom-Up Rewrite Generator), generates tiny and extremely fast code generators that do tree pattern matching as their way of mapping postfix instructions into target machine instructions. GBURG forsakes dynamic programming and unrestricted tree matching for a simpler, faster scheme that matches patterns greedily from a very restricted tree-pattern grammar. Fortunately, the quality of generated code suffers very little, while the code generation speed and size improve dramatically.

GBURG does not traverse trees, but rather it reads a postfix representation. Let's use the grammar in Figure 1 to compare GBURG to other schemes. Consider the following LVM code that represents a simple load.

```
PushRegU4[2] PushConstU4[4] AddU4 LoadU4
```

On a simple RISC target, this would map cleanly into a load instruction: `load rX, 4(r2)`.

A dynamic programming system would find all legal parses of the tree, and choose the least expensive for this grammar: `load rX, 4(r2)`. (The code generator would be responsible for assigning a register to the implicit temporary `rX`.)

What problems does a one-pass system have with this example? Translating the first LVM instruction (`PushRegU4`) poses an subtle choice. While there is only one rule that matches `PushRegU4`, which produces the `reg` nonterminal, it is not obvious which chain rules, if any, should be used. In general, it is impossible to know which chain rules should be applied until subsequent operators are inspected—which is precisely the reason that dynamic programming systems require two passes.

Peeking ahead and finding the `PushConstU4` instruction does help. Examination of the grammar reveals that *all* first operands (left children) must be `reg` nonterminals, so the code generator knows to apply the necessary chain rules to reduce the left-child nonterminal to a `reg` (which in this case is no rules at all). Note that if peeking ahead revealed a `LoadU4` instruction, then the `addr: reg` chain rule would be applied to get the necessary `addr` nonterminal.

```
cnsti: PushConstU4[N]     (0)     emit("N")
reg:   cnsti              (1)     emit("loadimm $0, $1")
reg:   PushRegU4[N]       (0)     emit("rN")
reg:   LoadU4(addr)       (1)     emit("load $0, $1;")
reg:   AddU4(reg, reg)    (1)     emit("addi $0, $1, $2;")
addr:  AddU4(reg, cnsti)  (0)     emit("$2($1)")
addr:  reg                (0)     emit("0($1)")
addr:  cnsti              (0)     emit("$1")
```

Figure 1: Sample Grammar

What chain rules should be applied after reducing `PushConstU4`? Should the `PushConstU4` be reduced to a `cnsti`, `addr`, or a `reg`? Unfortunately, this cannot be answered by looking just one instruction ahead. The required nonterminal depends on which `AddU4` rule should be used, which depends on yet another instruction—the instruction that uses the `AddU4`'s computation. In general, there is no bound on how far ahead the pattern matcher might need to look to determine the best rule(s) to apply.

To avoid looking arbitrarily far ahead, GBURG-generated pattern matchers *greedily* match base rules, and defer applying all chain rules until the very next instruction is examined. In the example, this means that `PushConstU4` would be reduced to a `cnsti` by the only base rule for `PushConstU4`. Upon encountering the `AddU4`, the pattern matcher would attempt to apply each `AddU4` rule, in turn, until one of them matched (with the possible assistance of chain rules for the *most recently encountered* operand) and then use that rule. So, `reg: AddU4(reg,reg)` would be used after forcing the application of a chain rule to promote the `cnsti` from the `PushConstU4` into a `reg`. The `AddU4`'s `reg` would in turn be promoted to an `addr` when the matcher hit the `LoadU4` instruction.

Note that it is impossible for this greedy scheme to *ever* use the `addr: AddU4(reg,cnsti)` rule, which directly creates an addressing mode. This follows because a `cnsti` can always be converted to a `reg` and, therefore, anytime the second `AddU4` rule is applicable, so is the first. GBURG input grammars have no concept of costs, so another means is necessary to overcome this deficiency. Fortunately, it can be overcome by rewriting the grammar so that the `AddU4` rules are reversed.

Unfortunately, some grammars still might require deferring chain rules for more than one instruction. Adding the rule `root: StoreU4(addr,reg)` introduces such a problem. Because `addr` is the first operand, and other binary operator rules have `reg` nonterminals as their left operand, it would be impossible to know what chain rules should be applied to a nonterminal that will be used as a left operand without knowing the parent operator. This rule causes a left-bias problem. Deferring such decisions would effectively require a second pass over the input.

Fortunately, there is a simple way out of this problem: change the `StoreU4` instruction in the IR to take the target address as its right child. For RISC targets like the SPARC, this eliminates the left-bias for the translation of a `StoreU4`. Unfortunately, this `StoreU4` definition does not solve a related code generation problem on the x86. On the x86, it is possible to store an immediate value into a target location specified by a complex addressing mode. Generating this instruction requires a rule like `root: StoreU4(imm, addr)`. This cannot be matched in a simple one-pass matcher, and GBURG's code will load the constant into a register before storing it into memory.

5 Code-Generator Generator

5.1 Machine Specifications

GBURG is a 622-line Icon program [GG83] that does only simple analysis of an input grammar to produce a code generator in C [KR88]. The input consists of the following parts:

Token Declarations The specification includes declarations for all operators in the grammar including their external encoding. All encodings must fit in 8 bits.

Constructive Grammar The specification includes a grammar that describes all legal tree derivations. GBURG eliminates some case analysis in its pattern matcher when the constructive grammar is more constrained than the machine specification grammar. The constructive grammar also declares how many bytes of "immediate value" follow the operator in an instruction stream (e.g., `PushConstU4[4]`).

```
%% // Constructive Grammar
root:  StoreU4(int,int)
root:  StoreF4(float,int)
int:   AddU4(int, int)
int:   SubU4(int, int)
float: AddF4(float, float)
float: SubU4(float, float)
int:   PushConstU4[4]
float: PushConstF4[4]
int:   LoadU4(int)
float: LoadF(int)
%% // Machine Specification Grammar
root:  StoreU4(reg,addr) { printf("root: StoreU4(reg,addr)"); }
root:  StoreF4(reg,addr) { printf("root: StoreF4(reg,addr)"); }
reg:   AddU4(reg, cnsti) { printf("reg: AddU4(reg, cnsti)"); }
reg:   AddU4(reg, reg)   { printf("reg: AddU4(reg, reg)"); }
reg:   SubU4(reg, cnsti) { printf("reg: SubU4(reg, cnsti)"); }
reg:   SubU4(reg, reg)   { printf("reg: SubU4(reg, reg)"); }
reg:   AddF4(reg, reg)   { printf("reg: AddF4(reg, reg)"); }
reg:   SubF4(reg, reg)   { printf("reg: SubF4(reg, reg)"); }
cnsti: PushConstU4       { printf("cnsti: PushConstU4"); }
cnstf: PushConstF4       { printf("cnstf: PushConstF4"); }
reg:   LoadU4(addr)      { printf("reg: LoadU4(addr)"); }
reg:   LoadF4(addr)      { printf("reg: LoadF4(addr)"); }
reg:   cnstf             { printf("reg: cnstf"); }
reg:   cnsti             { printf("reg: cnsti"); }
addr:  reg               { printf("addr: reg"); }
addr:  cnsti             { printf("addr: cnsti"); }
%%
// C code here
```

Figure 2: Sample Specification

Machine Grammar The specification includes parsing rules annotated with actions (encoded in C). A rule includes a left-hand side nonterminal, a pattern, and an action. The rules take two possible forms: a *chain rule* that has a nonterminal as its pattern, or a *base rule* that has a tree pattern—a terminal for the operator, and nonterminals as children. The matcher executes an action when it selects the associated rule.

Auxiliary C Routines The specification includes arbitrary C code that compiles and links with the generated matcher.

GBURG emits a single C function, `compile()`, which takes as an argument an array of postfix code to be matched. Figure 2 contains a sample grammar that would simply echo its derivation sequence.

5.2 Generating Matchers

GBURG matchers read stack-based input and perform actions before and after each operator. Before performing the appropriate action associated with the operator, it may be necessary to apply chain rule actions to the preceding nonterminal to enable a match. To allow a one-pass matcher, chain rules are only applied to the nonterminal produced by the immediately preceding operator. For instance, if the preceding operator produced a `reg`, and if the next operator is a `LoadU4`, the rule `addr: reg` must be applied before applying the `LoadU4` rule. Note that a subtle asymmetry exists here: chain rules are always applied to the nonterminal generated by the preceding operator, whereas base rules are always applied to the current operator.

As discussed above, the matchers never apply chain rules to nonterminals other than those generated by the preceding operator. When a unary or binary op-

erator is encountered, it is simple to determine which chain rule(s) to apply—use the chain rules necessary to get the appropriate nonterminal for a match. If more than one base rule exists for an operator, use the first rule that can be applied (even if it requires application of chain rules). This is a greedy matcher.

When a nullary operator (a leaf operator in the tree) is encountered, what chain rules should be applied to the previous nonterminal (if one exists)? This is determined by analysis of the grammar. In the grammar above, nullary operators will cause application of chain rules to derive the `reg` nonterminal. GBURG determines this trivially by noting that all left-child nonterminals in the grammar are `reg`s, and, therefore, any matches must match this as a `reg`. This follows from the observation that in a stack machine, all values below the top of the stack are only consumed by binary operators. GBURG allows more than one nonterminal to appear as the left child of various binary operators, but GBURG's analysis must prove, via a conservative analysis, that this presents no ambiguity during a match.

The conservative analysis is trivial. First, determine that all rules for a given operator have the same nonterminal as their left child. (Otherwise, it *might* be impossible to pick a rule based solely on the right child.) Further, check that no nonterminal can be derived from two distinct left-child nonterminals via chain rules. (Otherwise, it would be impossible to determine what chain rules to apply to the original nonterminal when "pushing it on the stack.")

5.2.1 Finite state machine

Because only the immediately preceding nonterminal can affect the pattern matching process, only that nonterminal needs to be remembered during matches. All pattern matching is determined by this single nonterminal and the current operator. This, of course, defines a simple finite-state machine in which the last nonterminal is the state, and the current operator is the input symbol. Each state corresponds to a nonterminal in the machine specification grammar. Therefore, the machine for the grammar above would have five states: `root`, `reg`, `cnsti`, `cnstf`, and `addr`.

All that is necessary to realize this state machine in code is a mechanism for performing the appropriate actions (associated with grammar rules) and making the appropriate state transitions. Note that for a given transition there will be zero or more chain rules applied to the preceding nonterminal (state), and exactly one base rule applied.

Determining which rules should be applied for a given (nonterminal,operator) pair is easy. First, GBURG determines which base rule to apply by greedily trying them in order, allowing as many chain rules as necessary to be appied to make a match work. (Hence, GBURG is greedy with respect to base rules, not chain rules. Given alternative sequences of chain rules to do the same conversion, GBURG will choose the shortest, breaking ties arbitrarily.) Given this analysis, emitting a pattern matcher is easy, although there are opportunities for optimization.

All of our schemes use a hard-coded matcher, rather than an interpretive, table-driven matcher. The hard-coded matchers that we generate avoid any explicit storage of the last nonterminal by encoding this in the program's PC. (This is analagous to a recursive-descent parser encoding the current nonterminal in its execution of a particular procedure.)

5.2.2 Switch Statement

Each nonterminal translates to code for handling an operator after a reduction to that nonterminal. C's `switch` statement is used to choose actions given the possible operators. Application of any rule causes execution of the associated action, and a transfer to the code corresponding to the left-hand side of the rule (i.e., a state transition). Applying a base rule also advances to the next operator. (The LVM includes an `EndOfProgram` operator, whose action returns from the matching procedure.) For the grammar above, abbreviated translations for the `reg` and `addr` nonterminals appear in Figure 3.

This code is simple, but it can be improved. Combining identical case arms can decrease its size. For instance, the `addr` switch statement is made smaller by using only one case arm for `SubU4` and `AddU4`, as in Figure 4.

5.2.3 Operator Propagation

One drawback of the previous scheme is that a given operator may flow through many switch statements—in fact, it will execute one switch statement per rule (chain and base) that it forces. This inefficiency is easy to eliminate by exploiting the fact that after the first switch statement, it is known exactly which case arm of all the subsequent switch statements will be executed. Therefore, it is simple to transfer control out of each switch statement directly into the appropriate case arm of the next switch statement. Given state `reg` and operator `LoadU4`, it must be the case that after applying chain rule `addr: reg` that the rule `reg: LoadU4(addr)` will be applied—so it can be jumped to directly. Optimized in this way, the switch statements above appear in Figure 5.

```
reg:
switch (operator) {
case AddU4:  { printf("reg: AddU4(reg, reg)"); operator = *PC++; goto reg; }
case SubU4:  { printf("reg: SubU4(reg, reg)"); operator = *PC++; goto reg; }
case LoadU4: { printf("addr: reg");                              goto addr; }
...
default:    /* error handling code */
}

addr:
switch (operator) {
case AddU4:  { printf("reg: addr");                           goto reg; }
case SubU4:  { printf("reg: addr");                           goto reg; }
case LoadU4: { printf("reg: LoadU4(addr)"); operator = *PC++; goto reg; }
...
default:    /* error handling code */
}
```

Figure 3: Simple State Machine

```
addr:
switch (operator) {
case AddU4:  /* share chain rule with SubU4 */
case SubU4:  { printf("reg: addr");                           goto reg; }
case LoadU4: { printf("reg: LoadU4(addr)"); operator = *PC++; goto reg; }
...
default:    /* error handling code */
}
```

Figure 4: Code Sharing

After this optimization, only base rules transfer control to a switch statement. Because many chain rules contain no actions, this optimization introduces branch chains, which, fortunately, many compilers are able to eliminate. Unfortunately, the previously described case-arm sharing disappears.

5.2.4 Chain Rule Inlining

It is possible to take the previous optimization a step further and eliminate the intermediate control transfers altogether. By inlining—in possibly many locations—chain rules, it is possible to guarantee exactly one control transfer per operator, regardless of the number of applied chain rules. After inlining, the code above is transformed into the code in Figure 6

The downside of this transformation is, of course, code bloat. (Amusingly, a compiler doing a cross-jumping optimization would re-introduce many of these eliminated jumps to decrease the bloat.)

5.2.5 Equivalence Classes

GBURG supports a crude macro-like facility that enables another optimization. GBURG specifications can group operators that have identical rule and action templates. For instance, `AddU4` and `SubU4` are nearly identical in the grammar above, and GBURG supports a grammar specification of the following form.

```
[AddU4:123 SubU4:456]
reg: $1(reg,reg) { printf("%d", $2); }
```

This is shorthand for writing rules for each of the operators. `$1` is a shorthand for the operators in the equivalence class (the left-hand side of the macro definitions), and `$2` is a shorthand for the right-hand side of each definition. GBURG restricts the right-hand side of macro definitions to be integers that will be stored in a table indexed by the operator (i.e., left-hand side). Macros make specifications more concise, and they provide GBURG with another optimization

```
reg:
switch (operator) {
reg_AddU4:
case AddU4:  { printf("reg: AddU4(reg, reg)"); operator = *PC++; goto reg; }
reg_SubU4:
case SubU4:  { printf("reg: SubU4(reg, reg)"); operator = *PC++; goto reg; }
reg_LoadU4:
case LoadU4: { printf("addr: reg");                              goto addr_LoadU4; }
...
default:    /* error handling code */
}

addr:
switch (operator) {
addr_AddU4:
case AddU4:  { printf("reg: addr");                          goto reg_AddU4; }
addr_SubU4:
case SubU4:  { printf("reg: addr");                          goto reg_SubU4; }
addr_LoadU4:
case LoadU4: { printf("reg: LoadU4(addr)"); operator = *PC++; goto reg; }
...
default:    /* error handling code */
}
```

Figure 5: Propagation

```
reg:
switch (operator) {
case AddU4:  { printf("reg: AddU4(reg, reg)"); operator = *PC++; goto reg; }
case SubU4:  { printf("reg: SubU4(reg, reg)"); operator = *PC++; goto reg; }
case LoadU4: { printf("addr: reg"); /* NOTE:  two rules applied here */
               printf("reg: LoadU4(addr)");    operator = *PC++; goto reg; }
...
default:    /* error handling code */
}

addr:
switch (operator) {
case AddU4:  { printf("reg: addr"); /* NOTE:  two rules applied here */
               printf("reg: AddU4(reg, reg)"); operator = *PC++; goto reg; }
case SubU4:  { printf("reg: addr"); /* NOTE:  two rules applied here */
               printf("reg: SubU4(reg, reg)"); operator = *PC++; goto reg; }
case LoadU4: { printf("reg: LoadU4(addr)");    operator = *PC++; goto reg; }
...
default:    /* error handling code */
}
```

Figure 6: Inlining

opportunity. GBURG can create matchers that operate on operator equivalence classes rather than individual operators. Thus, each equivalence class has a case arm rather than each operator. Of course, this requires that the associated rules be parameterized by the macro substitutions. For an extra level of indirection, equivalence classes provide a simple technique for decreasing the size of the specification and the matcher.

6 LVM Design

The development of GBURG influenced LVM instruction set design choices. The LVM's instruction set avoids, where possible, a left-bias that would yield inferior code quality. To do this, the left children of binary operators are those that are unlikely to create any ambiguity during translation to machine code for *most* target machines. Note that left-bias is *always* relative to a particular target machine. Therefore, in designing the LVM to avoid left-bias we must examine current machines and try to anticipate future machines.

Commutative operators (e.g., `AddU4`) provide an interesting challenge. A machine that provides an add-immediate instruction would likely have the following symmetric rules.

```
reg: AddU4(cnsti, reg)
reg: AddU4(reg, cnsti)
```

These rules introduce a left-bias problem because the matcher cannot know whether or not a `cnsti` should be promoted to a `reg` when used as a left operand. To solve this problem, we use only the second rule and rely on the LVM-generator to produce LVM code in a canonical form that has all literal values as right children of commutative operators.

As noted previously, `StoreU4` poses a potential left-bias. Most target machines have some sort of addressing mode that is more complex than simple indirect (e.g., *register+constant*, *register+register*, etc.). These addressing modes imply a machine specification nonterminal for the addressing mode computation (e.g., `addr`). If all target machines store only values from registers, the potential left-bias problem is avoided by designing the LVM's `StoreU4` instruction to take its target address as its right operand. Thus, the grammar would have a rule like `root: StoreU4(reg,addr)` and there would be no left bias (assuming `reg` as a left operand did not cause a bias). This takes care of the common case on all architectures.

A slightly more difficult situation occurs with the interaction between relational operators and conditional jumps. Relational operators produce a logical value, and conditional jumps require a logical value and a jump target. Our original design had relational operators producing an *integer* value rather than a logical one, which created a left-bias in the matcher: if a literal target address came first, it would be generated into a register, and if a conditional came first, its value would be computed into a register. Of course, this is unacceptible on machines that have conditional jumps to constant targets. The solution was to have conditional operators produce logical values that require an explicit operation to convert to an integer value. This requires more LVM instructions to express value-producing conditional expressions, but such expressions are rare.

We restrict legal LVM code in a number of ways to facilitate fast code generation. For instance, we require that programs be a sequence of completely formed "trees" (in postfix notation), thus eliminating complications in a tree-pattern matcher that must process incompletely formed trees or directed acyclic graphs (DAGs). This restriction means that the evaluation stack is empty at the start and end of each tree, and, therefore, at the start and end of each basic block. Such a restriction eliminates the need to do any sort of control-flow analysis to determine stack configurations at branch targets. (This is in contrast to a less restrictive rule in the Java VM [LY97].)

It is actually impossible to create a directed acyclic graph (DAG) with LVM code because the LVM instruction set does not include a `Dup` instruction that would create more than one reference to the same value. Restricting LVM to trees eliminates needing more complicated algorithms for doing pattern matching in DAGs—a problem known to be much more complicated. Furthermore, the LVM does not include a `Swap` instruction, which would enable the expression of code that might, for instance, create a second operand before the first operand of a binary operator. This restriction enables the LVM design to avoid, where possible, a left-bias with only a one-pass pattern match.

Legal LVM code must also be type-consistent. For instance, it is not legal to use an integer add instruction on two floating point numbers. This is, however, a much weaker restriction than Java's verifiability restrictions because we only do typing at a primitive level (e.g., integers and floats), and memory is completely untyped. Only the evaluation stack is typed. Because LVM code is type consistent, the code generator is spared costly checks.

Code generator generator	Code generator		
	Size	Speed	
	(KB)	Read LVM (MB/sec.)	Emit x86 (MB/sec.)
GBURG	7.3	5.5	3.6
burg	20.1	2.2	1.4
iburg	25.9	1.8	1.2

Table 1: Code Generator Sizes and Pattern Matching Speeds

Translation System	bubble.c (sec.)	perm.c (sec.)	puzzle.c (sec.)	queens.c (sec.)	towers.c (sec.)
GBURG	16.5	11.8	48.5	8.0	12.8
`lcc`	9.2	9.2	33.5	6.0	8.1
MSVC	10.1	8.0	39.2	6.4	6.7
MSVC /O2	4.5	5.4	20.5	4.2	4.0

Timing Ratios	bubble.c	perm.c	puzzle.c	queens.c	towers.c
GBURG/`lcc`	1.8	1.3	1.4	1.3	1.6
GBURG/MSVC	1.6	1.5	1.2	1.3	1.6
GBURG/MSVC/O2	3.7	2.2	2.4	1.9	3.2

Table 2: Small Benchmark Timings (1000 executions each)

7 Experimental Results

GBURG-generated code generators are tiny and produce good code quickly. A complete x86 code generator can be as small as 8 KB of x86 code and data, and it can generate x86 code at 3.6 MB/sec on a 266 MHz P6 machine. (The machine specification for the x86 is complete—it includes all rules that fit within the GBURG constraints and it was not abbreviated to reduce size.) For all our experiments, the ANSI C compiler `lcc` generates the LVM code [FH95].

For code generator size and speed tests, we compare tree-pattern matching code generators created by GBURG, `iburg`, and `burg`. `iburg` and `burg` are both two-pass systems that rely on tree-pattern matching and dynamic programming to select instructions; they differ in that `burg` does dynamic programming at compile-compile time and `iburg` does it at compile time. Both `iburg` and `burg` have been tuned for speed. Table 1 compares the sizes and speeds of x86 code generators generated from the same base grammars. All systems were compiled with Microsoft Visual C 5.0 with maximum speed optimizations enabled. All timings were done on a 266 MHz P6, by translating 52 KB of LVM code to x86 machine code 1000 times. The code generator sizes include all initialized text and data attributable to the code generators.

To test the conjecture that GBURG's code generators can compile and execute programs faster than typical interpreters, we have compiled and and run a few programs from the Stanford Benchmark Suite—programs comparable in size to small applets. Table 2 compares the run times using four different compilation strategies: using Microsoft Visual C 5.0 with maximum optimization, using Microsoft Visual C 5.0 with its default options, using `lcc`, and using the GBURG-generated LVM-to-x86 translator. The three compiler systems each generate stand-alone applications that execute the individual test 1000 times. The GBURG system reads LVM code from a file and then translates it to native code and executes it 1000 times. (Please note that it translates the LVM code for each iteration—the translation cost is *not* amortized over many runs.) `lcc` generated the LVM, which means that the GBURG vs. `lcc` is the closest to an apples-to-apples comparison.

Even when compared with a highly optimizing compiler that is charged *nothing* for its compilation time, GBURG-generated compile+execution times are more than twice as fast as the expected 10X slowdown from interpretation. Furthermore, when compared with native programs that are derived from the same source (i.e., `lcc`) as the LVM code, the slowdowns are

a remarkable factor of 2X. Thus, given that these code generators fit entirely in 8 KB, they are suitable replacements for interpreters on space-limited devices.

References

[AGT89] 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.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, 1996.

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[DF84] 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.

[Eng96] 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*, pages 160–170, May 1996.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, Redwood City, California, 1995.

[FHP92a] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[FHP92b] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

[GG78] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Proceedings of the 5th Annual Symposium on Principles of Programming Languages*, pages 231–240, 1978.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice-Hall, Inc., 1983. ISBN 0-13-449777-5.

[HO82] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[KR88] Brian W. Kernighan and Dennis M Ritchie. *The C Programming Language.* Prentice-Hall, Inc., second edition, 1988. ISBN 0-13-110370-9.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997. ISBN 0-201-63452-X.

[PLG88] Eduardo Pelegri-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages*, pages 294–308, New York, 1988. ACM.

[Pro95] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[PW96] Todd A. Proebsting and Benjamin R. Whaley. One-pass, optimal tree parsing — with or without trees. In *1996 International Conference on Compiler Construction*, pages 294—308, April 1996.