30 July 2001

SRC Research Report **171**

Denali: a goal-directed superoptimizer

Rajeev Joshi
Greg Nelson
Keith Randall

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Compaq Systems Research Center

SRC's charter is to advance the state of the art in computer systems by doing basic and applied research in support of our company's business objectives. Our interests and projects span scalable systems (including hardware, networking, distributed systems, and programming-language technology), the Internet (including the Web, e-commerce, and information retrieval), and human/computer interaction (including user-interface technology, computer-based appliances, and mobile computing). SRC was established in 1984 by Digital Equipment Corporation.

We test the value of our ideas by building hardware and software prototypes and assessing their utility in realistic settings. Interesting systems are too complex to be evaluated solely in the abstract; practical use enables us to investigate their properties in depth. This experience is useful in the short term in refining our designs and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this approach, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical character. Some of that lies in established fields of theoretical computer science, such as the analysis of algorithms, computer-aided geometric design, security and cryptography, and formal specification and verification. Other work explores new ground motivated by problems that arise in our systems research.

We are strongly committed to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences, while our technical note series allows timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Denali: a goal-directed superoptimizer

Rajeev Joshi, Greg Nelson, and Keith Randall

30 July 2001

Abstract

This paper provides a very preliminary report on a research project that is just getting under way 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.

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.

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 "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 be able to 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 that we know of that aimed at our goal [8]:

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

Massalin's approach was bold and creative, and an amazing 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 several different CPU architectures [4].

Our system, Denali, improves on Massalin's method 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 constrast, 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 stores 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. And it seems he was willing to let his code generator run for a week. Denali substitutes goal-directed search for brute-force enumeration, for an enormous gain in efficiency. We expect an overnight run to be able to generate an optimal sequence of several dozen instructions.

We have been experimenting with the idea behind Denali for a little over a year. Our current prototype generates code for the most recent 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 very short and 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 [9, 1].

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 [2, 7].

A second example of the principle was Tracy Larrabee's 1990 dissertation research on a hardware test vector generator that would find a test vector for a given fault by refuting the conjecture that no test vector for that fault existed, using a propositional automatic theorem-prover [5].

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.

and 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.

1.4 The problem with the obvious approach

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:

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

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.

1.5 The solution to the problem

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, how do we prove that the program fragment `reg6 := 2*reg7` is equivalent, for all inputs, to the one-instruction machine program

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

(Where we assume 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`.

So, instead of introducing an explicit quantifier over all inputs, we will 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 a correct way to compute its input.

For the kinds of conjectures that we must prove or refute in code generation, it turns out, as the rest of this paper will explain, 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 [12].

The remaining sections of this paper describe in order

- The input to 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 very preliminary results.

2 The input to Denali

The input to Denali is a program in a low-level language fairly close to C. In addition to the usual low-level language constructs, 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 useful for writing the body of an inner loop, for example, or for writing short test programs. 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.

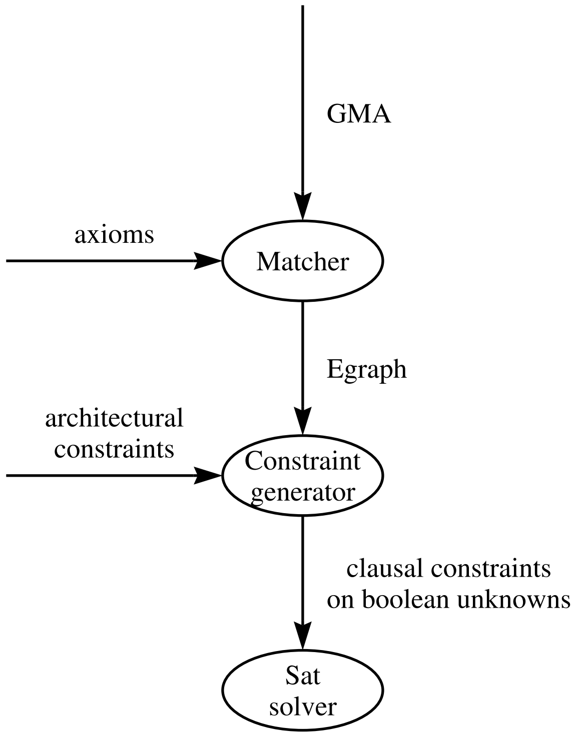

Figure 1: To compile a guarded multiassignment, the matcher instantiates the axioms of the theory of those operators that are computable by the target architecture, thereby determining all possible ways of using those operators to compute the goal terms of the GMA. The matcher determines all possible ways (all possible expression trees built from the operations available on the target architecture) of computing the goal terms. The constraint generator translates into a SAT problem the question of whether any of these ways can be computed within the cycle budget. Finally, the satisfiability solver finds a solution or determines that no solution exists.

A *guarded multi-assignment* (or GMA) is determined by a sequence *targets* of designators (also called or 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 the GMA is:

if G **then**
  $(\mathit{targets}) := (\mathit{newvals})$
else
  goto L
end

We generally write $G \to (\mathit{targets}) := (\text{newvals})$ to denote this GMA, leaving the exit label to be determined by the context. For example, before unrolling, the GMA for the inner loop of a copy routine might be:

$$\texttt{reg1} < \texttt{reg3} \to (\texttt{M}[\texttt{reg1}], \texttt{reg1}, \texttt{reg2}) := \texttt{M}[\texttt{reg2}], \texttt{reg1} + 8, \texttt{reg2} + 8$$

with an exit label appropriate for an exit from the copy loop. Because automatic theorem-provers treat entire arrays as values, the update to $\texttt{M}[\texttt{reg1}]$ is transformed in an early pass of Denali into an update to M, namely the update $\texttt{M} := \texttt{store}(\texttt{M}, \texttt{reg1}, \texttt{M}[\texttt{reg2}])$.

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. Figure 2 illustrates the crucial inner subroutine that translates a single GMA into optimal code in two phases: matching and satisfiability search.

The matcher emits an E-graph, which is a data structure that compactly represents all possible ways of computing the goal terms. It remains to be determined whether any of these ways can be computed by the target architecture within the cycle budget K. The constraint generator formulates this remaining question as a boolean satisfiability problem, and then a conventional boolean satisfiability solver is used to find a solution or determine that no solution exists. 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 the next two sections of this report.

3 Matching

The purpose of the matching phase of Denali is to use algebraic identities to identify all of the possible ways in which the expressions in the input 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 the *E-graph* data structure introduced

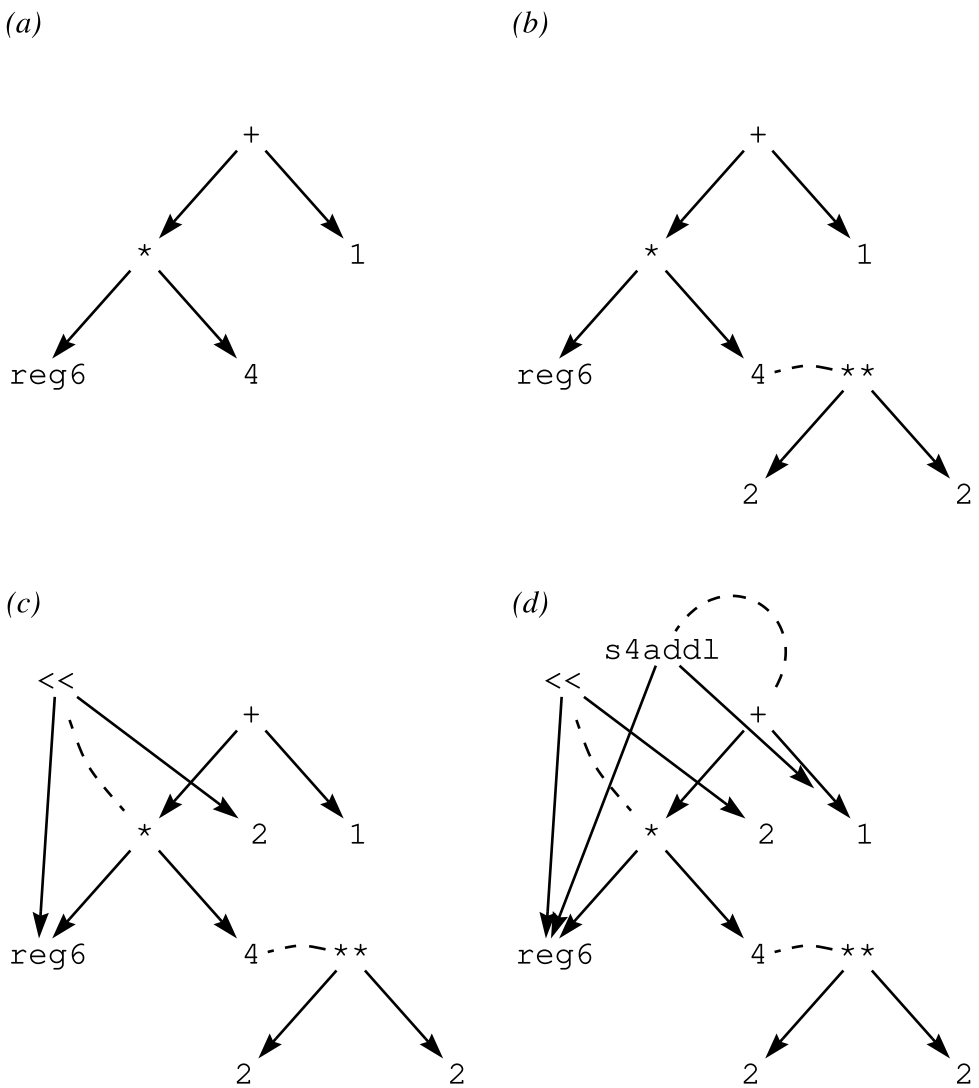

Figure 2: Solid arrows represent term DAG edges and dashed arcs represent equivalences in this illustration of matching in the E-graph.

in the Ph.D. work [9, 11] of one of the authors (Nelson), and refined extensively since then [1]. The crucial congruence closure algorithm used with the E-graph is the work of Tarjan and his colleagues [3].

An E-graph is a conventional term DAG augmented with an equivalence relation on the nodes of the DAG; where 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. 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 operations that are relevant for a particular version of Denali are the operations of the Denali source language together with the operations that can be computed by the target architecture of that version. The matcher relies on a *background* file that declares useful facts (axioms) about the relevant operations. The matcher repeatedly transforms the E-graph by instantiating a relevant and useful fact and asserting the fact in the E-graph. This is repeated until a quiescent state is reached in which the E-graph records all useful instances of all relevant facts in the background file. In the case of our example, the first relevant and useful 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 ** 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 and useful instance of the fact

$$\forall k, n :: k * 2^n = k \mathtt{<<} n$$

namely the instance with $(k, n) := (\mathtt{reg6}, 2)$. (An ordinary matcher would fail to match the pattern $k * 2 \mathtt{**} n$ against the term-DAG node $\mathtt{reg6} * 4$ 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 $2 \mathtt{**} 2$ 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 fact

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

When the matcher instantiates this with $(k, n) := (\mathtt{reg6}, 1)$ 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 this example 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 $2 \mathtt{**} 2$ 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 $n\mathtt{<<}1$. 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, and when it is painfully slow, the satisfiability solver is more often to blame than the matcher.

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. Denali's standard file of background facts records fundamental facts about the functions `select` and `store` that represent reads and writes of arrays. One of these fundamental facts 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:

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

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

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

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

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

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

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

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

4 Satisfiability solving

After the matcher has reached a quiescent state, it is sound to assume that the E-graph represents all possible ways of computing the terms that it represents. (More precisely, it is sound to assume that this will be true if the background facts 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. The operations appearing in the E-graph include "machine operations" that can be directly computed by the target architecture; they may also include non-machine operations that are allowed in the input (or in the file of universal facts) 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 (in particular, could mention powers of two).) 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.

We introduce a number of boolean unknowns. Specifically, for each cycle i of the K cycles available, and 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, the computation of T is launched at the beginning of cycle i.

$A(i, T)$: denotes that in the desired machine program, the 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.

In terms of these boolean unknowns we can formulate the conditions under which a K-cycle machine program exists that computes all the goal terms.

There are four basic conditions:

First, writing $\lambda(T)$ for the latency of the 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:

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

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:

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

Fourth, 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 that 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 showstopper. 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 (and how many levels of cache are missed). Having found this information, the programmer can communicate it to Denali using annotations in the Denali source program.

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 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".

5 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 accomodates a multiple instruction architecture where cycle indexes and instruction indexes 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,

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

In this case, the final instruction that computes the `reg6` + `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.

6 Preliminary results

We have implemented a prototype of Denali in Java for the most recent implementation of the Compaq Alpha, a quad-issue chip with multiple register banks and extra delays for moving values between banks, all of which complexity is modeled by our code generator.

One of our standard tests is the problem of reversing the byte order of a four byte value in a register, computing the result into another register. Denali computes the optimal code (which takes five cycles) in about twenty-five seconds, most of which is spent solving a SAT system of some six thousand constraints on twenty-five hundred variables. Other tests also give us confidence that the Denali approach can provide peak performance on ALU-bound register-to-register computations.

We are less far along with memory-bound computations. We have applied Denali to some matrix loops and to a packet checksum routine, and it seems to work nearly as expected, but these experiments have already pointed to changes that our prototype needs. More work and more experience are needed before conclusions can be drawn.

References

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

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

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

[4] 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.

[5] Tracy Larrabee. *Efficient generation of test patterns using boolean satisfiability*. PhD thesis, Stanford University, 1990. Also available as a research report [6].

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

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

[8] 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.

[9] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979. A revised version of this thesis was published as a Xerox PARC Computer Science Laboratory Research Report [10].

[10] G. 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.

[11] Derek C. Oppen and Greg Nelson. Fast decision algorithms based on congruence closure. *JACM*, 27(2), 1980. Also appeared in *SIGACT*.

[12] SAT Research at Princeton Home Page. Chaff satisfiability solver. http://www.ee.princeton.edu/ chaff.