

Near-Optimal Instruction Selection on DAGs

David Ryan Koes
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dkoes@cs.cmu.edu

Seth Copen Goldstein
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
seth@cs.cmu.edu

ABSTRACT

Instruction selection is a key component of code generation. High quality instruction selection is of particular importance in the embedded space where complex instruction sets are common and code size is a prime concern. Although instruction selection on tree expressions is a well understood and easily solved problem, instruction selection on directed acyclic graphs is NP-complete. In this paper we present NOLTIS, a near-optimal, linear time instruction selection algorithm for DAG expressions. NOLTIS is easy to implement, fast, and effective with a demonstrated average code size improvement of 5.1% compared to the traditional tree decomposition and tiling approach.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Optimization

General Terms

Algorithm, Performance

Keywords

Instruction Selection

1. INTRODUCTION

The *instruction selection problem* is to find an efficient mapping from the compiler's target-independent intermediate representation (IR) of a program to a target-specific assembly listing. Instruction selection is particularly important when targeting architectures with complex instruction sets, such as the Intel x86 architecture. In these architectures there are typically several possible implementations of the same IR operation, each with different properties (e.g., on x86 an addition of one can be implemented by an `inc`, `add`, or `lea` instruction). CISC architectures are popular in the embedded space as a rich, variable-length instruction set can make more efficient use of limited memory resources.

Code size, which is often ignored in the workstation space, is an important optimization goal when targeting embedded proces-

sors. Embedded designs often have a small, fixed amount of on-chip memory to store and execute code with. A small difference in code size could necessitate a costly redesign. Instruction selection is an important part of code size optimization since the instruction selector is responsible for effectively exploiting the complexity of the target instruction set. Ideally, the instruction selector would be able to find the optimal mapping from IR code to assembly code.

In the most general case, instruction selection is undecidable since an optimal instruction selector could solve the halting problem (halting side-effect free code would be replaced by a `nop` and non-halting code by an empty infinite loop). Because of this, instruction selection is usually defined as finding an optimal *tiling* of the intermediate code with a set of predefined machine instruction tiles. Each tile is a mapping from IR code to assembly code and has an associated cost. An optimal instruction tiling minimizes the total cost of the tiling. If the IR is a sequence of expression trees, then efficient optimal tiling algorithms exist [3]. However, if a more expressive directed acyclic graph (DAG) representation [1] is used the problem becomes NP-complete [8, 4, 33].

In this paper we describe NOLTIS, a near-optimal, linear time instruction selection algorithm for expression DAGs. NOLTIS extends existing instruction selection techniques. Empirically it is nearly optimal (an optimal result is found more than 99% of the time and the non-optimal solutions are very close to optimal). We show that NOLTIS significantly decreases code size compared to existing heuristics. The primary contribution of this paper is our *near-optimal, linear time DAG tiling algorithm*, NOLTIS. In addition, we

- prove that the DAG tiling problem is NP-complete without relying on restrictions such as two-address instructions, register constraints, or tile label matching,
- describe an optimal 0-1 integer programming formulation of the DAG tiling problem,
- and provide an extensive evaluation of our algorithm, as well as an evaluation of other DAG tiling heuristics, including heuristics which first decompose the DAG into trees and then optimally tile the trees.

The remainder of this paper is organized as follows. Section 2 provides additional background and related work. Section 3 formally defines the problem we solve as well as proves its hardness. Section 4 describes the NOLTIS algorithm. Section 5 describes a 0-1 integer program formulation of the problem we use to evaluate the optimality of the NOLTIS algorithm. Section 6 describes our implementation of the algorithm. Section 7 provides detailed empirical comparisons of the NOLTIS algorithm with other techniques. Section 8 discusses some limitations of our approach and opportunities for future work, and Section 9 provides a summary.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.
Copyright 2008 ACM 978-1-59593-978-4/08/04 ...\$5.00.

2. BACKGROUND

The classical approach to instruction selection has been to perform tiling on expression trees. This was initially done using dynamic programming [3, 36] for a variety of machine models including stack machines, multi-register machines, infinite register machines, and superscalar machines [7]. These techniques have been further developed to yield code-generator generators [21, 9] which take a declarative specification of an architecture and, at compiler-compile time, generate an instruction selector. These code-generator generators either perform the dynamic programming at compile time [2, 16, 14] or use BURS (bottom-up rewrite system) tree parsing theory [34, 32] to move the dynamic programming to compiler-compile time [17, 35]. In this paper we describe the NOLTIS algorithm, which uses an optimal tree matcher to find a near-optimal tiling of an expression DAG. Although we use a simple compile-time dynamic programming matcher, the NOLTIS algorithm could also easily use a BURS approach to matching.

Tiling expression DAGs is significantly more difficult than tiling expression trees. DAG tiling has been shown to be NP-complete for one-register machines [8] and for two-address, infinite register machine models [4]. Two-address machines have instructions of the form $r_i \leftarrow r_i \text{ op } r_j$ and $r_i \leftarrow r_j$. Since one of the source operands gets overwritten, the difficulty lies in minimizing the number of moves inserted to prevent values from being overwritten. Even with infinite registers and simple, single node tiles, the move minimization problem is NP-complete although approximation algorithms exist [4]. DAG tiling remains difficult on a three-address, infinite register machine if the exterior tile nodes have labels that must match [33]. These labels may correspond to value storage locations (e.g. register classes or memory) or to value types. Such labels are unnecessary if instruction selection is separated from register allocation and if the IR has already fully determined the value types of edges in the expression DAG. However, we show in Section 3 that the problem remains NP-complete even without labels.

Although DAG tiling is NP-complete in general, for some tile sets it can be solved in polynomial time [15]. If a tree tiling algorithm is adapted to tile a DAG and a DAG optimal tile set is used to perform the tiling, the result is an optimal tiling of the DAG. Although the tile sets for several architectures were found to be DAG optimal in [15], these tile sets used a simple cost model and the DAG optimality of the tile set is not preserved if a more complex cost model, such as code size, is used. For example, if the tiles in Figure 1 all had unit cost, they would be DAG optimal, but with the cost metric shown in Figure 1 they are not.

Traditionally, DAG tiling is performed by using a heuristic to break up the DAG into a forest of expression trees [5]. More heavyweight solutions, which solve the problem optimally, include using binate covering [27, 28], using constraint logic programming [26], using integer linear programming [31] or performing exhaustive search [23]. In addition, we describe a 0-1 integer programming representation of the problem in Section 5. These techniques all exhibit worst-case exponential behavior. Although these techniques may be desirable when code quality is of utmost importance and compile-time costs are immaterial, we believe that our linear time, near-optimal algorithm provides excellent code quality without sacrificing compile-time performance scalability.

An alternative, non-tiling, method of instruction selection, which is better suited for linear, as opposed to tree-like, IRs, is to incorporate instruction selection into peephole optimization [12, 18, 19, 24, 10]. In peephole optimization [30], pattern matching transformations are performed over a small window of instructions, the ‘‘peephole.’’ This window may be either a physical window, where the instructions considered are only those scheduled next to each

other in the current instruction list, or a logical window where the instructions considered are just those that are data or control related to the instruction currently being scanned. When performing peephole-based instruction selection, the peepholer simply converts a window of IR operations into target-specific instructions. If a logical window is being used, then this technique can be considered a heuristic method for tiling a DAG.

Instruction selection algorithms have been successfully adapted to solve the technology mapping problem in the automated circuit design domain [25]. Many domain-specific extensions to the basic tiling algorithm have been proposed (see [22, 13] for references), but, to the best of our knowledge, all DAG tiling algorithms proposed in this area have resorted to simple, domain-specific, heuristics for decomposing the DAG into trees before performing the tiling.

3. PROBLEM DESCRIPTION

Given an expression DAG which represents the computation of a basic block and a set of architecture specific instruction tiles, we wish to find an optimal tiling of the DAG which corresponds to the minimum cost instruction sequence. The expression DAG consists of nodes representing operations (such as add or load) and operands (such as a constant or memory location). We refer to a node with multiple parents as a *shared node*. The set of tiles consists of a collection of expression trees each with an assigned cost. If a leaf of an expression tree is not an operand, it is assumed that the inputs for that leaf node will be available from a register¹. Similarly, the output of the tree is assumed to be written to a register. A tile matches a node in the DAG if the root of the tile is the same kind of node as the DAG node and the subtrees of the tile recursively match the children of the DAG node. In order for a tiling to be legal and complete, the inputs of each tile must be available as the outputs of other tiles in the tiling, and all the root nodes of the DAG (those nodes with zero in-degree) must be matched to tiles. The optimal tiling is the legal and complete tiling where the sum of the costs of the tiles is minimized. More formally, we define an optimal instruction tiling as follows:

Definition Let K be a set of node kinds; $G = (V, E)$ be a directed acyclic graph where each node $v \in V$ has a kind $k(v) \in K$, a set of children $ch(v) \in 2^V$ such that $\forall c \in ch(v) (v \rightarrow c) \in E$, and a unique ordering of its children nodes $o_v : ch(v) \rightarrow \{1, 2, \dots, |ch(v)|\}$; T be a set of tree tiles $t_i = (V_i, E_i)$ where similarly every node $v_i \in V_i$ has a kind $k(v_i) \in K \cup \{\circ\}$ such that $k(v_i) = \circ$ implies $outdegree(v_i) = 0$ (nodes with kind \circ denote the edge of a tile and, instead of corresponding to an operation or operand, serve to link tiles together), children nodes $ch(v_i) \in 2^{V_i}$, and an ordering o_{v_i} ; and $cost : T \rightarrow \mathbb{Z}^+$ be a cost function which assigns a cost to each tree tile. We say a node $v \in V$ matches tree t_i with root $r \in V_i$ iff $k(v) = k(r)$, $|ch(v)| = |ch(r)|$, and, for all $c \in ch(v)$ and $c_i \in ch(r)$, $o_v(c) = o_r(c_i)$ implies that either $k(c_i) = \circ$ or c matches the tree rooted at c_i . For a given matching of v and t_i and a tree tile node $v_i \in V_i$, we define $m_{v,t_i} : V_i \rightarrow V$ to return the node in V that matches with the subtree rooted at v_i . A mapping $f : V \rightarrow 2^T$ from each DAG node to a set of tree tiles is legal iff $\forall v \in V$:

$$t_i \in f(v) \implies v \text{ matches } t_i$$

$$indegree(v) = 0 \implies |f(v)| > 0$$

$$\forall t_i \in f(v), \forall v_i \in t_i, k(v_i) = \circ \implies |f(m_{v,t_i}(v_i))| > 0$$

¹These are unallocated temporaries, not actual hard registers.

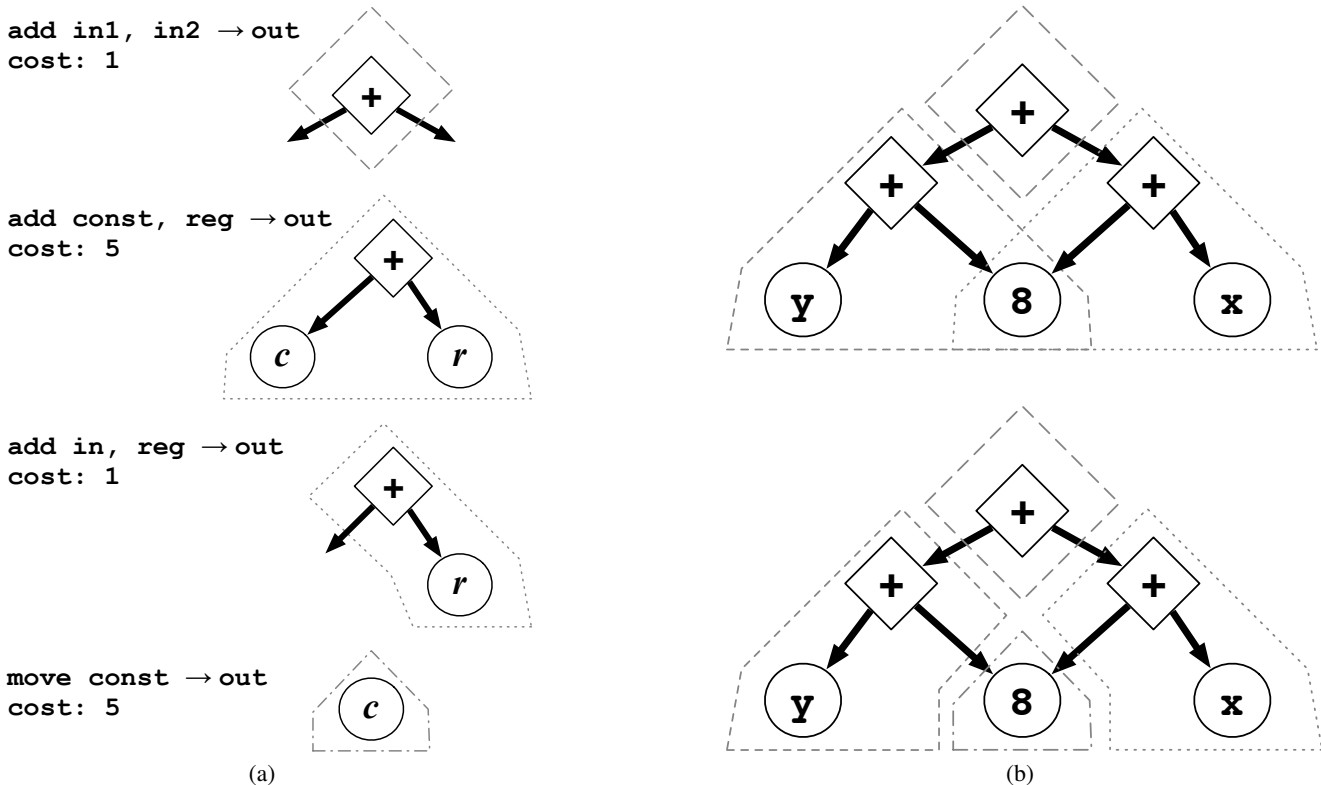


Figure 1: An example of instruction selection on a DAG. (a) The tile set used (commutative tiles are omitted). (b) Two possible tilings. In a simple cost model where every tile has a unit cost the top tiling would be optimal, but with the cost model shown the lower tiling is optimal.

An *optimal instruction tiling* is a legal mapping f which minimizes

$$\sum_{v \in V} \sum_{t_i \in f(v)} \text{cost}(t_i)$$

In some versions of the instruction tiling problem, the name of the storage location a tile writes or reads is important. For example, some tiles might write to memory or read from a specific register class. In this case, there is an additional constraint that a tile's inputs must not only match with other tiles' outputs, but the names of the respective input and output must also match. In practice, if instruction selection is performed independently of register allocation, the names of storage locations are irrelevant. Although previous proofs of the hardness of instruction selection have relied on complications such as storage location naming [33] or two-address instructions [4], we now show that even without these restrictions the problem remains NP-complete.

THEOREM 3.1. *The optimal instruction tiling problem (is there an optimal instruction tiling of cost less than k_{const} ?) is NP-complete.*

PROOF. Inspired by [33], we perform a reduction from satisfiability of Boolean expressions [20]. Given a Boolean expression consisting of variables $u \in U$ and Boolean connectives $\{\vee, \wedge, \neg\}$, we construct an instance of the optimal instruction tiling problem as follows:

Let the set of node kinds K be $\{\vee, \wedge, \neg, \square, R, v\}$. We refer to nodes with kind \square as box nodes. For every variable $u \in U$, create two nodes u_1 and u_2 and a directed edge $(u_1 \rightarrow u_2)$ in G such that $k(u_1) = \square$ and $k(u_2) = v$. Similarly, for every Boolean operator

op create two nodes op_1 and op_2 and a directed edge $(op_1 \rightarrow op_2)$ such that $k(op_1) = \square$ and $k(op_2)$ is the corresponding operation. Next, for every operation $a op b$ create edges $(op_2 \rightarrow a_1)$ and $(op_2 \rightarrow b_1)$ where $k(a_1) = k(b_1) = \square$ (in the case of the unary \neg operation a single edge is created). Note the ordering of child nodes is irrelevant since the Boolean operators are commutative. Finally, create a node r and edge $(r \rightarrow op)$ such that $k(r) = R$ and op is the root operation of the expression. An example of such a DAG is shown in Figure 2(a). Note that the only nodes with potentially more than one parent in this DAG are those box nodes corresponding to variables.

Now let the tree tile set T be as shown in Figure 2(b) where each tile contains a single non-box node and has unit cost. These tiles are designed so that it can be shown that a truth assignment of a Boolean expression corresponds directly with a legal tiling of a DAG constructed as described above. If a variable is true, then its corresponding node is covered with the tile $v : T$, otherwise it is covered with $v : F$. The rest of the tiling is built up in the natural way suggested from the tile names in Figure 2(b). This tiling is optimal since every leaf node of the DAG will have exactly one tile covering it (corresponding to the truth assignment of that variable) and, since the parents of leaf nodes are the only shared nodes in the DAG (they may have multiple parents), no other non-box node in the DAG can be covered by more than one tile in this tiling. Therefore, the cost of the tiling is equal to the number of non-box nodes and is optimal.

Given an optimal tiling of a DAG derived from a Boolean expression, if the cost of the tiling is equal to the number of non-box nodes, then we can easily construct a truth assignment that satisfies

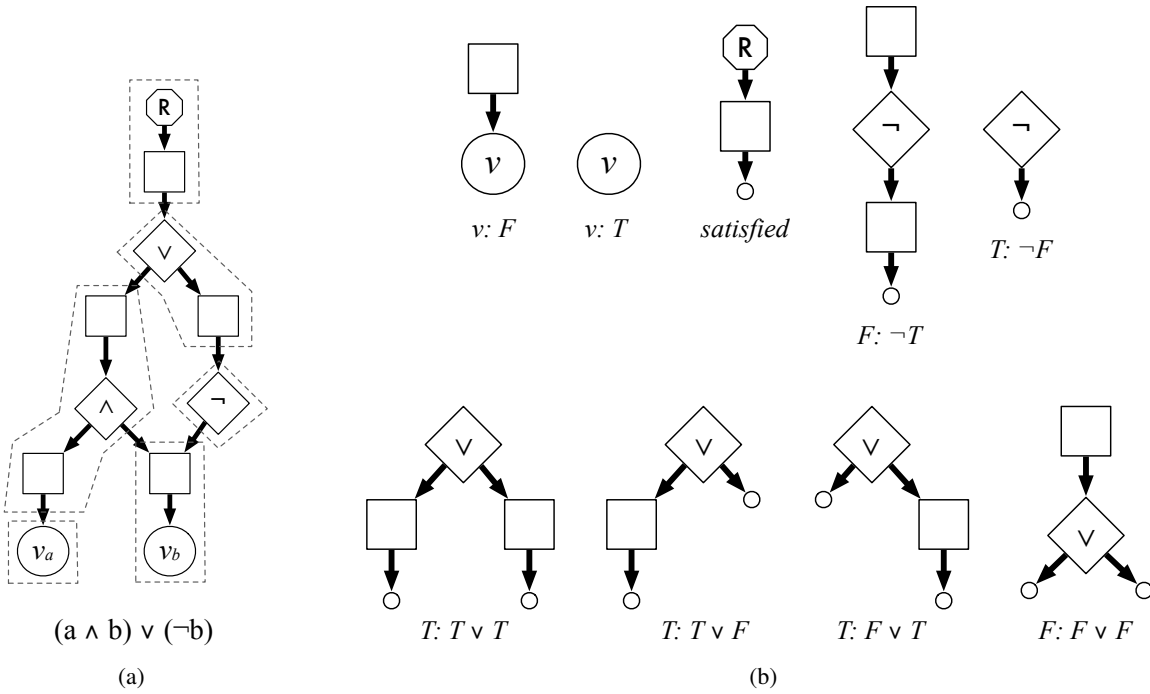


Figure 2: (a) An example of an expression DAG that represents a Boolean expression. (b) The tiles used to cover such an expression DAG. Each tile has unit cost. The tiles representing \wedge are omitted, but are similar to the \vee tiles with the two middle tiles having an additional box node at the root.

the expression by observing the tiles used to cover the leaves of the DAG. If the cost of the tiling is greater than the number of non-box nodes then the expression is not satisfiable. If it were, a cheaper tiling would have to exist.

We have shown the boolean satisfiability reduces to the optimal instruction tiling problem, and, therefore, the optimal instruction tiling problem is NP-complete. \square

4. NOLTIS

The NP-complete nature of the optimal instruction tiling problem necessitates the use of heuristics when performing instruction selection. A common approach is to first decompose the DAG into a forest of trees and then use an optimal tree tiling algorithm to tile each tree. Every common subexpression in the DAG is therefore at the root of a tree in the forest. However, as we will show in Section 7, this approach is not as successful as algorithms which work directly on the DAG. For example, if all the tiles in Figure 1 were assigned a unit cost, the tree decomposition solution would be suboptimal.

In this section we present NOLTIS, a linear-time algorithm which obtains near-optimal tilings of expression DAGs. The algorithm applies tree tiling directly to the DAG without first performing tree decomposition, uses this tiling to decide which parts of the DAG can be productively decomposed into trees, and then retiles the partially decomposed DAG.

First we apply dynamic programming on the DAG ignoring the presence of shared nodes using the procedure BOTTOMUPDP from Listing 1. Conceptually, we are tiling the tree that would be formed if every shared node (and its descendants) was duplicated to convert the DAG into a potentially exponentially larger tree. However, the algorithm remains linear since each node is visited only once. Once dynamic programming has labeled each node with the best tile for that node, a top down pass, TOPDOWNSELECT, creates a tiling

of the DAG. The existence of shared nodes may result in a tiling where nodes are covered by multiple tiles (i.e., the tiles overlap). However, since no node will be at the root of two tiles (this would imply that the exact same value would be computed twice), the number of tiles in a tiling is proportional to the number of nodes. Consequently, the top down pass, which traverses tiles, has linear time complexity.

The tiling found by the first tiling pass ignores the impact of shared nodes in the DAG and therefore may have excessive amounts of overlap. In the next step of the algorithm, we identify shared nodes where removing overlap locally improves the overall tiling. These nodes are added to the *fixedNodes* set. We then perform another tiling pass. In this pass, tiles are prohibited from spanning nodes in the *fixedNodes* set; these nodes must be matched to the root of a tile.

The procedure IMPROVECSEDECISIONS (Listing 2) is used to determine if a shared node should be fixed. For each shared node n with overlap we compute the cost of the overlap at n using the GETOVERLAPCOST function in Listing 3. This function computes the cost of the local area of overlap at n . Note that, in the rare case that the area of overlap subsumes another shared node, it is possible that IMPROVECSEDECISIONS will have super-linear time complexity; however, this can be addressed through the use of memoization, a detail which is not shown in the pseudocode.

The next step is to compute the cost which would be incurred if the tiles covering n were cut so that only a single tile, rooted at n , covered n . The cost of the tile tree rooted at n can be determined from the results of dynamic programming. To this cost we add the costs of cutting the tiles currently covering n , which are computed using the function GETTILECUTCOST shown in Listing 4. In determining the cost of cutting a tile t with root r at node n , we consider every tile which also matches at r and has n as an edge node. We then compute the cost difference between using this tile to match

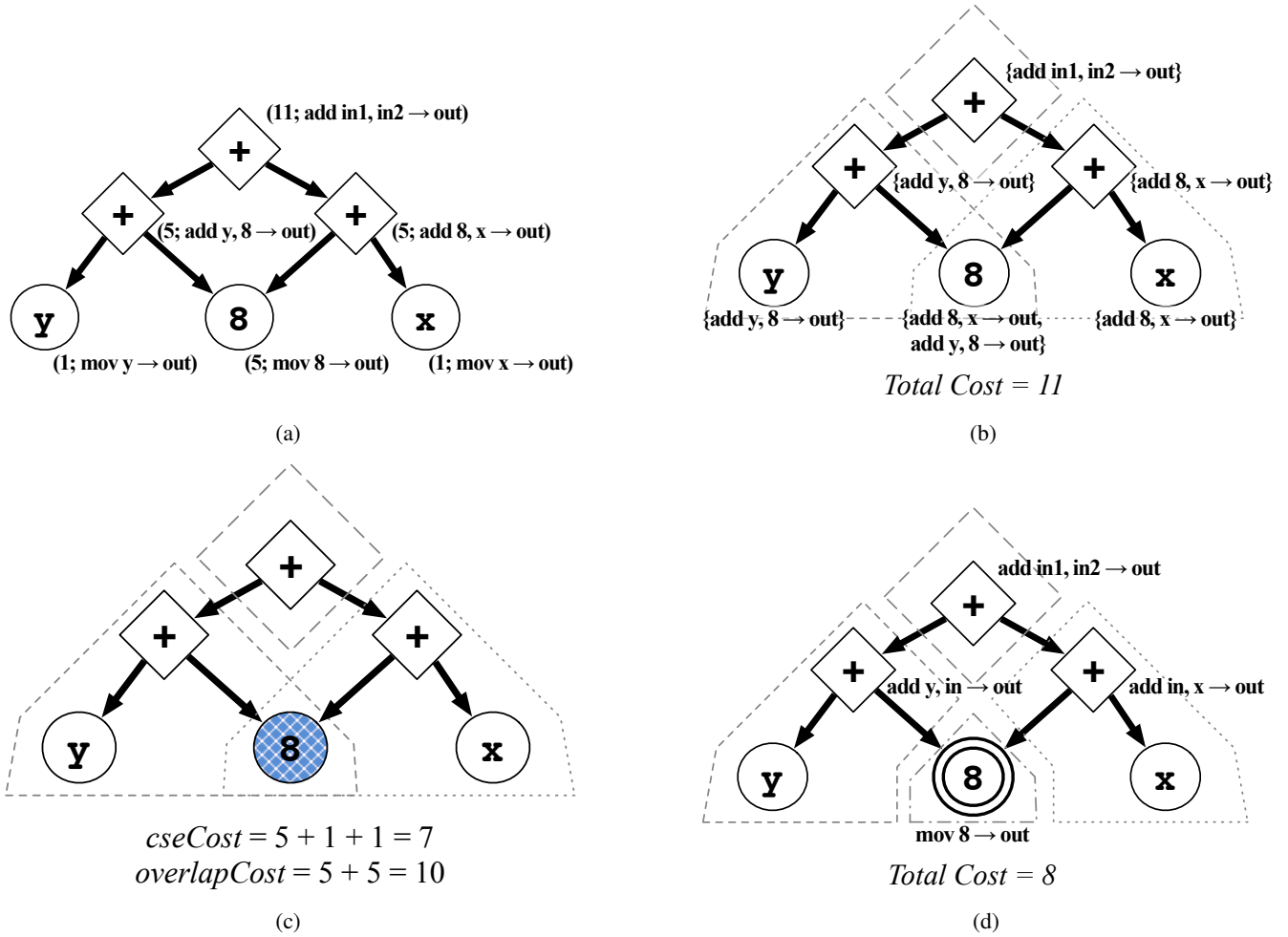


Figure 3: The application of the NOLTIS algorithm to the example from Figure 1. (a) BOTTOMPUPDP computes the dynamic programming solution for the DAG, initializing *bestChoiceForNode*. (b) TOPDOWNSELECT determines a tiling from the dynamic programming solution, initializing *coveringTiles*. (c) IMPROVECSEDECISIONS evaluates the shared node and determines it should be fixed. (d) The dynamic programming is then recomputed with the requirement that the fixed node not be overlapped and the optimal solution is found.

r and using t . We choose the minimum cost difference as the cost of cutting the tile. If the cost of the current overlapping tiling is more than the cost of removing the overlap and cutting the tiles, then we have found a local transformation which improves the existing tiling. Instead of immediately applying this transformation, we choose to fix node n , disabling overlap when we compute a new tiling. This results in a potentially better solution as the new tiling need not be limited to tiles rooted at r . Figure 3 shows the execution of the NOLTIS algorithm on the example from Figure 1.

The NOLTIS algorithm is not optimal as it depends upon several assumptions which do not necessarily hold. We assume that it is always possible to cut tiles at a shared node without affecting the tileability of the DAG. We assume that the best place to cut tiles to eliminate overlap is at a shared node. We assume the decision to fix a shared node can be made independently of other shared nodes. When deciding to fix a shared node we assume we can represent the impact of fixing the node by examining simple tile cuts. Despite these assumptions, in practice the NOLTIS algorithm achieves near-optimal results.

5. 0-1 PROGRAMMING SOLUTION

In order to establish the near-optimality of our algorithm, we formulate the instruction tiling problem as a 0-1 integer program which can be solved to optimality using a commercial solver. The formulation of the problem is straightforward. For every node i and tile j we have binary variable $M_{i,j}$ which is one if tile j matches node i (the root of tile j is at node i) in the tiling, zero otherwise. Let cost_j be the cost of tile j , roots be the root nodes of the DAG, and $\text{edgeNodes}(i, j)$ be the nodes at the edge of tile j when rooted at node i , then the optimal instruction tiling problem is:

$$\min \sum_{i,j} \text{cost}_j M_{i,j}$$

subject to

$$\forall i \in \text{roots} \sum_j M_{i,j} \geq 1 \quad (1)$$

$$\forall i,j \forall i' \in \text{edgeNodes}(i,j) M_{i,j} - \sum_{j'} M_{i',j'} \leq 0 \quad (2)$$

Listing 1 Dynamic programming instruction selection with modifications for near-optimal DAG selection

```

1: DAG : expression DAG
2: bestChoiceForNode : Node  $\rightarrow$  (Tile  $\times$  int)
3: fixedNodes : set of Node
4: matchedTiles : set of Tile
5: coveringTiles : Node  $\rightarrow$  set of Tile

6: procedure SELECT
7:   fixedNodes  $\leftarrow$  {}
8:   BOTTOMUPDP()  $\triangleright$  initializes bestChoiceForNode
9:   TOPDOWNSELECT()  $\triangleright$  initializes coveringTiles
10:  IMPROVECSEDECISIONS()  $\triangleright$  initializes fixedNodes
11:  BOTTOMUPDP()  $\triangleright$  uses fixedNodes
12:  TOPDOWNSELECT()  $\triangleright$  puts final tiling in matchedTiles

13: procedure BOTTOMUPDP
14:  for n  $\in$  reverseTopologicalSort(DAG) do
15:    bestChoiceForNode[n].cost  $\leftarrow$   $\infty$ 
16:    for tn  $\in$  matchingTiles(n) do
17:      if  $\neg$ hasInteriorFixedNode(tn, fixedNodes) then
18:        val  $\leftarrow$  cost(t) +
19:           $\sum_{n' \in \text{edgeNodes}(t_n)} \text{bestChoiceForNode}[n'].cost$ 
20:        if val < bestChoiceForNode[n].cost then
21:          bestChoiceForNode[n].cost  $\leftarrow$  val
22:          bestChoiceForNode[n].tile  $\leftarrow$  tn

22: procedure TOPDOWNSELECT
23:  matchedTiles.clear()
24:  coveringTiles.clear()
25:  q.push(roots(DAG))
26:  while  $\neg$ q.empty() do
27:    n  $\leftarrow$  q.pop()
28:    bestTile  $\leftarrow$  bestChoiceForNode[n].tile
29:    matchedTiles.add(bestTile)
30:    for every node nt covered by bestTile do
31:      coveringTiles[nt].add(bestTile)
32:    for n'  $\in$  edgeNodes(bestTile) do
33:      q.push(n')

```

where (1) requires that the root nodes of the DAG be matched to tiles and (2) requires that if a tile matches a node, then all of the inputs to that tile must be matched to tiles.

6. IMPLEMENTATION

We have implemented our algorithm in the LLVM 2.1 [29] compiler infrastructure targeting the Intel x86 architecture on the Ubuntu 7.10 Linux operating system. The default LLVM instruction selector constructs an expression DAG of target independent nodes and then performs a maximal munch algorithm [6]. Tiles are selected from the top-down. The largest tile (the tile that covers the most nodes) is greedily selected. Tile costs are only used to break ties. We have modified the LLVM algorithm to use code size when breaking ties.

In addition to the default LLVM algorithm and the NOLTIS algorithm, we have implemented three other algorithms:

cse-all The expression DAG is completely decomposed into trees and dynamic programming is performed on each tree. That is, every shared node is fixed. This is the conventional method for applying tree tiling to a DAG [5].

Listing 2 Given a DAG matching that ignored the effect of shared nodes, decide if the solution would be improved by pulling shared nodes out into common subexpressions (eliminating tile overlap).

```

1: procedure IMPROVECSEDECISIONS
2:  for n  $\in$  sharedNodes(DAG) do
3:    if coveringTiles[n].size() > 1 then  $\triangleright$  has overlap
4:      overlapCost  $\leftarrow$  getOverlapCost(n, coveringTiles)
5:      cseCost  $\leftarrow$  bestChoiceForNode[n].cost
6:      for tn  $\in$  coveringTiles[n] do
7:        cseCost  $\leftarrow$  cseCost + getTileCutCost(tn, n)
8:      if cseCost < overlapCost then
9:        fixedNodes.add(n)

```

Listing 3 Given a shared node *n* with overlapping tiles, compute the cost of the tree of tiles rooted at the tiles overlapping *n* without double counting areas where the tile trees do not overlap.

```

1: function GETOVERLAPCOST(n)
2:  cost  $\leftarrow$  0
3:  seen  $\leftarrow$  {}
4:  for t  $\in$  coveringTiles[n] do
5:    q.push(t)
6:    seen.add(t)
7:  while  $\neg$ q.empty() do
8:    t  $\leftarrow$  q.pop()
9:    cost  $\leftarrow$  cost + cost(tile)
10:   for n'  $\in$  edgeNodes(t) do
11:     if n' is reachable from n then
12:       t'  $\leftarrow$  bestChoiceForNode[t'].tile
13:       if coveringTiles[n'].size() = 1 then
14:         cost  $\leftarrow$  cost + bestChoiceForNode[n'].cost
15:       else if t'  $\notin$  seen then
16:         seen.add(t')
17:         q.push(t')
18:  return cost

```

cse-leaves The expression DAG is partially decomposed into trees and dynamic programming is performed. If the subgraph rooted at a shared node can be fully covered by a single tile, the shared node remain unaltered, otherwise shared nodes become roots of trees to be tiled. That is, shared nodes are fixed unless they represent an expression that can be fully covered by a single tile.

cse-none The expression DAG is not decomposed into trees and dynamic programming is performed. That is, no shared nodes are fixed (this is equivalent to the solution found before the IMPROVECSEDECISIONS procedure is executed in the NOLTIS algorithm).

All algorithms use the same tile set. The cost of each tile is the size in bytes of the corresponding x86 instruction(s). We do not allow tiles to overlap memory operations (i.e., a load or store node in the expression DAG will only be executed once). Similarly, as an implementation detail², overlap of function call addresses is not allowed. Valueless token edges enforce ordering dependencies in the expression DAG. Despite the two-address nature of the x86 architecture, all tiles represent three-address instructions. A pass after instruction selection converts the code to two-address form. A scheduling pass, which converts the code from DAG form into

²LLVM's support for different relocation models requires that function call addresses be part of the call instruction.

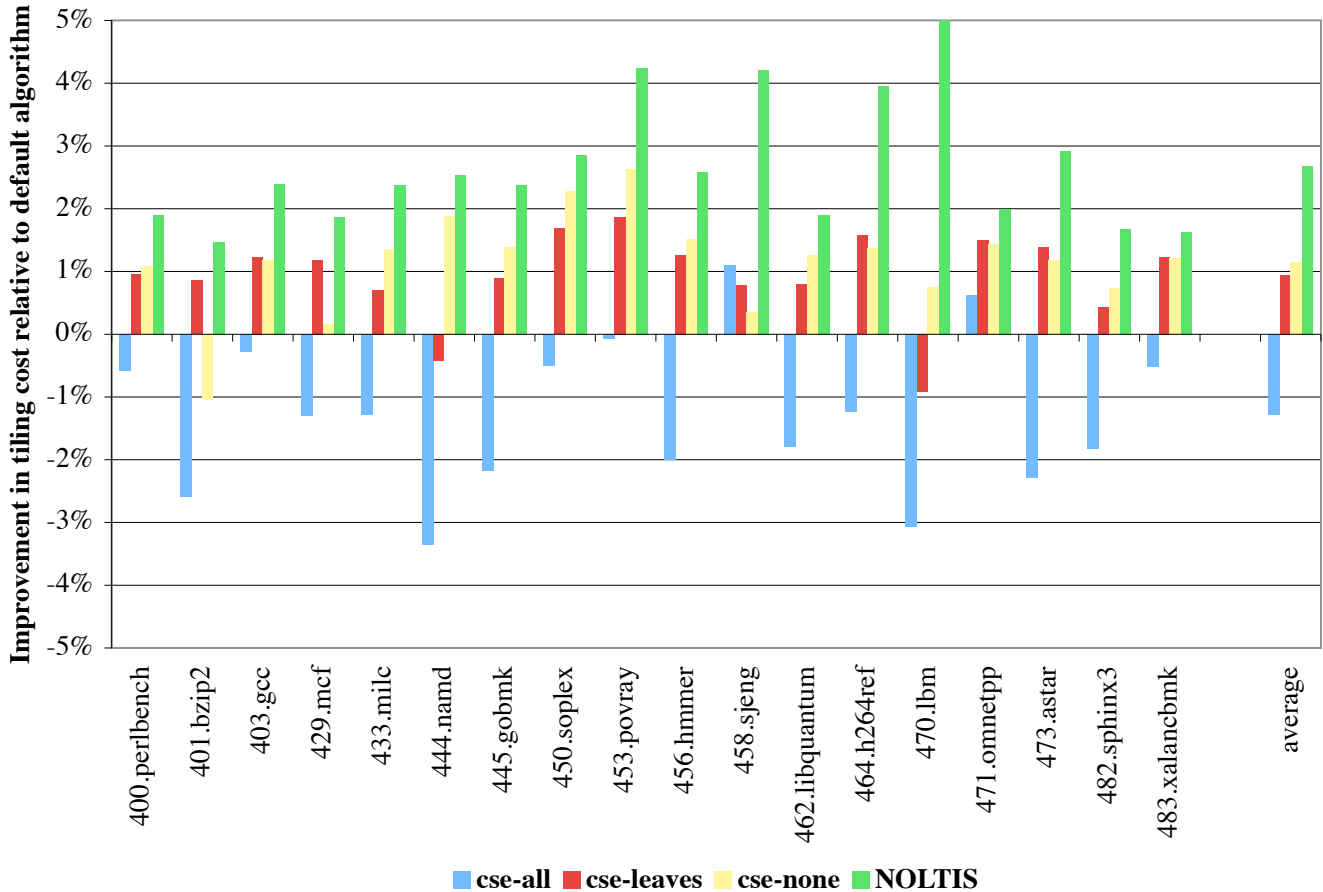


Figure 4: The improvement in tiling cost relative to the default maximal munch algorithm for individual SPEC CPU2006 benchmarks. The average improvements of the *cse-all*, *cse-leaves*, *cse-none*, and NOLTIS algorithms are -1.28%, 0.94%, 1.15%, and 2.68% respectively.

an assembly listing, attempts to minimize the register pressure of the schedule using Sethi-Ullman numbering [36].

7. RESULTS

We evaluate the various instruction selection algorithms by compiling the C and C++ benchmarks of the SPEC CPU2006 [39], MediaBench [37], MiBench [38], and VersaBench [40] benchmark suites and observing both the immediate, post-selection cost of the tiling and the final code size of the benchmark. We evaluate the optimality of the NOLTIS algorithm, demonstrate its superiority compared to existing heuristics, investigate its impact on the code size of fully compiled code, and describe its compile-time behavior.

7.1 Optimality

In order to determine an optimal solution for an expression DAG, we create a 0-1 integer programming problem as described in Section 5 and then solve it using ILOG CPLEX 10.0 [11]. We evaluated all the basic blocks of the SPEC CPU2006 benchmarks, resulting in nearly half million tiling problems. We utilized a cluster of Pentium 4 machines ranging in speed from 2.8Ghz to 3.0Ghz to solve the problems. CPLEX was able to find a provably optimal solution within a 15 minute time limit for 99.8% of the tiling problems. Of the problems with provably optimal solutions, the NOLTIS algorithm successfully found the optimal solution 99.7% of the time. Furthermore, suboptimal solutions were typically very

close to optimal (only a few bytes larger). Of the 0.2% of problems where CPLEX did not find a provably optimal solution, the NOLTIS algorithm found a solution as good as, and in some cases better than, the best solution found by CPLEX 75% of the time implying our algorithm is effective even for very difficult tiling problems.

The overall improvement obtained by using the best CPLEX solution versus using the NOLTIS algorithm was a negligible 0.05%. We feel these results clearly demonstrate that the NOLTIS algorithm is, in fact, near-optimal.

7.2 Comparison of Algorithms

In addition to being near-optimal, the NOLTIS algorithm provides significantly better solutions to the tiling problem than conventional heuristics. Detailed results for the SPEC CPU2006 benchmarks are shown in Figure 4 and average improvements are shown in Figure 5. The *cse-all* algorithm, despite finding an optimal tiling for each tree in the full tree decomposition of the DAG, performs poorly relative to all other algorithms suggesting that DAG tiling algorithms are necessary for maximum code quality. Both the *cse-leaves* and *cse-none* algorithms benefit from using dynamic programming and outperform the greedy algorithm, although neither algorithm is clearly superior to the other. The NOLTIS algorithm, as expected, significantly outperforms the other algorithms and has the best improvement in every benchmark.

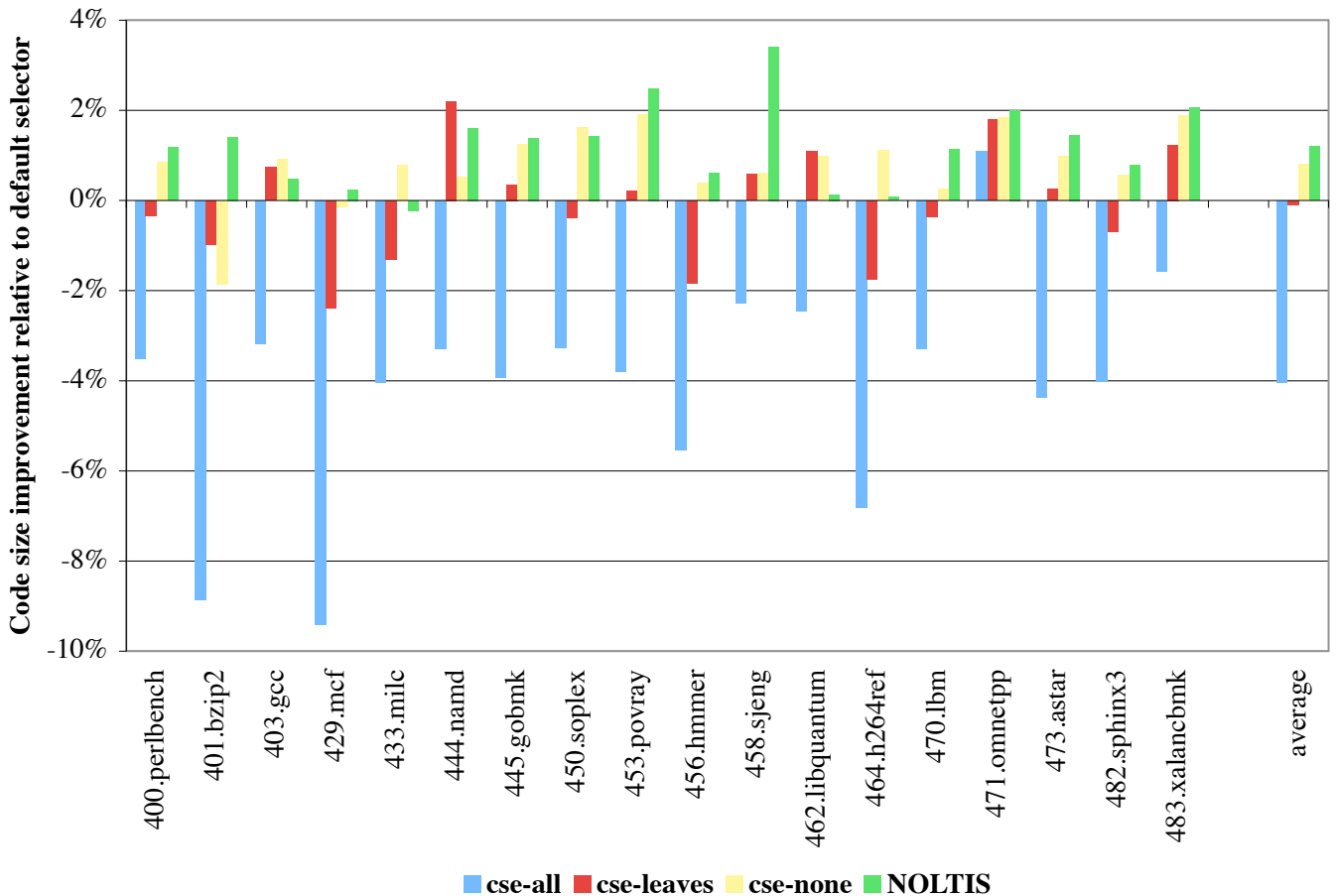


Figure 6: The final code size improvement relative to the default maximal munch algorithm for individual SPEC CPU2006 benchmarks. The average improvements of the *cse-all*, *cse-leaves*, *cse-none*, and NOLTIS algorithms are -4.03%, -0.09%, 0.81%, and 1.20% respectively. The average improvement of the NOLTIS algorithm relative to the more traditional *cse-all* algorithm is 4.42%.

7.3 Impact on Code Size

Instruction tiling is only one component of code generation. The two-address conversion pass, scheduling pass, and register allocation pass all further impact the final quality of the compiled code. Final code size results for the SPEC CPU2006 benchmarks are shown in Figure 6 and average improvements are shown in Figure 7.

Although the average code size improvements exhibited by the NOLTIS algorithm may seem marginal, even such seemingly small code size reductions may be significant when targeting highly constrained embedded architectures. Furthermore, it is important to note that these results are relative to an algorithm that has already been adapted to work directly on expression DAGs. Compared to the classical textbook approach of tree decomposition (the *cse-all* algorithm), the NOLTIS algorithm exhibits an overall average code size improvement of 5.1%.

The mixed nature of the final code size results appears to be mostly caused by the interaction with the register allocator, in particular the number of loads and stores the allocator inserts. Decomposing the graph into trees results in the creation of temporaries with multiple uses. These potentially long-lived temporaries result in more register pressure and more values must be spilled to memory. Hence the *cse-all* algorithm performs particularly poorly. Allowing unlimited overlap can also have a negative effect on register allocation as the inputs of overlapping tiles are also potentially

long lived temporaries. Another factor influencing register allocation is the number of tiles. If more, smaller, tiles are used, there are correspondingly more temporaries to allocate.

Ultimately, the interaction between instruction selection and register allocation cannot be easily characterized and is beyond the scope of this work. It is likely that architectures with complex instruction sets but plentiful (e.g., more than eight) registers would see more benefit from the NOLTIS algorithm. Furthermore, given a framework for characterizing the interaction between instruction selection and register allocation, the near-optimality of the NOLTIS algorithm would make it the natural choice for performing tiling.

7.4 Compile Time Performance

As shown in Figure 8, the two pass nature of the NOLTIS algorithm means that its running time is slightly more than twice that of the other dynamic programming based algorithms. The dynamic programming algorithms are approximately 30% slower than the greedy algorithm since they must perform a tile selection operation at every node of the expression DAG. The greedy algorithm can ignore any nodes which are completely covered by the greedily selected tile.

8. LIMITATIONS AND FUTURE WORK

Instruction selection algorithms have been used successfully to solve the technology mapping problem in the automated circuit de-

Listing 4 Given a tile t and node n , determine the cost of cutting t at node n so that the root of t remains the same but n becomes an edge node.

```

1: function GETTILECUTCOST( $t, n$ )
2:    $bestCost \leftarrow \infty$ 
3:    $r \leftarrow root(tile)$ 
4:   for  $t' \in matchingTiles(r)$  do
5:     if  $n \in edgeNodes(t')$  then
6:        $cost \leftarrow cost(t')$ 
7:       for  $n' \in edgeNodes(t') \wedge n' \neq n$  do
8:          $cost \leftarrow cost + bestChoiceForNode[n'].cost$ 
9:       if  $cost < bestCost$  then
10:         $bestCost \leftarrow cost$ 
11:   for  $n' \in edgeNodes(t)$  do  $\triangleright$  Subtract edge costs of
    original tile
12:   if path  $r \rightsquigarrow n' \in t$  does not contain  $n$  then
13:      $bestCost \leftarrow bestCost - bestChoiceForNode[n'].cost$ 
14:   return  $bestCost$ 

```

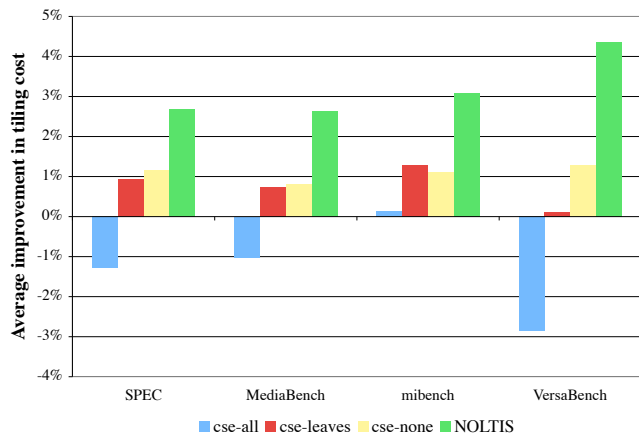


Figure 5: The average improvement in tiling cost relative to the default maximal munch algorithm for four benchmark suites.

sign domain. It remains an open question whether the NOLTIS algorithm can be successfully adapted to this domain where multiple optimization goals (area, delay, routing resources) must be simultaneously addressed.

Although the NOLTIS algorithm is linear in the size of the program, its running time is largely determined by how efficiently the matching of a single node to a set of tiles can be performed. The algorithm, as we have presented it, uses a simple, but inefficient, matching algorithm. More efficient algorithms, such as tree parsing, exist [17, 35, 32, 2] and should be used in a production implementation. Additionally, the second pass of dynamic programming could be made more efficient by intelligently recomputing only portions of the DAG.

The classical representation of instruction selection as a tiling problem relies on instructions being represented by tree tiles. In some cases, such as with single instruction multiple data (SIMD) instructions and instructions with side-effects, an instruction cannot be represented as a tree of data dependences. Additional, non-tiling, techniques are required to handle such instructions.

The abstract machine model used by our tiling algorithms is a three-address, infinite register machine. Finding a linear time, near-optimal algorithm that does not depend upon these assumptions remains an open problem. Given the hardness of the register alloca-

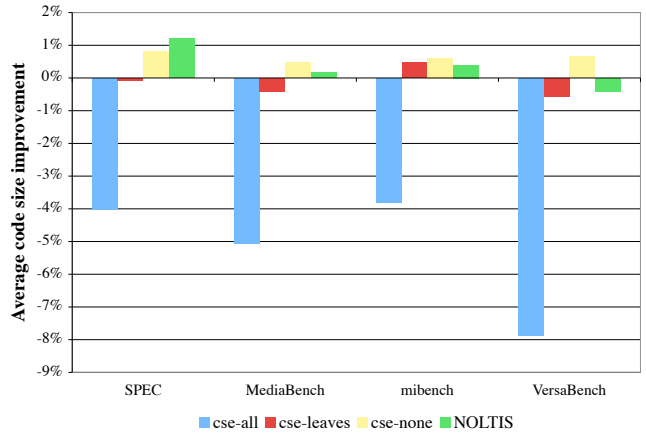


Figure 7: The average final code size improvement relative to the default maximal munch algorithm for four benchmark suites.

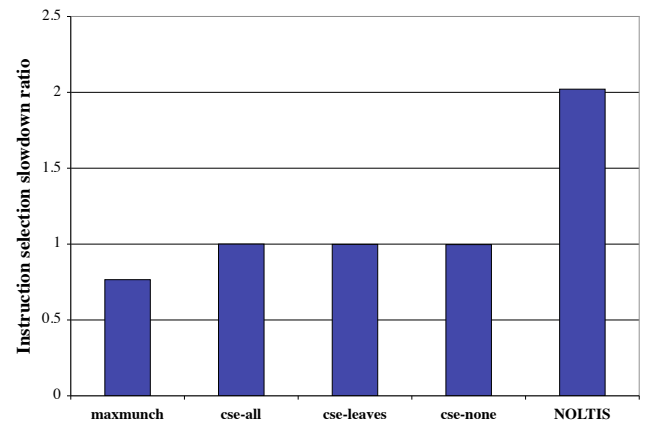


Figure 8: The total slowdown of each instruction selection algorithm relative to the *cse-all* algorithm. The two pass nature of the NOLTIS algorithm results in it taking slightly more than twice as long as the other dynamic programming based algorithms.

tion problem, it seems unlikely that such an algorithm exists. However, it may be possible to construct a framework which integrates register allocation and instruction selection. The two passes would then work cooperatively with the instruction selector guiding register allocation decisions and vice versa. For example, the instruction selector might generate an approximate tiling which the register allocator is responsible for finalizing based on register availability. Or the register allocator might provide feedback to the instruction selector which changes the costs of tiles. We believe that NOLTIS would be a valuable part of such a framework.

9. SUMMARY

In this paper we have described NOLTIS, an easy to implement, fast, and effective algorithm for finding an instruction tiling of an expression DAG. We have shown empirically that the NOLTIS algorithm achieves near-optimal results and significantly outperforms existing tiling heuristics. Although the interaction between instruction selection and register allocation bears further study, we have shown that NOLTIS is capable of improving code size compared to existing techniques.

10. ACKNOWLEDGEMENTS

This research was sponsored in part by the National Science Foundation under grant CCF-0702640.

11. REFERENCES

- [1] AHO, A., AND ULLMAN, J. Optimization of stright line programs. *SIAM Journal on Computing* 1, 1 (1972), 1–19.
- [2] AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 491–516.
- [3] AHO, A. V., AND JOHNSON, S. C. Optimal code generation for expression trees. *J. ACM* 23, 3 (1976), 488–501.
- [4] AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. Code generation for expressions with common subexpressions. *J. ACM* 24, 1 (1977), 146–160.
- [5] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [6] APPEL, A. W. *Modern Compiler Implementation in Java: Basic Techniques*. Cambridge University Press, 1997.
- [7] BOSE, P. Optimal code generation for expressions on super scalar machines. In *Proc. of the ACM Fall Joint Computer Conf.* (Los Alamitos, CA, USA, 1986), IEEE Computer Society Press, pp. 372–379.
- [8] BRUNO, J., AND SETHI, R. Code generation for a one-register machine. *J. ACM* 23, 3 (1976), 502–510.
- [9] CATTELL, R. G. Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst.* 2, 2 (1980), 173–190.
- [10] COOPER, K. D., AND TORCZON, L. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2004.
- [11] CPLEX. <http://ilog.com/products/cplex>.
- [12] DAVIDSON, J. W., AND FRASER, C. W. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.* 6, 4 (1984), 505–526.
- [13] DE MICHELI, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994, ch. 10.
- [14] EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, L. Beg: a generation for efficient back ends. In *Proc. of ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.* (New York, NY, USA, 1989), ACM Press, pp. 227–237.
- [15] ERTL, M. A. Optimal code selection in dags. In *Proc. of ACM/SIGPLAN-SIGACT Symposium on Principles of Prog. Lang.* (New York, NY, USA, 1999), ACM Press, pp. 242–249.
- [16] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code-generator generator. *ACM Lett. Prog. Lang. Syst.* 1, 3 (1992), 213–226.
- [17] FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.* 27, 4 (1992), 68–76.
- [18] FRASER, C. W., AND WENDT, A. L. Integrating code generation and optimization. In *Proc. of SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1986), ACM Press, pp. 242–248.
- [19] FRASER, C. W., AND WENDT, A. L. Automatic generation of fast optimizing code generators. In *Proc. of ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.* (New York, NY, USA, 1988), ACM Press, pp. 79–84.
- [20] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
- [21] GLANVILLE, R. S., AND GRAHAM, S. L. A new method for compiler code generation. In *Proc. of ACM/SIGPLAN-SIGACT Symposium on Principles of Prog. Lang.* (New York, NY, USA, 1978), ACM Press, pp. 231–254.
- [22] HASSOUN, S., AND SASAO, T., Eds. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
- [23] KESSLER, C., AND BEDNARSKI, A. A dynamic programming approach to optimal integrated code generation. In *Proc. of the ACM/SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems* (New York, NY, USA, 2001), ACM Press, pp. 165–174.
- [24] KESSLER, R. R. Peep: an architectural description driven peephole optimizer. In *Proceedings of SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1984), ACM Press, pp. 106–110.
- [25] KEUTZER, K. DAGON: technology binding and local optimization by dag matching. In *Proc. of the ACM/IEEE Conf. on Design Automation* (New York, NY, USA, 1987), ACM Press, pp. 341–347.
- [26] LEUPERS, R., AND BASHFORD, S. Graph-based code selection techniques for embedded processors. *ACM Trans. Des. Autom. Electron. Syst.* 5, 4 (2000), 794–814.
- [27] LIAO, S., DEVADAS, S., KEUTZER, K., AND TJIANG, S. Instruction selection using binate covering for code size optimization. In *Proc. of the IEEE/ACM Intl. Conf. on Computer-aided Design* (Washington, DC, USA, 1995), IEEE Computer Society, pp. 393–399.
- [28] LIAO, S., KEUTZER, K., TJIANG, S., AND DEVADAS, S. A new viewpoint on code generation for directed acyclic graphs. *ACM Trans. Des. Autom. Electron. Syst.* 3, 1 (1998), 51–75.
- [29] The LLVM compiler infrastructure project. <http://llvm.org>.
- [30] MCKEEMAN, W. M. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444.
- [31] NAIK, M., AND PALSBERG, J. Compiling with code-size constraints. *Trans. on Embedded Computing Sys.* 3, 1 (2004), 163–181.
- [32] PELEGRÍ-LLOPART, E., AND GRAHAM, S. L. Optimal code generation for expression trees: an application burs theory. In *Proc. of ACM/SIGPLAN-SIGACT Symposium on Principles of Prog. Lang.* (New York, NY, USA, 1988), ACM Press, pp. 294–308.
- [33] PROEBSTING, T. Least-cost instruction selection in dags is NP-complete. <http://research.microsoft.com/~toddpro/papers/proof.htm>.
- [34] PROEBSTING, T. A. Simple and efficient burs table generation. In *Proc. of ACM/SIGPLAN Conf. on Prog. Lang. Design and Impl.* (New York, NY, USA, 1992), ACM Press, pp. 331–340.
- [35] PROEBSTING, T. A. Burs automata generation. *ACM Trans. Program. Lang. Syst.* 17, 3 (1995), 461–486.
- [36] SETHI, R., AND ULLMAN, J. D. The generation of optimal code for arithmetic expressions. *J. ACM* 17, 4 (1970), 715–728.
- [37] MediaBench I <http://euler.slu.edu/~fritts/mediabench/>
- [38] MiBench <http://www.eecs.umich.edu/mibench/>
- [39] SPEC CPU2006 <http://www.spec.org/cpu2006/>.
- [40] VersaBench: A Benchmark Suite for Versatile Architectures <http://cag.csail.mit.edu/versabench/>