Denali: A Goal-directed Superoptimizer

Rajeev Joshi Greg Nelson Keith Randall[*]

Compaq Systems Research Center
130 Lytton Ave, Palo Alto, CA

ABSTRACT

This paper provides a preliminary report on a new research project that aims to construct a code generator that uses an automatic theorem prover to produce very high-quality (in fact, nearly mathematically optimal) machine code for modern architectures. The code generator is not intended for use in an ordinary compiler, but is intended to be used for inner loops and critical subroutines in those cases where peak performance is required, no available compiler generates adequately efficient code, and where current engineering practice is to use hand-coded machine language. The paper describes the design of the superoptimizer, and presents some encouraging preliminary results.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Theory

Keywords

superoptimizer, optimizing compiler

1. INTRODUCTION

1.1 Goals

Automatic code generation is not a young subject, but after all these decades it still happens in many programming projects that for some portion of the program, the code generated by the best compiler available is not adequately efficient. When this happens, current engineering practice is to find a senior engineer with intimate knowledge of the relevant processor architecture and assign this unlucky individual the task of coding the relevant portions of the program in machine language by hand. This generally does produce the required efficient code, but since senior engineers have many pressing demands on their time, it is an expensive way to get the job done, and software productivity would be increased if automatic code generation could match or beat the best code of the machine language guru.

The performance improvement to be expected by hand-generating machine code is more a matter of folklore and rule of thumb than of documented engineering experience. One report in the literature (on the performance of the Firefly RPC system) states that a factor of three is typical [19]. In any case, the performance improvement is sufficient to cause many implementers to use hand-generated code in parts of their systems, and the cost of this practice only begins with the time required to write the machine code in the first place. The additional cost is that the process of porting the system to a new architecture is no longer automatic. If the Denali approach fulfills its potential, both costs will be eliminated since the code fragments will be generated automatically.

This problem is not one of automating the invention of algorithms or the design of loops, which even we shy away from, but the much easier problem of automating the tedious backtracking search to find a straight-line machine code sequence that computes a given vector of expressions in the minimum number of cycles, achieving multiple issue whenever possible, respecting the latency constraints of memory and the various functional units, doing an optimal job of common subexpression elimination, and so on.

This is not a busy research area, perhaps for the following reason: Most programmers spend most of their day executing an edit-compile-debug loop; and most automatic code generators (even so-called "optimizers") are designed to run as part of a compiler that is used in this manner, and therefore are constrained by the requirement that they generate hundreds, thousands, or millions of instructions per second. Such a code generator has little hope of generating code that will be good enough for our purpose. Consequently a great deal is known about quickly generating indifferent code, and very little is known about generating optimal code, which is our goal. Indeed, the label "optimization" has been given to a field that does not aspire to optimize but only to improve. This misnomer presented a difficulty to Henry Massalin, who invented the only other code generation technique we know of that aimed at our goal [14]: the difficulty was that if Massalin called his system an optimizer, people would assume that it was only a code improver. So Massalin called his

[*]Randall's current affiliation is Google Inc., Mountain View, CA

system a superoptimizer. In our title, we have adopted his nomenclature.

Massalin's approach was bold and creative, and a striking example of Ken Thompson's principle "When in doubt, use brute force". His superoptimizer performed an exhaustive enumeration of all possible code sequences in order of increasing length. For each sequence, the superoptimizer executed the sequence against a suite of tests, and a sequence that passed all tests was printed as a candidate.

Massalin's original implementation was for the 68000 only, but his method has been employed by Granlund and others to produce superoptimizers for other machines[7].

Massalin's work was path-breaking, but the problem is important enough that we decided to explore an alternative search strategy that we hope will scale better than exhaustive enumeration. As yet, we have only preliminary results for our system, but they are consistent with our expectation that Denali will improve on Massalin's superoptimizer in several ways:

- Massalin's approach finds the shortest program. On Massalin's 68000, the shortest would also be the fastest, but on multiple-issue architectures this need not be so.

- To require the user to prepare a bank of tests for each fragment of code to be generated is painfully onerous. By contrast, the input to Denali is similar to the input to a conventional code generator.

- Passing tests is not the same as being correct, so the output of Massalin's superoptimizer must be studied carefully to check that it is correct. By contrast, the output of Denali is correct by design.

- Since executing random code could have undesirable effects, the enumeration of candidate sequences must be limited to some repertoire of sufficiently safe instructions, so that executing the candidates doesn't crash the program or interfere with the code generator itself. It would appear from his paper that Massalin generated only register-to-register computations that performed no accesses to memory. Denali has no such limitation.

- Brute-force enumeration of all code sequences is glacially slow. Massalin succeeded in finding impressive short code sequences, but his method seems to be limited to sequences of around half-a-dozen instructions. Denali substitutes goal-directed search for brute-force enumeration, for an enormous gain in efficiency. Our prototype is able to generate a near-optimal sequence of thirty-one instructions in around four hours.

We have been experimenting with the idea behind Denali for a little over a year. Our current prototype generates code for the Alpha EV6, the latest publicly available implementation of the Compaq Alpha processor. The prototype consists of some 15,000 lines of C and Java and some 700 lines of axioms. We are currently making the changes necessary to target the Intel Itanium architecture. It appears that this shift will not require any radical changes (and the changes will mostly be to the axioms), but this preliminary note will describe the Alpha version of Denali only.

1.2 The search principle

It may seem to some readers that an automatic theorem prover is an unlikely engine to use as a code generator. In an effort to correct this misperception, we would like to emphasize an important principle.

The search principle: A refutation-based automatic theorem prover is in fact a general-purpose goal-directed search engine, which can perform a goal-directed search for anything that can be specified in its declarative input language. Successful proofs correspond to unsuccessful searches, and vice-versa.

A refutation-based prover is a prover that attempts to prove a conjecture C by establishing the unsatisfiability of its negation $\neg C$. The search principle is not true of all refutation-based provers, but it is true of an important kind of prover with which our research laboratory has much experience [15, 2].

As an example of the search principle, to search for errors in a computer program, we express in formal logic the conjecture that there are no errors, and give this conjecture to a refutation-based automatic theorem prover. If the proof succeeds, the search for errors has failed. If the proof fails, embedded within the failed proof is the error (or errors) that caused the proof to fail, which can be extracted and presented to the user of the program checker. This approach has been used by the Extended Static Checking research project [3, 13, 6].

A second example of the principle is Tracy Larrabee's hardware test vector generator, which finds a test vector for a given fault by refuting the conjecture that no test vector for the fault exists, using a satisfiability solver [11].

1.3 The obvious approach

The search principle suggests an obvious way to build the code generator we desire. To generate optimal code for a program fragment P, we express in formal logic a conjecture of the following form:

conjecture No program of the target architecture computes P in at most 8 cycles.

We then submit the conjecture to an appropriate automatic theorem prover. If the proof succeeds, then 8 cycles are not enough, and we try again, with, say, 16 cycles. On the other hand, if the proof of the conjecture fails, then embedded in the failed proof is an (at most) 8-cycle program that computes P. We extract that program, and try again with 4 cycles. Continuing with binary search, we eventually find, for some K, a K-cycle program that computes P, together with a proof that $K - 1$ cycles are insufficient: that is, an optimal program to compute P on the given architecture. (Since the costs of the probes are far from constant, binary search might not be the best strategy, but we have not explored alternatives.)

It is easier to describe the obvious approach than to make it work. If carried out naively, the conjectures submitted to the prover become unwieldy. Suppose, for example, that we proceeded by defining in formal logic the two functions

exec(M, i) The machine state produced by executing the machine code sequence M on the input state i.

meaning(P) The meaning of a program (or program fragment) P as a function from input states to output states.

Then the conjecture that no machine code sequence M computes a given program fragment P in K cycles becomes:

$$\begin{array}{l}\neg(\exists\, M : M \text{ is a } K\text{-cycle program} : \\ \quad (\forall\, i : i \text{ is an input state} : \\ \quad\quad \mathbf{exec}(M, i) = \mathbf{meaning}(P)(i)))\end{array}$$

Conjectures of this form are daunting for two reasons: First, the universal quantifier nested within the existential quantifier is difficult for automatic theorem provers to handle. Second, the many cases in the definitions of **exec** and **meaning** tend to lead automatic theorem provers into a morass of case analyses.

1.4 The Denali Approach

Luckily, the alternating quantifiers and the full definitions of **exec** and **meaning** are unnecessary. For a sufficiently simple program fragment P, the equivalence of M and P for all inputs is essentially the universal validity of an equality between two vectors of terms, the vector of terms that M computes and the vector of terms that P specifies must be computed. Such equivalences can be proved using matching, a well-understood automatic theorem-proving technique. For example, suppose that we want to prove that the program fragment `reg6 := 2*reg7` is equivalent, for all inputs, to the one-instruction machine program

```
leftshift reg7,1,reg6
```

(We use a three-operand assembly language with the destination given in the third argument.) Denali's matcher will prove this equivalence by instantiating the algebraic identity

$$(\forall\, x : 2 * x = x << 1)$$

with the instantiation $x :=$ `reg7`. The algebraic identity shown above is an example of a collection of identities used by Denali and expressed in declarative symbolic form (as a Denali *axiom*). We will describe axioms in more detail in section 4.

So, instead of introducing an explicit quantifier over all inputs, we accept the limitation that the only proofs of equivalence for all inputs that we will consider between a program fragment and a machine code sequence are proofs by matching. If this limitation caused a valid proof of equivalence to be missed, then Denali might miss the most efficient way of computing some term, and its output might fail to be optimal, but its output would still be correct.

For the kinds of conjectures that we encounter in code generation, it turns out, perhaps surprisingly, that, once the proof of equivalence for all inputs is handled by matching, all that remains of the proof can be handled by purely propositional reasoning, which boils down to boolean satisfiability solving (SAT solving). The matcher finds all possible ways of computing the result, and the SAT solver selects from these the fastest, considering common subexpressions, delay constraints of the architecture, multiple issue constraints, and so forth. Roughly speaking, the matcher solves the undecidable part of the optimal code generation problem, and the satisfiability solver solves the NP-complete part. It is an effective division of labor. Our current (very limited) experience suggests that in practice, the most expensive step is the satisfiability solver. But the architecture of Denali separates this solver so effectively from the rest of the code generator that we can easily substitute the current champion satisfiability solver and use it instead of its predecessor. Indeed, as short as the project's history is, we have already made several substitutions of this sort. The solver used by default by our current prototype is the CHAFF SAT solver [1].

The essential novelty of Denali is the combination of the two phases and their application to the superoptimization problem. We do not claim novelty of either phase by itself: the E-graph matching phase applies techniques that have been used in automatic theorem-proving for ten or fifteen years [2] (although the earliest accounts of the technique failed to publish the matching algorithms [4, 15, 17]), and the satisfiability search phase can be viewed as an application to code generation of the recent ideas of Henry Kautz and Bart Selman about the AI planning problem [9, 8, 10].

The remaining sections of this paper describe in order

- the input to Denali,
- the translation strategy,
- the axioms used by Denali,
- how Denali's matcher works,
- how the propositional constraints are generated,
- some additional issues whose solutions are beyond the scope of this short paper, and finally
- some preliminary results.

2. THE INPUT TO DENALI

The input to Denali is a program in a language with a low-level machine model, similar to C or assembly language. The language includes higher-level control constructs, such as conditionals and loops. In addition, the language includes features by which the programmer can indicate that certain loops are to be unrolled or that certain memory references are likely to miss in the cache, or that the code generator should trust the programmer that certain conditions hold at certain control points in the program. The language is not intended for writing programs of any size directly; it is intended to be used for writing the body of an inner loop, for example, or for writing short subroutines. Figures 3 and 5 contain examples.

3. THE TRANSLATION STRATEGY

The Denali prototype translates its input into an equivalent assembly language source file. The translation strategy is as follows: Each procedure in the input is converted into a set of *guarded multi-assignments*, which are the inputs to the crucial inner subroutine of the code generator.

A *guarded multi-assignment* (or GMA) is determined by a sequence *targets* of designators (also called L-values), an equally long sequence *newvals* of expressions (also called R-values), a boolean expression G called the *guard*, and an *exit label* L. The meaning of such a GMA is:

```
if $G$ then
    ($targets$) := ($newvals$)
else
    goto $L$
end
```

We generally write $G \rightarrow (\mathit{targets}) := (\mathit{newvals})$ to denote this GMA, leaving the exit label to be determined by the

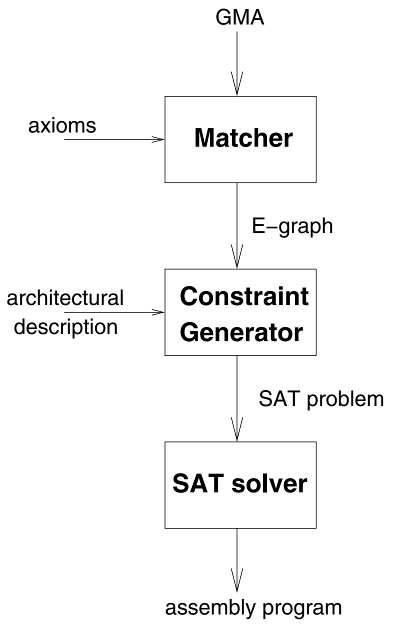

Figure 1: Generating code for a GMA consists of matching followed by satisfiability solving.

context. For example, before unrolling, the GMA for the inner loop of a copy routine might be:

$$\mathtt{p} < \mathtt{r} \rightarrow (\mathtt{*p}, \mathtt{p}, \mathtt{q}) := (\mathtt{*q}, \mathtt{p} + 8, \mathtt{q} + 8)$$

with an exit label appropriate for an exit from the copy loop. Denali translates the pointer references in this GMA into accesses to the memory M:

$$\mathtt{p} < \mathtt{r} \rightarrow (\mathtt{M[p]}, \mathtt{p}, \mathtt{q}) := (\mathtt{M[q]}, \mathtt{p} + 8, \mathtt{q} + 8)$$

Next, because our automatic theorem prover treats entire arrays as values, the update to M[p] is transformed by Denali into an update to M:

$$\mathtt{p} < \mathtt{r} \rightarrow (\mathtt{M}, \mathtt{p}, \mathtt{q}) := (\mathtt{store(M}, \mathtt{p}, \mathtt{M[q]}), \mathtt{p} + 8, \mathtt{q} + 8)$$

The Denali prototype converts each procedure in its input into a set of GMAs, and then uses the crucial inner subroutine to convert each GMA into near-optimal machine code, using the search principle as modified to rely on matching and satisfiability search. Our efforts have been concentrated on improving this inner subroutine rather than on improving the factorization of a procedure body into a collection of GMAs, where many conventional techniques could usefully be applied (including register allocation, which the current prototype ignores). Figure 1 illustrates the crucial inner subroutine that translates a single GMA into optimal code in two phases: matching and satisfiability search.

The guarded multi-assignment is a special case of the *Register Transfer List* (RTL) [18]. The extra generality of the RTL is the possibility of different guards for the different updates of the multi-assignment. This extra generality seems to be useful when RTLs are used in machine descriptions, but for our purpose of describing the input to the code generator, we have not encountered any examples where the extra generality of the RTL was wanted.

In the overall flow of figure 1, the matcher converts the GMA into an E-graph, which is a data structure that compactly represents all possible ways of computing the goal terms. In addition to the GMA itself, the matcher takes as input a set of axioms about the operators that are computable by the target architecture. It remains to be determined whether any of the ways of computing the goal terms can be computed by the target architecture within the cycle budget K. The constraint generator formulates this remaining question as a boolean satisfiability problem. In addition to the E-graph, an important input to the constraint generator is an *architectural description*, which includes tables specifying which functional units can execute which instructions, and a table of latencies of the various ALU operations, and, in the case of multiple register banks, of the latencies of the data paths connecting different banks. Finally, a conventional boolean satisfiability solver is used to find a solution or determine that no solution exists. The matching step is performed only once per GMA; the constraint generation and satisfiability solution steps are repeated for various cycle budgets until an optimal machine program is found.

The matcher and the constraint generator are the subjects of sections 5 and 6 of this report.

4. AXIOMS

As mentioned above, axiom files record in declarative form facts about the operations relevant to efficient code generation. We find that we use two kinds of built-in axioms: *mathematical axioms*, which provide facts about functions and relations that would be useful in describing many different target architectures, and *architectural axioms*, which define or describe operations relevant to a particular target architecture. In addition to built-in axioms, Denali allows *program-specific axioms*.

We next give some examples of mathematical axioms, followed by examples of architectural axioms for the Alpha instruction set architecture. We have taken these examples from Denali's standard axiom files. For expository purposes, we have made two changes: converting from LISP-like parenthesized expressions into traditional mathematical notation, and suppressing *patterns*, which determine the instances of universally quantified axioms that will be introduced by the matcher. Readers who are interested in the details will find examples of axioms expressed in Denali's LISP-like input syntax in section 8.

The mathematical function add64 denotes integer addition modulo 2^{64}. Three representative mathematical axioms postulate that add64 is commutative, associative, and has identity 0:

$$(\forall\, x, y :: \mathtt{add64}(x, y) = \mathtt{add64}(y, x))$$
$$(\forall\, x, y, z :: \mathtt{add64}(x, \mathtt{add64}(y, z)) = \mathtt{add64}(\mathtt{add64}(x, y), z))$$
$$(\forall\, x :: \mathtt{add64}(x, 0) = x)$$

Denali's mathematical axioms include fundamental properties of the functions select and store that represent reads and writes of arrays. One of these is the *select-store axiom*, which says that writing element i of an array a doesn't change any element with an index j different from i:

$$(\forall\, a, i, j, x\ :: i = j \\ \vee\ \mathtt{select}(\mathtt{store}(a, i, x), j) = \mathtt{select}(a, j))$$

Denali's mathematical axioms also define the functions `selectb` and `storeb`, which are like `select` and `store`, except that they treat integers as arrays of bytes: $\texttt{selectb}(w, i)$ denotes byte i of word w.

A typical architectural axiom is simply an equality that defines some operation of the target architecture in terms of mathematical functions. For example, the Alpha has the assembly instructions `extbl`, `insbl` and `mskbl` ($\texttt{extbl}(w, i)$ "extracts" byte i of longword w; $\texttt{insbl}(w, i)$ creates a longword with byte i equal to the least significant byte of w and other bytes zero; $\texttt{mskbl}(w, i)$ creates a copy of longword w with byte i set to zero). These are defined to Denali by:

$$(\forall\, w, i :: \texttt{extbl}(w, i) = \texttt{selectb}(w, i))$$
$$(\forall\, w, i :: \texttt{insbl}(w, i) = \texttt{selectb}(w, 0) \texttt{ << } 8 * i)$$
$$(\forall\, w, i :: \texttt{mskbl}(w, i) = \texttt{storeb}(w, i, 0))$$

As these examples illustrate, we usually use the same name for an instruction and for the function that it computes.

The Denali prototype automatically loads a file of mathematical axioms and a file of architectural axioms for the Alpha EV6. These files have grown over the course of our project, and will need to grow further before they are satisfactory. Currently, there are 44 mathematical axioms, comprising 127 source lines, and 275 Alpha axioms, comprising 637 lines. Together, these are the *built-in* axioms of our prototype.

In addition, a Denali source program may include axioms that are not important enough to include in the built-in axiom files, but are useful to the compilation of that particular program. Such axioms can be used as a powerful substitute for conventional macros. For example, one of our tests requires computing the ones complement checksum of an array of 16-bit integers (see section 8). For convenience, this program defines its own addition operator `add` by means of the following axiom:

$$(\forall\, x, y :: \texttt{add}(x, y) = \texttt{add64}(\texttt{add64}(x, y), \texttt{carry}(x, y)))$$

The `carry` operation used in the definition of `add` is also defined locally in the program by the following two axioms:

$$(\forall\, x, y :: \texttt{carry}(x, y) = \texttt{cmpult}(\texttt{add64}(x, y), x))$$
$$(\forall\, x, y :: \texttt{carry}(x, y) = \texttt{cmpult}(\texttt{add64}(x, y), y))$$

The definition by two axioms instead of one gives the code generator the freedom to compute $\texttt{carry}(a, b)$ by comparing $\texttt{add64}(a, b)$ with either a or b.

5. MATCHING

The purpose of the matching phase of Denali is to use the axioms to identify all of the possible ways in which the expressions in the GMA can be computed. Since the number of ways may be enormous (exponentially larger than the size of the expressions) it is important to choose a data structure carefully. The matching phase of Denali uses a data structure called the *E-graph*. Early descriptions of the E-graph include the PhD thesis of one of the authors (Nelson)[15], a journal paper by Nelson and Oppen [17], and the crucial Downey-Sethi-Tarjan congruence closure algorithm [4]. None of these papers say much about the matching algorithm. Denali uses the same matching algorithm as the automatic theorem prover Simplify. This matching algorithm is described in an upcoming research report [2].

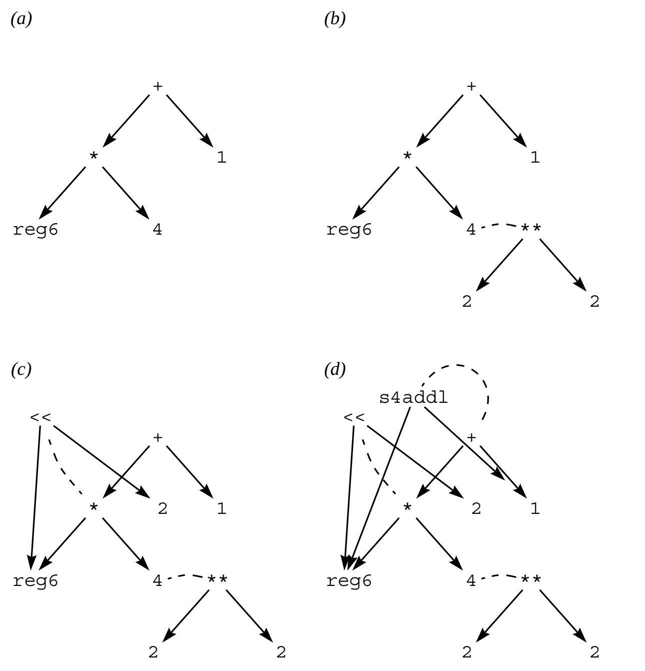

Figure 2: Solid arrows represent term DAG edges and dashed arcs represent equivalences in this illustration of matching in the E-graph

An E-graph is a conventional term DAG augmented with an equivalence relation on the nodes of the DAG; two nodes are equivalent if the terms they represent are identical in value. Hence the value of an equivalence class can be computed by computing any term in the class; having selected a term in the class, the values of each argument of the term likewise can be computed by selecting any term equivalent to the argument term, and so forth. Thus an E-graph of size $O(n)$ can represent $\Theta(2^n)$ distinct ways of computing a term of size n.

The machine code for a GMA must evaluate the boolean expression that is the guard of the GMA, and must also evaluate the expressions on the right side of the assignment statement, as well as the addresses of any targets that are not registers (that is, address arguments to `select` and `store`). Let us call all these expressions the *goal* expressions, since the essential goal of the required machine code is to evaluate them.

Typical GMAs have several goal terms, but Figure 2 illustrates Denali's matcher for the artificially simplified situation of a single goal term, namely the term `reg6*4+1`, which we have chosen to illustrate several points about matching. The first step in the matching phase is to construct an E-graph that represents all the goal terms. Figure 2(a) shows the initial E-graph of our simple example. It is a conventional term DAG: that is, a term of the form $f(t_1, t_2, \ldots, t_n)$ is represented by a node labelled f with an outgoing sequence of edges pointing to the nodes that represent the t's. If no matching were performed at all, so that Figure 2(a) were the final E-graph, then the only way to compute the goal term would be by a multiply followed by an add.

The matcher repeatedly transforms the E-graph by instantiating a relevant axiom and asserting the instance in the E-graph. This is repeated until a quiescent state is reached in which the E-graph records all relevant instances of axioms. In the case of our example, the first relevant fact that we will add to the graph is the fact $4 = 2^2$. When this fact is added, the E-graph is changed by adding a new node to represent the term 2^2 (or $2 \texttt{**} 2$) and adding this new node to the equivalence class of the existing node for "4". Figure 2(b) shows the result of this transformation. (We use dashed edges to connect nodes that are equivalent.) Of course, the Alpha does not have an instruction for computing `**`, so this match has not directly introduced any new ways of computing the goal term: if matching terminated with the E-graph of Figure 2(b), the only way to compute the goal term would be by the same multiply and add sequence available already in the initial graph. But this does not mean the change to the E-graph was useless, because it enables new matches. Specifically, matching now continues by finding a relevant instance of the fact

$$(\forall k, n :: k * 2^n = k \texttt{<<} n)$$

namely the instance with $(k, n) := (\texttt{reg6}, 2)$. (An ordinary matcher would fail to match the pattern $k * 2 \texttt{**} n$ against the term-DAG node $\texttt{reg6}*4$ because the node labelled "4" is not of the form 2^n, but an E-graph matcher will search the equivalence class and find the node $2 \texttt{**} 2$ and the match will succeed.) The resulting E-graph is shown in Figure 2(c). If matching were terminated at this point, then in addition to the multiply-add sequence there would be a shift-and-add sequence (which is faster and therefore would probably be selected). Finally, the Alpha contains an instruction called `s4addl` which scales by four and adds. The background facts for Denali therefore include the architectural axiom

$$(\forall k, n :: k * 4 + n = \texttt{s4addl}(k, n)) \quad .$$

When the matcher instantiates this with $(k, n) := (\texttt{reg6}, 1)$ and updates the E-graph, the result is the graph shown in Figure 2d. This adds a new possibility for computing the goal term (superior to both of the other possibilities) using a single `s4addl` instruction.

Here are three comments about this example.

First, the order in which the matches would occur in practice might very well be different than the order described: `s4addl` could have been introduced immediately. However, the `<<` node could not be introduced until the equality of 4 with $2 \texttt{**} 2$ was introduced.

Second, we contrast E-graph matching with conventional matching. Many conventional matchers are rewriting engines, in the sense that they directly rewrite a term into a new form, recursively rewriting subexpressions before rewriting a root expression. For example, they might rewrite $n*2$ into n`<<1`. Such a rewriting engine would be unlikely to rewrite 4 as `2**2`, since the latter term is not an efficient way to compute the former. Similarly, a rewriting engine that produced the fairly efficient `reg6<<2` might miss the most efficient version with `s4addl`, since the pattern for the fact involving `s4addl` most naturally involves multiplication by four, not left-shifting by two. In general, to reach the optimal version by a sequence of elementary rewrites may require rewriting some subterms in ways that reduce efficiency rather than improve it, and, in general, a transformation that improves efficiency may cause the failure of subsequent matches that would have produced even greater gains. These are well-known and thorny problems for rewriting engines. The E-graph doesn't suffer from these problems, since, instead of rewriting A as B, it records $A = B$ in its data structure, leaving both A and B around, where they can be used both for future matching and as candidates for the final selection of instructions.

A third comment is that the attractive features of the E-graph approach mentioned in the second comment are not without their price. Matching in an E-graph is more expensive than matching a pattern against a simple term DAG. Also, many matches are required to reach quiescence, and the quiescent state may be quite a large E-graph. For example, Denali's matcher uses the commutativity and associativity of addition to find more than a hundred different ways of computing $a + b + c + d + e$. Nevertheless, Denali seems to be efficient enough to be useful.

In our description of Denali's matcher, we have so far considered only facts that are (quantified or unquantified) equalities between terms (that is, facts of the form $T = U$). Two other kinds of facts that the matcher uses are (quantified or unquantified) *distinctions* and *clauses*. As with equalities, quantified distinctions and clauses are transformed into the corresponding unquantified kind of fact by finding heuristically relevant instances, so it suffices to explain how Denali uses unquantified distinctions and clauses.

A (binary) distinction is a fact of the form $T \neq U$ for two terms T and U. Binary distinctions are the only kind of distinction that we will consider in this preliminary report. A distinction $T \neq U$ is asserted in the E-graph by recording the constraint that the equivalence classes of T and of U are *uncombinable*.

Equalities and Distinctions are collectively called *literals*. The third kind of fact that Denali uses is a *clause*, which is a disjunction ("or") of literals, that is, a fact of the form

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

where the L's are literals. An unquantified clause is used by recording it in a data structure and then continuing as follows. Whenever any of its literals becomes *untenable*, the untenable literal is deleted from the recorded clause. Furthermore, if the deletion of the untenable literal from a recorded clause leaves the clause with a single literal, then that lone literal is asserted. An equality $T = U$ is untenable if the equivalence classes of T and of U have been constrained to be uncombinable. A distinction $T \neq U$ is untenable if T and U are in the same equivalence class.

We conclude this section with an example that illustrates how the matcher uses clauses and distinctions. If a GMA involved storing some value (say x) to address p and then loading from address $p + 8$, then the E-graph would include the term

$$\texttt{select}(\texttt{store}(\texttt{M}, p, x), p + 8)$$

The presence of this term would cause the body of the select-store axiom (see section 4) to be instantiated by $(a, i, j) := (\texttt{M}, p, p + 8)$, causing the matcher to make a record of the unquantified clause

$$\begin{array}{l} \quad p = p + 8 \\ \vee \ \texttt{select}(\texttt{store}(\texttt{M}, p, x), p + 8) = \texttt{select}(\texttt{M}, p + 8) \end{array}$$

By mechanisms that we will not describe in this preliminary report, the literal $p = p{+}8$ will be discovered to be untenable and deleted, leading to the assertion of the equality

$$\texttt{select}(\texttt{store}(\texttt{M}, p, x), p + 8) = \texttt{select}(\texttt{M}, p + 8)$$

The presence of this equality in the E-graph gives the code generator the option of doing the load and store in either order.

The matching phase uses identities and reasons about equalities, distinctions, and clauses. When the matching phase is complete, the final equivalence relation of the E-graph is all that matters: the distinctions and clauses used on the way are not used in any way by the satisfiability search phase that follows.

6. SATISFIABILITY SOLVING

After the matcher has reached a quiescent state, the E-graph represents all possible ways of computing the terms that it represents. (More precisely, this is true if the axioms include a complete axiomatization of the first order theory of the relevant operations and if the matching phase is allowed to run long enough, and if the heuristics that are designed to keep the matcher from running forever don't mistakenly stop it from running long enough. These caveats about the matcher are the first reason that we call Denali's output "near-optimal" instead of "optimal".) In order to obtain optimal code, it remains to formulate a conjecture of the form

No program of the target architecture computes the values of the goal terms within K cycles

where K is a specified literal integer constant. Happily, this can be formulated in propositional calculus, so that it can be tested with a satisfiability solver. The exact details are somewhat architecture-dependent, but the basic idea is simple. To illustrate the basic idea we assume a machine without multiple issue, so that at most one instruction is issued per cycle. We define a function to be a *machine operation* if some instruction of the target architecture is capable of computing the function. In addition to machine operations, the E-graph also includes non-machine operations that are allowed in the input (and in the axioms) but that cannot be computed by the machine directly. (The matching example in the previous section used the non-machine operation `**`, so that universal facts could be expressed conveniently.)

We define a term (that is, a node of the E-graph) to be a *machine term* if it is an application of a machine operation, and a non-machine term otherwise. The arguments to a machine term need not themselves be machine terms. (We are not interested yet in whether the term is computable from the inputs, only whether it could be computed if its arguments were available in registers.)

We introduce a number of boolean unknowns. Specifically, for each cycle i of the K cycles available, for each machine term T, and for each equivalence class Q, we introduce the following boolean unknowns:

$L(i,T)$: denotes that in the desired machine program, a computation of term T is Launched at the beginning of cycle i

$A(i,T)$: denotes that in the desired machine program, a computation of T is completed At the end of cycle i

$B(i,Q)$: denotes that the desired machine program has computed the value of equivalence class Q By the end of cycle i

Thus we have $(2m + Q)K$ boolean unknowns, where m is the number of machine terms and Q is the number of equivalence classes in the E-graph. In terms of these unknowns we can formulate the conditions under which a K-cycle machine program exists that computes all the goal terms.

There are five basic conditions. In writing these conditions, we use the dummy i to range over all cycle indices, T and U to range over machine terms in the E-graph, and Q to range over equivalence classes in the E-graph.

First, writing $\lambda(T)$ for the latency of the machine term T, that is, the number of cycles required to apply the root operator of T to its arguments, we observe that the interval of time occupied by the computation of T consists of $\lambda(T)$ consecutive cycles, leading to the following obvious relation between the cycle in which T's computation is launched and the cycle in which it is completed:

$$\bigwedge_{i,T} (L(i,T) \equiv A(i + \lambda(T) - 1, T))$$

Second, writing $\texttt{args}(T)$ for the set of equivalence classes of the top level arguments of a term T, we observe that an operation cannot be launched until its arguments are available, and therefore:

$$\bigwedge_{i,T} \; \bigwedge_{Q \in \texttt{args}(T)} (L(i,T) \Rightarrow B(i-1,Q))$$

Third, the only way to compute the value of an equivalence class Q by the end of cycle i is by computing the value of one of its machine terms T at the end of some cycle $j \leq i$ and therefore:

$$\bigwedge_{i,Q} \left(B(i,Q) \;\; \equiv \bigvee_{\substack{j \leq i \\ T \in Q}} A(j,T) \right)$$

Fourth, since we are assuming for simplicity a single-issue architecture, only one operation can be launched per cycle, so no two distinct machine terms T and U can be launched in any cycle i:

$$\bigwedge_{\substack{i,T,U \\ T \neq U}} (\neg L(i,T) \vee \neg L(i,U))$$

Fifth, letting $\mathcal{G}$ denote the set of equivalence classes of goal terms, each of these equivalence classes must be computed within K cycles. Numbering cycles from zero, that would be by the end of cycle $K - 1$:

$$\bigwedge_{Q \in \mathcal{G}} B(K-1, Q)$$

We need to continue adding constraints until the boolean unknowns are so constrained that any solution to them corresponds to a K-cycle machine program that computes the goal terms. More constraints are needed than we have shown so far, but we have shown enough to convey the essence of the approach.

For a fixed E-graph and a fixed cycle budget, the constraints are explicit propositional constraints on a finite set of boolean unknowns. The assertion that no K-cycle machine program exists is equivalent to the assertion that their conjunction is unsatisfiable, a conjecture that can be tested with a SAT solver, which is of all automatic theorem provers the one that most clearly satisfies the search principle, since it refutes the conjecture by finding explicit values for the L's, A's and B's that satisfy the constraints. The L's that are assigned true by the solver determine which machine operations are launched at each cycle, from which the required machine program can be read off.

We conclude this section with a few remarks about latencies. The Denali method requires that the latency $\lambda(T)$ of each term T be known to the code generator. For ALU operations, this requirement is not problematical, but for memory accesses it may at first seem to be a show-stopper. Certainly an ordinary code generator cannot statically predict the latencies of memory accesses. But the scenario in which Denali is designed to be used is not the scenario in which an ordinary compiler is used. The scenario is an inner loop or crucial subroutine whose performance is important enough to warrant hand-coding in machine language. In this scenario, the first step is to use profiling tools to determine which memory accesses miss in the cache. Having found this information, the programmer can communicate it to Denali using annotations in the Denali source program. Our engineering goal is to do at least as well as the machine language guru, who also designs her code on the basis of profile-generated assumptions about memory latencies.

Since the information gleaned from profiling is statistical, not absolute, we would still be in trouble if the correctness of the generated code depended on the accuracy of the memory latency annotations, but (precisely because caching makes memory latencies unpredictable statically) any reasonable modern processor (including both the Alpha and the Itanium) includes hardware to stall or replay when necessary, so that latency annotations are important for performance but not for correctness: the code generated will be correct even if the annotations are inaccurate. Thus we can expect some stalls or replay traps on the first few iterations of a Denali-optimized inner loop, but to the extent that statistical information about inner loops is reliable, the loop will soon settle into the optimal computation that was modelled by the boolean constraints. The statistical nature of profiling information is the second reason that we call Denali's output "near-optimal" instead of "optimal".

7. ADDITIONAL CONSTRAINTS

The satisfiability constraints in the previous section were simplified by the assumption of a single-issue machine, since the cycle index i could also be thought of as an index in the instruction stream. But the same approach easily accommodates a multiple instruction architecture where cycle indices and instruction indices both appear and must be carefully distinguished.

Some expressions (in particular, memory accesses) on the right side of a guarded multi-assignment may be unsafe to compute if the guard expression is false. Therefore Denali generates satisfiability constraints that force the guard to be tested before any such expressions are evaluated. It is straightforward to add additional propositional constraints on the boolean unknowns to enforce this order.

The expressions on the right side of a guarded multiassignment may use the same targets that it updates; for example,

$$(\mathtt{reg6}, \mathtt{reg7}) := (\mathtt{reg6} + \mathtt{reg7}, \mathtt{reg6}) \quad .$$

In this case, the final instruction that computes the $\mathtt{reg6} + \mathtt{reg7}$ may not be able to place the computed value in its final destination. In the worst case, we may be forced to choose between adding an early move to save an input that will be overwritten by the rest of the code sequence or computing a value into a temporary register and adding a late move to put it finally into the correct location. On multiple-issue architectures the choice between these two alternatives may make a difference to performance. Denali encodes the choice into the boolean constraints where it becomes just one more bit for the solver to determine.

The ordering of procedure calls is more constrained than the ordering of other operations, because in general, a procedure call is assumed to both modify and read the memory. This circumstance leads to additional constraints that we also encode in the propositional constraints, but we must leave the details for future papers.

Some instructions of some architectures compute multiple results into multiple registers. In this situation we model the instruction's operation as a machine operation that computes a tuple of the various results. We also introduce into the axiom files non-machine *projection* operations that extract the individual components of the tuple. The final E-graph will contain expressions that apply the composition of the machine operation followed by the projection function. Such a composition represents a way of computing the value by applying the instruction and using the result that corresponds to the projection function.

```
\proc byteswap4 : [ a : int ] -> int =
  \var r : int \in
    r := 0 ;
    r<0> := a<3> ;
    r<1> := a<2> ;
    r<2> := a<1> ;
    r<3> := a<0> ;
    \res := r ;
  \end
```

Figure 3: Envisioned program for 4-byte swap. `w<i>` denotes byte `i` of word `w`, that is, $\mathtt{selectb}(w, i)$. Our current prototype requires a parenthesized input syntax in the style of figure 6.

8. PRELIMINARY RESULTS

We have implemented a prototype of Denali in Java and C for the Alpha EV6, a quad-issue processor with multiple register banks and extra delays for moving values between banks, almost all of whose complexity is modeled by our code generator. All the experiments described in this section were carried out on a 667Mhz Alpha machine with 500MB of main memory.

We created many tests for our prototype; we also invited our colleagues to supply us with challenge problems.

One source of test problems are byte swap problems: the problem of reversing the order of the n lower bytes of a register. For $n = 4$, this was was a challenge problem given long ago by a product engineering group who supported a SPARC emulator running on the Alpha. Figure 3 shows a representation of the input program for the 4 byte swap given to Denali. Our prototype takes just over a minute to generate code for this problem. Less than 0.3 seconds is spent in the SAT solver. The sizes of the four SAT prob-

```
// Register Map: {a=$16, r=$1, \res=$0, 0=$31}
byteswap4:               #   assume  a = wxyz
  extbl   $16, 1, $2     # 0, U1  ; $2 = 000y
  insbl   $16, 3, $3     # 0, U0  ; $3 = z000
  nop                    # 0
  nop                    # 0

  insbl   $2, 2, $2      # 1, U1  ; $2 = 0y00
  extbl   $16, 3, $4     # 1, U0  ; $4 = 000w
  nop                    # 1
  nop                    # 1

  or      $4, $3, $3     # 2, L0  ; $3 = z00w
  extbl   $16, 1, $4     # 2, U1 (unused)
  extbl   $16, 2, $4     # 2, U0  ; $4 = 000x
  nop                    # 2

  insbl   $4, 1, $4      # 3, U0  ; $4 = 00x0
  or      $2, $3, $2     # 3, L0  ; $2 = zy0w
  nop                    # 3
  nop                    # 3

  or      $4, $2, $0     # 4, U0  ; $0 = zyxw
  ret     ($26)          # 4, L0
  nop                    # 4
  nop                    # 4
.end byteswap4
```

Figure 4: Generated EV6 assembly program for four byte swap. The unused instruction is necessary: if it were a `nop`, the following `extbl` instruction would be scheduled on the wrong cluster.

lems solved for this example range from 1639 variables and 4613 clauses for the 4-cycle refutation to 9203 variables and 26415 clauses for the 8-cycle solution. The 5-cycle EV6 code generated is shown in Figure 4. Each instruction in this program is annotated with the cycle number and functional unit (one of L0, L1, U0, U1) at which it is issued. To the best of our knowledge, this five cycle program is optimal. With some effort, we were able to coax the production C compiler to tie this result, giving it aggressive switches (`-fast -arch ev6`), and helpful input:

```
long r = ((a       & 0xff) << 24)
       | ((a >>  8) & 0xff) << 16)
       | ((a >> 16) & 0xff) <<  8)
       | ((a >> 24) & 0xff) ;
```

For the 5-byte swap problem, Denali does one cycle better than the C compiler. (In these experiments, the running time of the code produced by the C compiler was computed by hand.)

We attempted to compare Denali with the Alpha version of the GNU superoptimizer which is available on the web [5]. But our attempts were not very successful. Version 2.5 of the GNU superoptimizer seems to model the Alpha incompletely: It models the Alpha instruction set architecture, not the EV6. It is missing several opcodes. Finally, while we were able to generate five-instruction sequences, we were unable to generate longer sequences in an amount of time that we were willing to wait (several days).

The largest challenge program we have tackled so far with Denali is a packet checksum routine, which computes the 16-bit sum of a set of 16-bit integers, with wraparound carry. Three techniques are required to generate efficient code for this problem: loop unrolling, software pipelining (the computation in one loop iteration of a result that is used on the next iteration), and word parallelism (in the case of this example, using 64-bit addition instead of four 16-bit additions). The current Denali prototype implements loop unrolling. We have a design for software pipelining, but haven't implemented it yet. In the meantime, to make progress on this example, we hand-specified the required pipelining by introducing temporaries to carry intermediate values across loop iterations. As for word parallelism, we don't aspire to do this automatically: we contend this is better done by the programmer using Denali.

```
\op add : [ long, long ] -> long ;
\op carry : [ long, long ] -> long ;

\axiom (\forall [a b] add(a,b) = add(b,a)) ;
\axiom (\forall [a b c]
    add(a,add(b,c)) = add(add(a,b),c)) ;
\axiom (\forall [ a b]
    add(a,b) = \add64(\add64(a,b), carry(a,b))) ;

\axiom (\forall [ a b ]
    carry(a,b) = \cmpult(\add64(a,b), a)) ;
\axiom (\forall [ a b ]
    carry(a,b) = \cmpult(\add64(a,b), b)) ;

\proc checksum : [ ptr,ptrend : long* ] -> short =
  \var sum : long := 0 \in
    \do ptr < ptrend ->
          sum := add(sum, *ptr) ; ptr := ptr + 8
    \od ;
    sum := \extwl(sum, 0) + \extwl(sum,1)
          + \extwl(sum,2) + \extwl(sum,3) ;
    sum := \extwl(sum,0) + \extwl(sum,1) ;
    \res := \cast(sum, short)
  \end
```

Figure 5: Envisioned program for checksum.

Figure 5 shows the Denali input that we envision. Figure 6 shows the actual input to our current prototype for this problem. As remarked above, the input uses temporaries (v1, v2, v3, v4) to hand-specify software pipelining. This forces us to hand-specify the loop unrolling as well, so we cannot use Denali's `unroll` feature. Denali took about 4 hours to generate code for this program; the code for the loop body consisted of 10 cycles and 31 instructions.

In addition to the challenge problems above, we have used Denali on a matrix routine `rowop`, and on the problem of the least common power of 2 of two registers (in addition to a number of problems we invented for ourselves). Although few in number, these tests give us confidence that the Denali approach can provide peak performance on ALU-bound register-to-register computations. The experiments we have done to date on memory-bound computations (the checksum example, and some matrix loops) suggest that Denali will also be effective on these increasingly important problems. However, we won't venture a final conclusion on memory-bound computations until we have implemented software

```
; carry returns the carry bit resulting from the
; unsigned 64-bit sum of its arguments.
(\opdecl carry (long long) long)

(\axiom (forall (a b) (pats (carry a b))
    (eq (carry a b) (\cmpult (\add64 a b) a))))
(\axiom (forall (a b) (pats (carry a b))
    (eq (carry a b) (\cmpult (\add64 a b) b))))

; unsigned 64-bit carry-wraparound add
(\opdecl add (long long) long)

; associativity of add
(\axiom (forall (a b c) (pats (add a (add b c)))
    (eq (add a (add b c)) (add (add a b) c))))
(\axiom (forall (a b c) (pats (add (add a b) c))
    (eq (add a (add b c)) (add (add a b) c))))

; commutativity of add
(\axiom (forall (a b) (pats (add a b))
    (eq (add a b) (add b a))))

; implementation of add
(\axiom (forall (a b) (pats (add a b))
    (eq (add a b) (\add64 (\add64 a b) (carry a b)))))

; main procedure
(\procdecl checksum ((ptr (\ref long))
                     (ptrend (\ref long))) short
  (\var (sum1 long 0) (\var (sum2 long 0)
  (\var (sum3 long 0) (\var (sum4 long 0)
  (\var (v1 long (\deref ptr))
  (\var (v2 long (\deref (+ ptr 8)))
  (\var (v3 long (\deref (+ ptr 16)))
  (\var (v4 long (\deref (+ ptr 24)))
  (\semi
  (\do (-> (< ptr ptrend)
    (\semi
      (:= (sum1 (add sum1 v1)) (sum2 (add sum2 v2))
          (sum3 (add sum3 v3)) (sum4 (add sum4 v4)))
      (:= (ptr (+ ptr 32)))
      (:= (v1 (\deref ptr)))
      (:= (v2 (\deref (+ ptr 8))))
      (:= (v3 (\deref (+ ptr 16))))
      (:= (v4 (\deref (+ ptr 24)))))))

  (\var (c1 long) (\var (c2 long) (\var (c3 long)
  (\var (s1 long) (\var (s2 long) (\var (s long)
  (\semi
      (:= (s1 (+ sum1 sum2)))
      (:= (c1 (carry sum1 sum2)))
      (:= (s2 (+ sum3 sum4)))
      (:= (c2 (carry sum3 sum4)))
      (:= (s (+ s1 s2)))
      (:= (c3 (carry s1 s2)))
      (:= (s (+ (\extwl s 0) (+ (\extwl s 1)
            (+ (\extwl s 2) (\extwl s 3))))))
      (:= (s (+ (\extwl s 0) (+ (\extwl s 1)
            (+ c1 (+ c2 c3))))))
      (:= (\res (\cast short s))))))))))))))))))))
```

Figure 6: Actual input to current prototype for checksum program of figure 5.

pipelining and done more examples, which we plan to do in the next few months.

9. REFERENCES

[1] SAT Research at Princeton Home Page. CHAFF satisfiability solver. `http://www.ee.princeton.edu/~chaff`.

[2] David Detlefs, Greg Nelson, and James B. Saxe. A theorem-prover for program checking. Technical Report Research Report 178, Compaq Systems Research Center, 2002. In preparation.

[3] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.

[4] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *JACM*, 27(4), 1980.

[5] Torbjorn Granlund et al. GNU Superoptimizer FTP site. ftp://prep.ai.mit.edu/pub/gnu/superopt.

[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *Proceedings of the ACM Programming Language Design and Implementation (PLDI) Conference*, 2002.

[7] Torbjorn Granlund and Richard Kenner. Eliminating branches using a Superoptimizer and the GNU C compiler. *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 341–52, 1992.

[8] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[9] Henry Kautz and Bart Selman. Planning as satisfiability. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92)*, 1992.

[10] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[11] Tracy Larrabee. *Efficient generation of test patterns using boolean satisfiability*. PhD thesis, Stanford University, 1990. See also [12].

[12] Tracy Larrabee. Efficient generation of test patterns using boolean satisfiability. Technical Report 90/2, DEC Western Research Lab, February 1990.

[13] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000-002, Compaq Systems Research Center, 2000.

[14] Henry Massalin. Superoptimizer: a look at the shortest program. *Proceedings of the second international conference on architectural support for programming languages and operating systems*, pages 122–26, 1987.

[15] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979. A revised version was published as a Xerox PARC Computer Science Laboratory Research Report [16].

[16] Greg Nelson. Techniques for Program Verification. Technical Report CSL-81-10, Xerox PARC, 1981. This

report is out of print, but the author still has a couple of photocopies.

[17] Greg Nelson and Derek C. Oppen. Fast decision algorithms based on congruence closure. *JACM*, 27(2), October 1979.

[18] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, pages 172–188, 1998.

[19] Michael D. Schroeder and Michael Burrows. Performance of the Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, 1990.