RC 20232 (10/19/95)
Computer Sciance/Mathematics

IBM Research Report

Generating Machine Specific Optimizing Compilers

R. Hoover, K Zadeck

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Generating Machine Specific Optimizing Compilers

Roger Hoover*
Kenneth Zadeck†

1 Introduction

Most portable compilers ignore the target architecture for optimization phases, deferring the introduction of machine specific information to a final code generation pass. Optimization phases in these compilers must make arbitrary assumptions about which techniques will produce faster code. In contrast, most vendor compilers utilize aspects of the machine architecture at all phases of the back end and as a result have sustained a track record of generating better code.

Because of the extensive use of machine knowledge throughout all back end phases of a vendor compiler, these compilers have proved difficult to port. As architecture lifetimes have shortened and the number of architecture variants have increased, developing such optimizing compilers has become prohibitively expensive. In this paper, we show how these costs can be amortized across a number of architectures with a framework for abstracting the machine information from a vendor compiler. We also discuss the tools we are building to tailor the abstract compiler to an architecture specified by a comprehensive machine description.

In the TOAST (Tailored Optimization And Semantic Translation) abstract compiler, all references to target architecture specific information are made indirectly through information derived from a machine description. A TOAST machine description contains the physical resources available, a timing model, and the semantics of each instruction expressed as a small set of primitives. The abstract compiler has no knowledge of the kinds of the instructions available on the architecture or the relative performance of these instructions. In some cases, the information needed by the abstract compiler is easily derivable from the machine description. In other cases, analysis of the semantics of the instructions is required.

The tools to perform this analysis are the novel aspect of the TOAST system and are the primary focus of this paper. We use the following terminology throughout:

target architecture: a computer architecure for which we wish to build a compiler.

semantic primitives: a small set of typed, low-level, machine independent primitives, such as `constant`, `memoryRead`, `add`, `multiply`, `and`, `or`, or `shift`.

*Computer Science Department, IBM TJ Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, rhoover @ watson.ibm.com, (914) 784-7697

†Computer Science Department, IBM TJ Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, zadeck @ watson.ibm.com, (914) 784-6681

1996 ACM Principles of Programming Languages Conference

semantic graph: a graph of semantic primitives.

machine instruction: an instruction on the target architecure. The meaning of each instruction is defined by a semantic graph.

desired operation: a semantic graph that represents the computations needed to perform some action. Usually this is some high-level operation that the compiled program must execute. Desired operations are specified with semantic graphs.

translation: a graph of machine instructions that perform the same actions as a desired operation.

semantic comparator: a theorem proving function that takes as input a desired operation and a set of machine instructions and produces one or more translations. The machine description's timing model can be used to assign a cost to each translation.

Some parts of the compiler can be tailored to an architecture by simply finding a reasonable translation for each desired operation and replacing those desired operations with their translations. Other tailoring of the compiler requires that decision procedures be written to test the relative performance of the translations of different desired operations.

1.1 Strength Reduction as an Example

Consider strength reduction. This optimization replaces multiplications by a loop's induction variable with additions to a loop invariant variable. Most optimizing compilers perform some form of strength reduction. However, the significant differences among implementations lie not in the particular technique used, but to the set of operations to which the technique is applied.

The original motivation for strength reduction was that multiplication was more expensive than addition. Most portable compilers consider this to generally be true and apply the technique wherever they can. A vendor compiler can be more selective, as it may know that multiplication and addition take the same time, or that some multiplications (such as those by a power of two) may be less expensive with addressing modes that perform shifts.

To tailor strength reduction for a particular architecture, TOAST compares the cost of multiply operations with the cost of additions in several contexts that are amenable to strength reduction. One pair of desired operations is used for each context. The first desired operation of a pair is the the unreduced form. The second desired operation is an expression containing the reduced form. The pair of desired operations are translated by the semantic comparator into instruction sequences and the resulting costs of these sequences are compared. If the best sequence for the second desired operation is

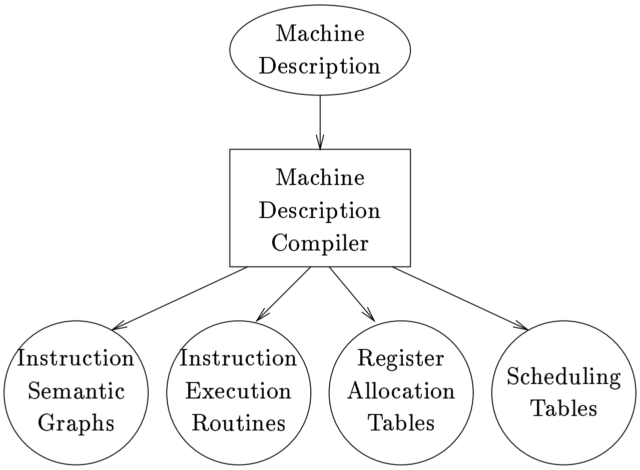

Figure 1: Machine Description Compiler

no faster than the first, strength reduction is inhibited for this class of multiplications[1].

This type of analysis enables TOAST to tailor an implementation to avoid code transformations that are either not productive or potentially regressive. Many optimizations can be tailored in this fashion.

1.2 Outline

In this paper we discuss the TOAST compiler design and show how several novel technologies (including semantic instruction descriptions and semantic comparison) can enable such portable optimization tailoring. We do this in four main sections:

- A high level overview,
- A number of examples of how compilation and optimization tasks are tailored by TOAST,
- Details of our machine description and semantic comparison technology,
- A comparison with related work.

2 High Level Overview

The compiler generated by TOAST looks similar to any vendor compiler, including a front end, machine dependent optimizations, a scheduler, and a register allocator. TOAST differs from conventional optimizing compilers in the way these phases are generated.

TOAST consists of several parts:

1. A machine description that contains a declarative description of an architecture's resources, instructions, and timing,
2. A machine description compiler, illustrated in Figure 1, that converts the machine description into data structures and program fragments.

[1] The tie here should go to inhibiting the reduction since it has a tendency to increase register pressure.

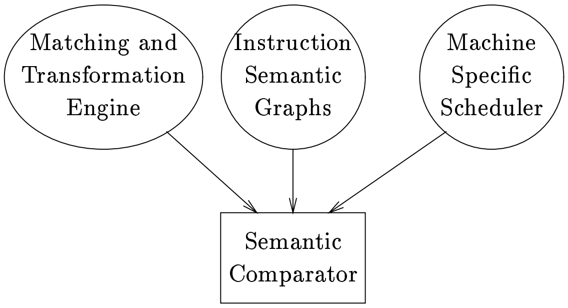

Figure 2: Semantic Comparator

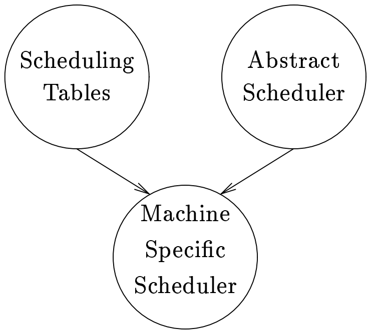

Figure 3: Scheduler

3. A semantic comparator, illustrated in Figure 2, whose knowledge of the target architecture is provided by data structures and program fragments produced by the machine description compiler. This semantic comparator is used to answer complex questions about the target architecture.
4. An abstract compiler whose only direct knowledge of the target architecture comes from the tables and code produced by the machine description compiler and the semantic comparator. For instance, the machine specific parts of register allocation (Figure 3), scheduling (Figure 4), and constant propagation (Figure 5) are computed directly by the machine description compiler, since this information can be derived by examination of individual instructions. However, the machine specific parts of the front end (Figure 6), code improve-

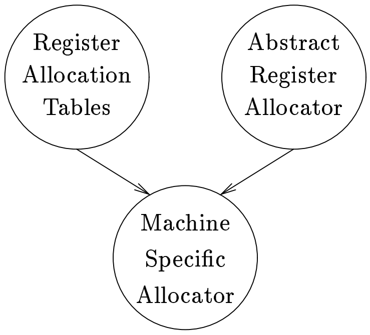

Figure 4: Register Allocator

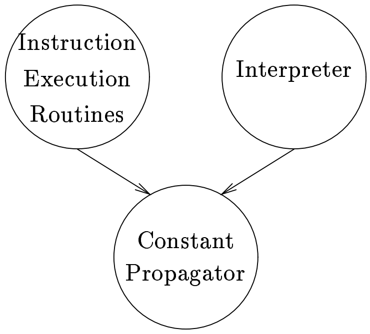

Figure 5: Constant Propagation

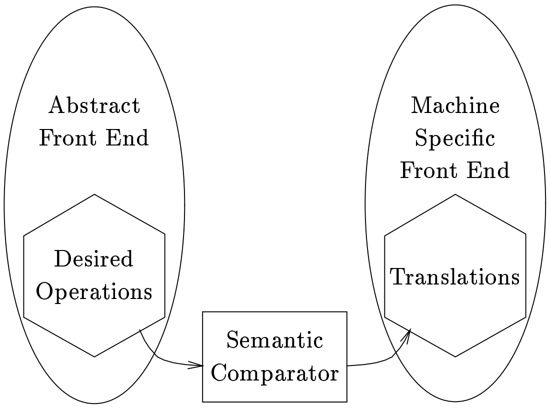

Figure 6: Front End with Code Generator

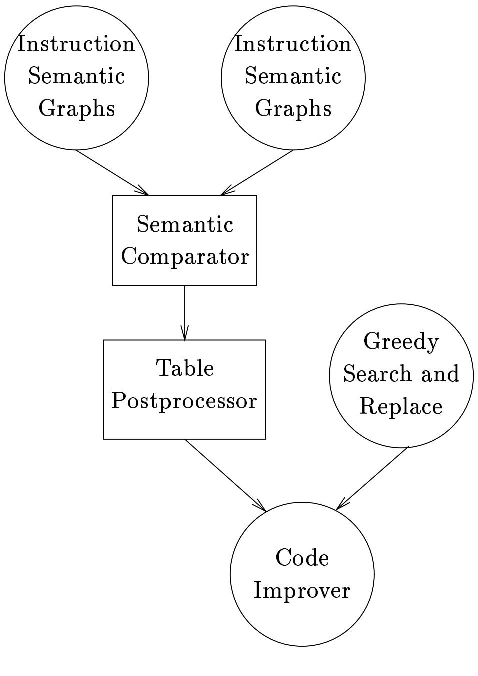

Figure 7: Code Improvement

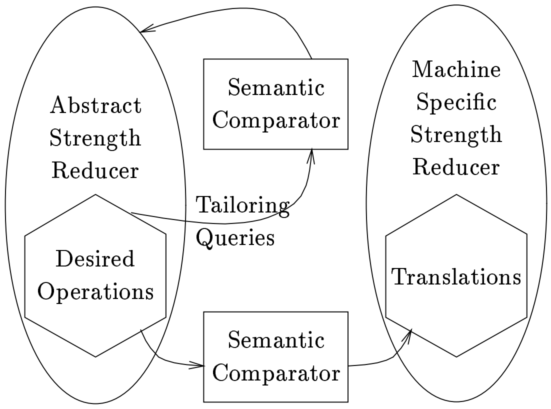

Figure 8: Strength Reduction

ment (Figure 7), and strength reduction (Figure 8) require global knowledge of the semantics and timing of the target architecture. Here, the semantic comparator establishes the relationship between the operations required by the abstract compiler and the instruction set of the target architecture.

The TOAST machine description has a declarative specification of the semantics of each output of each machine instruction. These specifications are based upon a small number of *semantic primitives* (e.g. or, add or memoryRead) and are compiled (via the TOAST machine description compiler in Figure 1) into an internal graph representation that is a variation of PIM [Fie92]. Each desired operation that the compiler needs to implement is likewise specified in terms of these same semantic primitives. Unlike most other machine description based systems, TOAST knows the meaning of these specifications. It has an interpreter that can execute the semantic primitives and it has embedded transformations that relate structurally dissimilar but equivalent specifications.

The fundamental technique used to perform the customization analysis is performed by the *semantic comparator*. TOAST uses a simple theorem prover to implement this function. The semantic comparator returns all combinations of *instructions* that it can prove are semantically equivalent to *desired operation*. A *translation* is one such collection of instructions. Each translation has a *context*, which records the restrictions on instruction usage required for semantic equivalence (e.g. immediate input values and dead outputs). In typical usage, the desired operation is some function that the compiler needs to implement and the instructions are actual machine instructions on the target architecture. Other usages are of interest, however. Comparing a machine instruction (or composition of machine instructions) with other machine instructions yields instruction set equivalences. Comparing a machine instruction on one architecture with machine instructions on a second architecture will generate code for cross architecture simulation or translation.

The timing model of the machine is used to determine the *cost* of each translation, and the list of translations is typically filtered using this cost to determine the best translation.

It is important to note that structural graph matching techniques are not sufficient for the purpose of compiler generation. For example, the unsigned expression $x * 2$ is structurally quite different than $\mathrm{leftShift}(x, 1)$, but semantically they are equal. While it is possible to employ structural matching to find some equivalences between carefully crafted specifications, machine description technology is only useful if the descriptions can be shared over a number of generated compilers (which are likely to be written by different authors). In addition, instructions are typically more general than what is needed in a specific operation. It is therefore necessary to compare graphs that have different forms but have *identical semantics*. TOAST implements semantic comparison by considering all graphs that will match structurally when rewritten with a small collection of semantics preserving transformations.

TOAST is not intended to be a fixed system. We envision many potential uses of TOAST's specification and semantic comparison framework, including:

- A workbench for the development of portable optimization phases,
- Instruction set design and processor simulation using code from a TOAST compiler that was generated specifically for a proposed instruction set,
- A tool for transforming machine specific optimizations into ones that can be automatically ported.

3 Tailoring Optimizations

In most portable compilers, code generation is necessarily a single monolithic phase that occurs after optimization. Vendor compilers decompose the task of code generation into subproblems that are best solved in conjunction with other optimizations. For instance, the problem of choosing the correct addressing modes may be spread over several optimizations. Strength reduction is coded to avoid reducing multiplications that can be done by an addressing mode. Reassociation is coded to arrange additions, subtractions, and multiplications to look like the addressing modes of the machine's instructions.

Code generation in a TOAST compiler is done in the vendor compiler style. Instruction selection is performed directly by a tailored front end and is improved by the subsequent optimization phases. While this gives the compiler additional flexibility, it also introduces complexity that other portable compilers avoid. The IL consists of a graph containing target architecture instructions rather than a stream of portable abstract operations. Every optimization must be tailored to perform its optimizations over this machine specific language. Because TOAST has a description of each instruction, it is possible to have it generate the necessary glue that allows each phase to understand all of the machine dependant forms in the IL. We consider instruction selection and several other important compiler phases in the following subsections.

3.1 Instruction Selection

The TOAST machine dependent IL is emitted by a front-end that is tailored as follows. First, the front-end is written as a typical procedural program that parses the target language and performs any language specific high level optimizations. All operations that the front-end emits, however, are coded as special purpose *desired operations* using the same specification language used in the machine description. Second, the specifications for these desired operations are compiled into semantic graphs by the TOAST machine description compiler. Finally, TOAST replaces the desired operation in the front end with the lowest cost translation computed by a semantic comparison between the desired operation and the machine's instruction set. The process is illustrated in Figure 6 and the result is a front-end that emits an IL that is quite close to the instruction set of the given architecture (but may have the actual machine instruction abstracted due to issues like offset size that cannot be resolved until actual object code generation). The front end produces code that is locally good but is globally mediocre. It is up to the subsequent phases to globally improve this code.

3.2 Code Improvement

It is the responsibility of the code improver to replace subgraphs of instructions with more efficient subgraphs with equivalent semantics. The code improver, illustrated in Figure 7, is tailored for a particular instruction set by comparing that instruction set with itself to yield a table of many-to-one instruction transformations that result in faster execution. This table of subgraph to subgraph transformations are greedily applied at compile-time by a simple automaton that traverses the IL.

We build the transformation table as follows. For each instruction i in the instruction set I we compute the set:

$$\{m \in \mathrm{semanticCompare}(i, I) \mid \mathrm{cost}(m) > \mathrm{cost}(i)\}$$

For each of the translations in this set, we record the context in which the translation was successful (dead outputs of instructions, instructions with specific immediate inputs, etc.). The instructions and their translation and their context become the left hand side of a transformation. Wherever m is successfully located in the code by the automaton, the entire subgraph is replaced by i.

3.3 Predicate Rewriting

As processor pipelines have become longer, the cost of conditional branching has increased. For many desired operations, it is possible to rewrite the code to reduce the number of conditional branches. Unfortunately, the relative performance of branches and their interaction with other instructions varies greatly among different architectures and implementations. Therefore, a portable compiler cannot assume that it is always desirable to choose a particular branching construct or to rewrite an expression using non-branching operations.

For commonly occurring idioms, such as those representable as conditional assignments, the relative desirability of various code transformations can be tested via semantic comparison. The predicate rewriter is tailored to use the best sequences for each idiom as computed by the semantic comparator.

3.4 Reassociation

Reassociation rewrites expressions into a canonical form (typically sums of products), and it is useful if this canonical form matches the capabilities of the target architecture. For instance, the size, type, and cost of immediate fields depends upon the machine architecture. A machine dependent reassociation phase manipulates constants and offsets to fit into the available fields. For example, the IBM 370 restricts some immediate fields to $[0, 4095]$. If an array reference of the form $X(i-1)$ occurs within a loop where i is the index variable, reassociation can rewrite the expressions to have the base register point to $X(-1)$, so that all offsets are positive.

The size, type, and relative costs of each immediate field is explicitly represented in the instruction semantics and can be used to tailor reassociation.

3.5 Other Optimizations

There are many additional compiler tailoring tasks that are not expressible as semantic comparison problems but are greatly simplified given the comprehensive TOAST machine description. For example, the semantic instruction descriptions allow the generation

```
DO
  INSTRUCTION 'Integer.Xor.VV.Rec;'
  FORMAT X(Reg0.RS(RS=Op1), Reg1.WRA(WRA=Target),
           Reg2.RB(RB=Op2), Opcode=31, EOpcode=316, Rec=1);
  NAME 'xor.';

  VAR Target  : WRITE     Word GPR[0..31];
  VAR Op1     : READ      Word GPR[0..31];
  VAR Op2     : READ      Word GPR[0..31];
  VAR PC      : MOD       AD_4 PC;

  VAR CRLT    : WRITE     Boolean CR[LT]          DELAY 4;
  VAR CRGT    : WRITE     Boolean CR[GT]          DELAY 4;
  VAR CREQ    : WRITE     Boolean CR[EQ]          DELAY 4;
  VAR CRSO    : WRITE     Boolean CR[SO]          DELAY 4;
  VAR XERSO   : READ      Boolean XER[SO];

  ASSIGN CRLT           <- IntegerSignedLT(IntegerXor(Op1, Op2), 0);
  ASSIGN CRGT           <- IntegerSignedGT(IntegerXor(Op1, Op2), 0);
  ASSIGN CREQ           <- IntegerEQ(IntegerXor(Op1, Op2), 0);
  ASSIGN CRSO           <- XERSO;
  ASSIGN Target         <- IntegerXor(Op1, Op2);
  ASSIGN PC             <- AddrNext(PC, 4a);

  IN RiosExt PAGE 54; IN PcExt32 PAGE 66;
  TIME IntegerLogicalClass 1;
END;
```

Figure 9: **xor record** instruction.

of simulation routines that propagate constants through individual instructions. Scheduling is likewise aided by the architectural cost model. Register allocation makes use of instruction resource constraints. These and other bookkeeping tasks keep the need for special case procedural code to a minimum.

4 TOAST Technology

In this section we describe the technical details of:

- TOAST semantic specifications for both machine instructions and the desired operations, and
- The semantic comparator that allows TOAST to relate these specifications.

4.1 Semantic Representation

TOAST requires a substantial machine description to describe an architecture as each instruction's full semantics and timing must be specified. The description is declarative rather than procedural[2] and is processed by a machine description compiler that emits data structure definitions and code to be compiled into the semantic comparator.

We illustrate the important aspects of the machine description using the RS6000's **xor record** instruction, shown in Figure 9. The `INSTRUCTION` clause contains the internal TOAST name of the instruction. The `FORMAT` clause is used to guide both decoding and encoding of the object code representation. The `NAME` clause gives the assembly language instruction name.

[2] We do have the ability to code special cases procedurally when all else fails.

The `VAR` clauses describe registers that the instruction either reads or writes. The fields in this clause define a local name for the register, how it will be used (`READ`, `WRITE`, `MOD` or `IMMEDIATE`), a type, a register pattern, and possibly a delay. The local name and type are used in the definition of the semantics. The register pattern indicates onto which machine registers this variable can be mapped. The local variables `Target`, `Op1`, and `Op2` of the example **xor record** instruction can be mapped into any of the 32 general purpose registers (`GPR`), while `CRLT` can be mapped only to the `LT` portion of the condition register (`CR`). The `DELAY` clause on the `LT` variable indicates that this value is not available until four cycles after the instruction's completion.

```
FUNCTION IntegerXor(Op1:T, Op2:T):T
  TYPE T = INTEGER(*);
  EXPRESSION
    IntegerAnd(
      IntegerOr(Op1, Op2),
      IntegerComplement(
        IntegerAnd(Op1, Op2)));
END;
```

Figure 10: `xor` macro.

The `ASSIGN` clause specifies the semantics of each writable variable using a side-effect free function of semantic primitives over readable variables. To keep these specification manageable we have a facility for polymorphic, strongly typed macros that allow us to build more complex operations from the simple ones. As shown in Figure 10, `IntegerXor` is implemented in terms of the semantic primitives `IntegerAnd`, `IntegerComplement`, and IntegerOr for INTEGER types of all signs and lengths.

IN may prefix any clause or block in the machine description to allow the specification of differences in architecture variants. In this example it modifies the PAGE clause, which documents the instruction's different location in the RS6000 and PowerPC architecture reference manuals. While IN is typically used for TIME or PAGE clauses, it is also useful in front of entire instructions, PROCESSORUNIT and TYPE clauses.

```
IN RiosIExt
  PROCESSORUNIT IntegerUnit 1
    CONTAINS
      IntegerLogicalClass,
      IntegerAddClass,
      IntegerShiftClass,
      IntegerMultiplyClass,
      IntegerDivideClass;
IN Power604
  PROCESSORUNIT SimpleIntegerUnit 2
    CONTAINS
      IntegerLogicalClass,
      IntegerAddClass,
      IntegerShiftClass;
IN Power604
   PROCESSORUNIT ComplexIntegerUnit 1
     CONTAINS
       IntegerMultiplyClass,
       IntegerDivideClass;
```

Figure 11: Part of the description of the functional units.

The TIME clause indicates the class of processor unit that an instruction can be assigned to and the number of cycles the instruction consumes on that unit. Since architecture implementations tend to differ in the number and type of functional units, part of the implementation specific description is a block that describes the number of processor units and the classes of instructions that can be assigned to each unit. In the example is shown in Figure 11, the RS6000's IntegerLogicalClass instructions are assigned to a single IntegerUnit, whereas the 604 implementation of the PowerPC can assigned them to either of two SimpleIntegerUnits. The PROCESSORUNIT statements, along with the TIME and DELAY clauses allow us to build a fairly accurate timing estimate of the cost of a sequence of instructions. However, since it is computed in isolation it cannot include all pipeline or any cache effects.

4.2 Machine Description Factoring

A common feature of most machine description languages is a *factoring* mechanism that allows the writer to isolate commonly occurring architectural features. For example, the Intel X86 machines have many addressing modes that apply to many instructions. Enumerating every addressing mode with every instruction would result in an enormous specification. Factoring allows the addressing modes to be specified separately and then correlated with the instructions to which they apply.

Factoring may serve two purposes. First, it drastically reduces the size of the specification. Second, it makes explicit commonly occurring patterns that can be used to build the code generator. While matching on the factorization simplifies the task, the quality of the code generator produced by these tools is dependant on choosing the correct set of features to factor the instruction set. Because of this, TOAST uses factoring only for the first purpose.

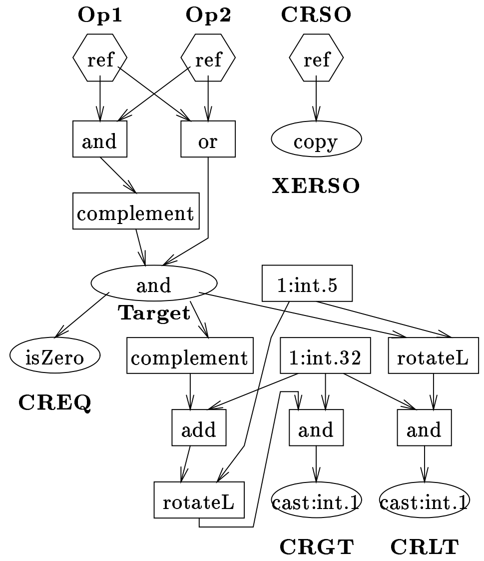

Figure 12: Semantic Graph of **xor record**

Unlike other compiler generators, our machine description compiler expands out all of the factoring before any other processing is performed.

While the expansion of factoring and subsequent processing of the machine description gives TOAST an internal representation that is not tightly tied to the specification, the expanded representation can grow to an unmanageable size if a large selection of addressing modes are available. Fortunately, it is possible to automatically factor the internal representation by identifying identical *super-expressions* that share multiple sub-expressions. Not only can this re-factored representation be used to save space, but since the semantic comparator (described below) works from outputs to inputs, coalescing allows its work to be shared among the identical super-expressions.

For example, the CRLT, CRGT, and CREQ outputs shown in Figure 9 are generated from three macros common to all of the many *record* variants in the RS6000 instruction set. Beginning with all semantic graph outputs of all machine instructions, we identify syntactically identical semantic primitives and represent these identical primitives by a node in a coalesced semantic graph. Edges of this coalesced graph are actually a vector of edges, each element branching to nodes representing the different semantic primitives in the instruction semantic graphs. Thus, the semantic primitives generated by one instance of the IntegerSignedLT macro (as used in the **xor record** instruction of Figure 9) are shared with the corresponding nodes of all other record instructions. The semantic graphs diverge at the nodes that correspond to the macro's arguments.

Super-expressions also occur when a given instruction has multiple addressing modes. The semantic graphs for each instruction variation is identical with the exception of the semantic primitives used to obtain the instruction's input values. The coalesced graph branches to represent these different inputs.

4.3 Machine Description Processing

When invoked upon a particular architecture A, the machine description compiler:

- Extracts only the sections of the description that pertain to the implementation of A,
- Performs syntax checking and semantic checking (including type checking all semantic specifications) of the description,
- Expands all of the factoring, and
- Emits a data structure representation[3] of the machine description.
- Produces data structures and code used by phases of the abstract compiler that do not require processing by the semantic comparator. These include tables used by the register allocation and scheduling phases and code to interpret instructions in the constant propagation phase.

These data structures are compiled with the semantic comparator and upon initialization the following optimizations are performed over each semantic instruction description.

- common sub-expression elimination
- unreachable code removal
- constant propagation

These optimizations are necessary because outputs are specified independently and abstractly. Figure 12 shows the resulting representation for the RS6000's **xor record** instruction.

4.4 Semantic Comparison

Semantic comparison is the process of identifying equivalences between semantic representations. In TOAST, we compare the semantics of the desired operation with the semantic descriptions of the architecture's instruction set. As equal representations are likely to be syntactically different, the semantic comparator must understand the meaning of the representations and construct a proof of their equality. This theorem proving must be limited, however, as the number of semantic comparisons needed for compiler generation is large. In this section, we discuss our implementation of semantic comparison.

The semantic comparator is implemented in three phases, which we discuss separately below. The first phase discovers *toe prints*, single output computations of instruction graphs that are semantically equivalent to subgraphs of the desired operation. The second phase forms *foot prints*, which are valid combinations of toe prints that can be provided by machine instructions. The final phase combines foot prints into translations that have semantics equal to the desired operation graph. While all three phases use potentially exponential exhaustive searches, our prototype implementation has demonstrated that their computation is quite feasible in practice for real instruction sets.[4] Much of the computation required by these three phases is concentrated in finding toe prints, a process that is easily parallelized.

4.4.1 Toe Prints

Each *toe print* represents an equivalence between a subgraph of the desired operation and a subgraph of a single instruction. While the subgraph of the desired operation is unrestricted[5], we consider only instruction subgraphs that form a *slice* [Wei84] from a single instruction output. That is, we consider exactly those nodes of the instruction that contribute to the given output, which corresponds to the smallest portion of a machine instruction i that can be utilized in a program. Figure 13 illustrates the slice from the *Target* output of the **xor record** instruction of Figure 12.

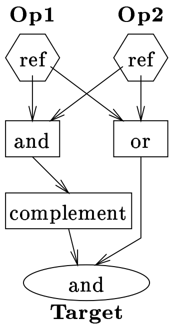

Figure 13: Slice of **xor record**

We search for toe prints by comparing each output of i with every node of the desired operation. At each successful comparison, the search is continued recursively to the respective predecessor nodes in both desired operation and instruction graphs. The search succeeds when the inputs of i are reached and a record of the matched nodes is stored on a list of toe prints for instruction i.

To complete the description of the toe print search, we must specify how the nodes of the two graphs are compared. We need to determine whether or not the nodes of the instruction *implement* the same semantics as the nodes of the desired operation. Clearly, a node implements an identical node as does an instruction node that provides strictly more defined behavior than the desired operation node (such as two add operations, the instruction add with larger integer type[6]). Dissimilar nodes, however, might implement the same semantics. We use two techniques *satisfaction* and *transformation* to identify such identities wherever possible.

Satisfaction is used when the desired operation node is a constant and the instruction node represents some non-constant expression. The expression implements the constant if we can determine some set of input values for the instruction such that its expression equals the desired operation constant. The satisfaction heuristic takes a list of *magic values*, special values based upon each input type, and exhaustively tries all input combinations with a semantic graph interpreter, terminating if an expression value equal to the constant is found. Examples of magic numbers include $\{0, 1\}$ for boolean, $\{-1, 0, 1, 2, 4, 8, \ldots\}$ for integer, etc. If the input is via a register, a load immediate instruction is added to the translation. Input values of a successful satisfaction become part of the translation context.

Transformation exhaustively applies a carefully selected set of semantics preserving transformations to the desired operation graph wherever they might help the instruction's semantic primitives to implement the desired operation, either by syntactic comparison or by satisfaction. The TOAST semantic comparator contains a flexible transformer, which allows easy experimentation with the transformation set. The results shown later in the paper (Figure 18), use the identity transformations:

$$\begin{array}{cc} x \Rightarrow x + 0 & x \Rightarrow 0 + x \\ x \Rightarrow x * 1 & x \Rightarrow 1 * x \\ x \Rightarrow x \vee 0 & x \Rightarrow 0 \vee x \\ x \Rightarrow x \wedge -1 & x \Rightarrow -1 \wedge x \end{array}$$

[3] We currently use ML as the implementation language.

[4] We can generate the code improver tables for the RS6000 architecture in less than 36 cpu hours on an RS6000.

[5] We introduce a restriction on the desired operation in our later discussion of flawed foot prints.

[6] Note that overflow condition bits have their own semantic specifications.

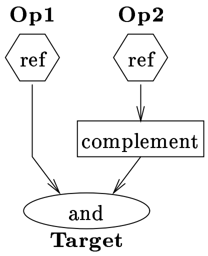

Figure 14: Semantic Graph of **and complement**

the annihilator transformations:[7]

$$\begin{array}{rr} 0 \Rightarrow \mathtt{star} * 0 & 0 \Rightarrow \mathtt{star} * 0 \\ -1 \Rightarrow \mathtt{star} \vee -1 & -1 \Rightarrow -1 \vee \mathtt{star} \\ 0 \Rightarrow \mathtt{star} \wedge 0 & 0 \Rightarrow 0 \wedge \mathtt{star} \end{array}$$

and the data conversion transformation: $x : t \Rightarrow \mathtt{cast}(x, t)$.

Consider an instruction operation $a + b$ that is to match the corresponding desired operation t. Using the first two identity transformations, we try matching t, $t + 0$ and $0 + t$. In order for the translation to succeed, t must be an addition of equal or smaller type, or t is matched by a (alternatively b) and b (alternatively a) matches the constant 0. Of course, these subsequent matches may involve further transformation or satisfaction.

Transformations are performed only to the desired operation and are selected only when the right hand side is guaranteed to match the next instruction node in the output-to-input matching traversal. Since a transformed semantic graph node must be matched to an instruction semantic graph node before undergoing subsequent transformation, all transformation sequences are bounded.

As an example, consider the RS6000 **and complement** instruction of Figure 14 as the desired operation in a semantic comparison with the RS6000 instruction set (which includes the **xor record** instruction of Figure 12)[8]. Since the toe print search tries to compare all nodes of the desired operation with all instruction outputs, at some point the complement node of Figure 14 will be compared to the bottom **and** node of the slice of Figure 13. Since these nodes are dissimilar, a series of transformations shown in Figure 15 are applied to the subgraphs of the desired operation, each bringing the desired operation structurally closer to the xor slice.

The first (identity) transformation introduces an and to match the and at the **Target** output of **xor record**. The second (annihilator) transformation replaces the -1 of the previous transformation with an or of star (which matches anything, in this case **Op1**) and -1 (which is matched with **Op2** by trivial satisfaction. The third (identity) transformation introduces another and, whose inputs match **Op1** (equal references) and **Op2** (trivial satisfaction). A subsequent pass verifies that duplicate matches in the DAG are compatible, in this case **Op1** has been matched with star and ref; **Op2** has been matched twice with -1. The resulting toe print is illustrated in Figure 16. Note that we have only described one sequence of successful transformations here. Many other transformations are tried and many are rejected, resulting in 34 toe prints in the semantic comparison over the entire instruction set.

4.4.2 Foot Prints

Toe prints correspond to the use of a single output of an instruction. However, instructions often generate multiple outputs (condition

[7] star is a special semantic primitive that can be implemented by any other semantic primitive.

[8] This computation is part of building the RS6000 code improver.

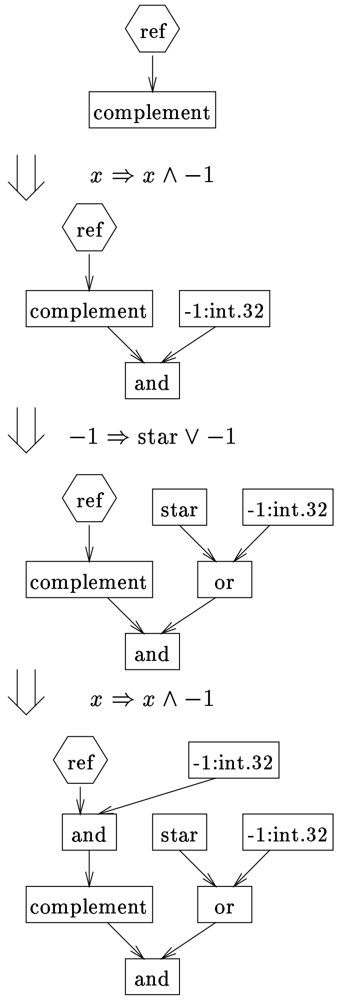

Figure 15: Transformation and Satisfaction

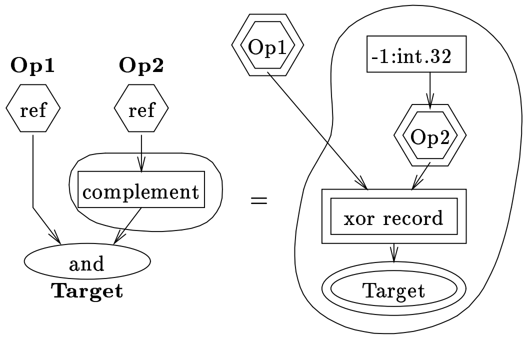

Figure 16: Toe Print of **xor record**

codes, etc). A *foot print* represents the use of an entire instruction, which might only involve a single output but in general could make use of multiple outputs. Thus, we consider as potential foot prints all combinations of toe prints for a given instruction.

A potential foot print is elevated to become a foot print if it satisfies two properties. First, it must be *consistent* and second, it must be *flawless*. A foot print is *inconsistent* if any instruction input has been assigned incompatible values by the satisfaction process. We consider a foot print to be *flawed* if it only partially fulfills the computation needs of the desired operation. This occurs when the all of the following hold.

1. The desired operation requires a sub-expression s of the instruction
2. s is not available as an output of the instruction
3. s is used in the desired operation in computations not matched by the foot print

Since the matching phase does not consider combinations that duplicate nontrivial expressions, flawed foot prints will inhibit a successful translation. The example of Figure 16 is both consistent and flawless and is therefore a valid foot print.

Foot prints that pass these tests are added to a foot print set global to this semantic comparison. Note, however, that as many instructions for a given architecture may provide similar capabilities, many foot prints will match identical subgraphs of the desired operation. As each of these instructions are interchangeable for the purpose at hand, we filter the global foot print set into a set of *foot print categories*. Each category contains a set of foot prints that match the same subgraph of the desired operation and have identical translation contexts. The semantic comparison example (above) finds all three possible foot print categories.

4.4.3 Translations

The translation phase searches the entire space of all combinations of foot print categories with two limitations. First, each foot print must match at least one semantic primitive in the desired operation. Second, only trivial amounts of the desired operation (i.e. constants) may be provided by more than one foot print. Both restrictions dramatically reduce the search space. These restrictions avoid the inclusion of a potentially unbounded number of no-op instructions, completely dead instructions, or instructions that duplicate the computation of other instructions.

The translation algorithm exhaustively tiles the desired operation graph with all combinations of foot prints. There are two important optimizations to note:

1. Initially, any foot print may be placed. Once placed, however, this foot print will preempt the placement of any foot prints that cover any of the same nodes of the desired operation. After each placement, the foot print list is filtered to eliminate all foot prints that contain covered nodes.
2. Since all nodes of the desired operation must be covered for a translation, a check is made to see if all nodes could possibly be covered by foot prints remaining after step (1). If it could not possibly cover, the search is abandoned.

All successful tilings of the desired operation graph are returned by the TOAST semantic comparison function. While each translation represents many similar instruction matches, TOAST retains the factored list expanding by foot print categories only where individual instruction matches are needed. Figure 17 shows the **and complement** instruction matched by foot prints of **xor record** and **and**. The semantic comparison example tiles **and complement** with foot print categories two ways. These two foot print category matches represent 114 different instruction combinations with semantics equal to the **and complement** instruction.

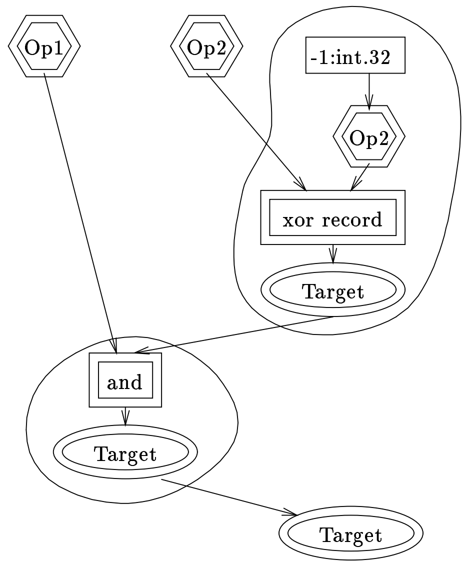

Figure 17: Translation for **and complement**

5 Related Work

The major difference between TOAST and other compiler-compilers is the depth of analysis that the system performs and the amount of customization that is done to the generated compiler. Both improvements are a direct result of the comprehensive nature of our machine description.

5.1 Tree-based techniques

Much of the recent work on portable code generation technology [SU70, AJ76, GG78, AG89, Pro92] has used tree based instruction specifications to generate code from a tree based intermediate language. The primary result of this work is that locally optimal code can be efficiently generated in this tree context.

TOAST is based on the assumption that reasonable code generation over DAGS is better than optimal code generation over trees. With a few exceptions (most notably MIPS), architectures contain many instructions with more than one output. While it may be acceptable to generate two instructions to obtain both outputs of a *divide-remainder* instruction, it is not acceptable to duplicate instructions that produce condition codes. Furthermore, intermediate code containing common sub-expressions is not reasonably handled by tree-based techniques.

5.2 Work of Kessler

Our work originally started out in the style of Kessler's dissertation [Kes84][9]. He had a machine description that described the semantics of each instruction in terms of a small set of low-level

[9] Neither Kessler's dissertation nor his subsequent paper [Kes86] describe his matching algorithm in detail. Our knowledge is based on personal conversations with him.

primitives. This description was then used as input to a matching engine that attempted to discover special cases of one instruction (or a combination of instructions) that performed the same operations as another instruction.

The problem with this approach is that the set of matches is highly dependent on the *style* used to describe the instructions. Kessler's matcher is almost entirely structural, though he did commutative and associative transformations. While many matches are found with such a simple approach, many are also missed. Our technique is more powerful than Kessler's because of our transformation techniques.

We believe that our semantic comparison techniques find more translations than Kessler. A direct comparison would be difficult as Kessler's test cases were older CISC architectures (VAX, MC68000) and his system is no longer operational to perform comparisons using modern machines.

5.3 Peephole Optimizer (PO)

The PO system [DF84b, DF84a] contains technology similar to, though weaker, than our semantic comparator. Unlike our system, (and Kessler's system), PO determines the matching sequences by looking at combinations that occur in intermediate code derived by running the front end of the compiler on a training suite. Pairs and triples of instructions that occur inside the range of the peephole are looked up in the machine description. The matching technique is structural but, because of the factoring done by the machine description writer, it often succeeds. If a match is found, it is written to a log file. The generated compiler is derived by manipulating the this log. The quality of matches found is dependent on the quality of the test suite. While a comprehensive test suite will cover commonly occurring cases, it is unreasonable to assume that a test suite can be developed to cover all possible cases.

Comparisons made between Kessler's technique and PO seem to indicate that PO rarely finds anything that the Kessler's technique would not find; though the converse is certainly not true. Kessler's technique finds substantially more matches than PO (although it could be argued that many of these matches are never utilized).

While PO is capable of producing a code generator similar to ours, it does not have the ability to use this matching capability to customize other phases of the compiler where high level actions are based on comparing the cost of matches as we do with strength reduction.

5.4 GNU Superoptimizer

The Gnu SuperOptimizer (GSO) [GK92] uses a matching strategy by Massalin [Mas87] that is significantly more powerful than our tiling and transformations. It is also significantly more expensive and cannot be extended to consider the entire processor problem state instruction set.

TOAST is based on manipulating the semantics description of instructions until a sequence can be tiled over a single instruction. In GSO, the semantics are entirely operational; an interpreter is built for each instruction that is to be simulated, if possible using the C *asm* construct. A C function is also written that computes the desired operation that GSO is searching for. Every possible combination of an instruction subset (up to a small sequence length) is tried using a small number of specially chosen inputs and the outputs are compared to the values computed by the desired operation. A match is declared when some sequence finds the right answer on all of the specially chosen inputs.

GSO is capable of discovering truly obscure sequences that perform some desired operations, but is limited by its high search costs. On today's hardware, the Massalin technique is limited to testing short sequences (< 6 long) chosen from a small instruction subset (< 10). It is unlikely that this search technique could be expanded to consider the entire instruction set of a processor or to find matches that contain long sequences of instructions.

The TOAST satisfaction technique is similar to the GSO technique. Our matching discoveries are limited by the immediate input values and transformations considered. TOAST cannot discover obscure mathematical identities not provable by these manipulations. TOAST's primary advantage is its ability to match with the entire instruction set. In addition, TOAST's matching performance vs. running time can be scaled by adding or removing transformations and immediate inputs in the matching engine.

5.5 GNU CC

The most successful portable compiler to date is the Gnu C compiler. Although it has a different implementation, the GCC code generator is largely modeled on PO [GK92]. It has been ported to a wide array of processor architectures and in many cases generates code that exceeds the performance of code compiled by the processor manufacturers' compilers.

Our abstract compiler is, in principal, very similar to the Gnu C. Both compilers use an IL derived directly from the instruction set and embed knowledge of the machine throughout the back end. However, the way that the two compilers are *taught* about the machine is entirely different.

Each GCC port includes a set of hand specified configuration data that gives machine independent compiler hints about the target architecture. While the specification may contain enough information to figure out which instruction sequences can be replaced by simpler sequences, the specification contains no timing information. The only cost model available is that shorter is better. No attempt is made to derive any nontrivial information from the specification. For example, the person responsible for the port must identify which forms of multiplication should be strength reduced. TOAST technology automates this process.

6 Summary and Current Status

There are many parts of a compiler that can benefit from customization based upon the target architecture. While much of the current portable compiler work limits this customization to code generation, allocation, and perhaps scheduling, the TOAST compiler architecture demonstrates that virtually all phases of the compiler can benefit from knowledge of the machine architecture.

The TOAST machine description compiler is written as an attribute grammar using Linguist. We have a complete machine description for the IBM Power family and a partial machine description for the Intel X86 family.

Our machine descriptions are divided into two parts. The first is a machine independent part that is common to all architectures. The version that we currently use contains 37 semantic primitives and 89 higher level macros. This specification is about 1000 lines long. The second part describes an entire family of architectures. The Power architecture contains the full description of 11 variations of both the PowerPC and RS6000 lineage, and is about 8000 lines.

We have built a prototype matching engine (in SML), which we have used to construct the code improvement table described in Section 3.2. Figure 18 illustrates the number of match categories found ($\circ$) and the number of implied instruction matches ($+$). As of the writing of this paper, it is unclear what portion of these matches are useful and usable for code improvement. There are several factors here that require further study. First, is unclear how

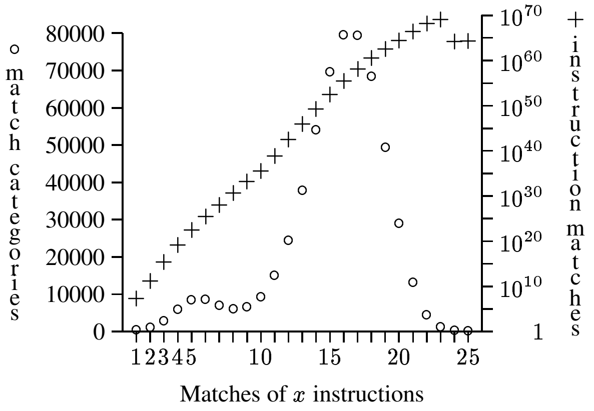

Figure 18: RS6000 Code Improver

often matches of a dozen or two instructions in length will occur in code generated by real programs. Also, some larger matches that do occur are composed of smaller matches, which our greedy replacement algorithm may be able to process. Thus, we may wish to limit the code improver to smaller matches. Second, it will be necessary to preprocess the translations emitted by the code generator at compiler generation time to limit the number of match categories that must be searched to find a match at compile time.

We have designed TOAST with facilities for experimentation. For example, our semantic comparator easily allows its transformation set and other parameters to be manipulated at compiler generation. We are currently bringing the scheduler and register allocator on line, which will complete our test bed for determining the effects of changes in the underlying matching technology upon the quality of the generated code.

References

[AG89] A. V. Aho and S. W. K. Ganapathi, M. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23(3):488–501, July 1976.

[DF84a] J. W. Davidson and C. W. Fraser. Automatic generation of peephole optimizations. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 111–116, June 1984. Published as *SIGPLAN Notices* Vol. 19, No. 6.

[DF84b] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Trans. on Programming Languages and Systems*, 6(4):505–526, October 1984.

[Fie92] John Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, June 1992. Published as Yale University Technical Report YALEU/DCS/RR–909.

[GG78] R. S. Glanville and S. L. Graham. A new method for compiler code generation. *Conf. Rec. Fifth ACM Symp. on Principles of Programming Languages*, pages 231–240, January 1978.

[GK92] T Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *Proc. SIGPLAN'92 Symp. on Compiler Construction*, 7(7):341–352, June 1992. Published as *SIGPLAN Notices* Vol. 27, No. 6.

[Kes84] P. B. Kessler. *Automated Discovery of Machine-Specific Code Improvements*. PhD thesis, Computer Sci. Dept., U. of California at Berkeley, Berkeley, CA, December 1984.

[Kes86] P. B. Kessler. Discovering machine-specific code improvements. *Proc. SIGPLAN'86 Symp. on Compiler Construction*, pages 249–254, July 1986. Published as *SIGPLAN Notices* Vol. 21, No. 7.

[Mas87] H. Massalin. Superoptimizer: A look at the smallest program. *Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 22(10):122–126, October 1987.

[Pro92] T. A. Proebsting. Simple and efficient burs table generation. *Proc. SIGPLAN'92 Symp. on Compiler Construction*, 27(7):331–340, July 1992.

[SU70] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, October 1970.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, July 1984.