Submitted for confidential review to: *The 2008 International Symposium on Code Generation and Optimization*

Near-Optimal Instruction Selection on DAGs

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 demonstrated average code size improvements of 1.48%.

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 processors. 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 [4, 8, 33].

In this paper we describe NOLTIS, a **n**ear-**o**ptimal, **l**inear **t**ime **i**nstruction **s**election algorithm for expression DAGs. NOLTIS builds upon 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.

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 [9, 20] 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, 13, 15] or use BURS (bottom-up rewrite system) tree parsing theory [32, 34] to move the dynamic programming to compiler-compile time [16, 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 $r_i \leftarrow r_i$ **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 [14]. 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 [14], 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.

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 [10, 11, 17, 18, 24]. 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.

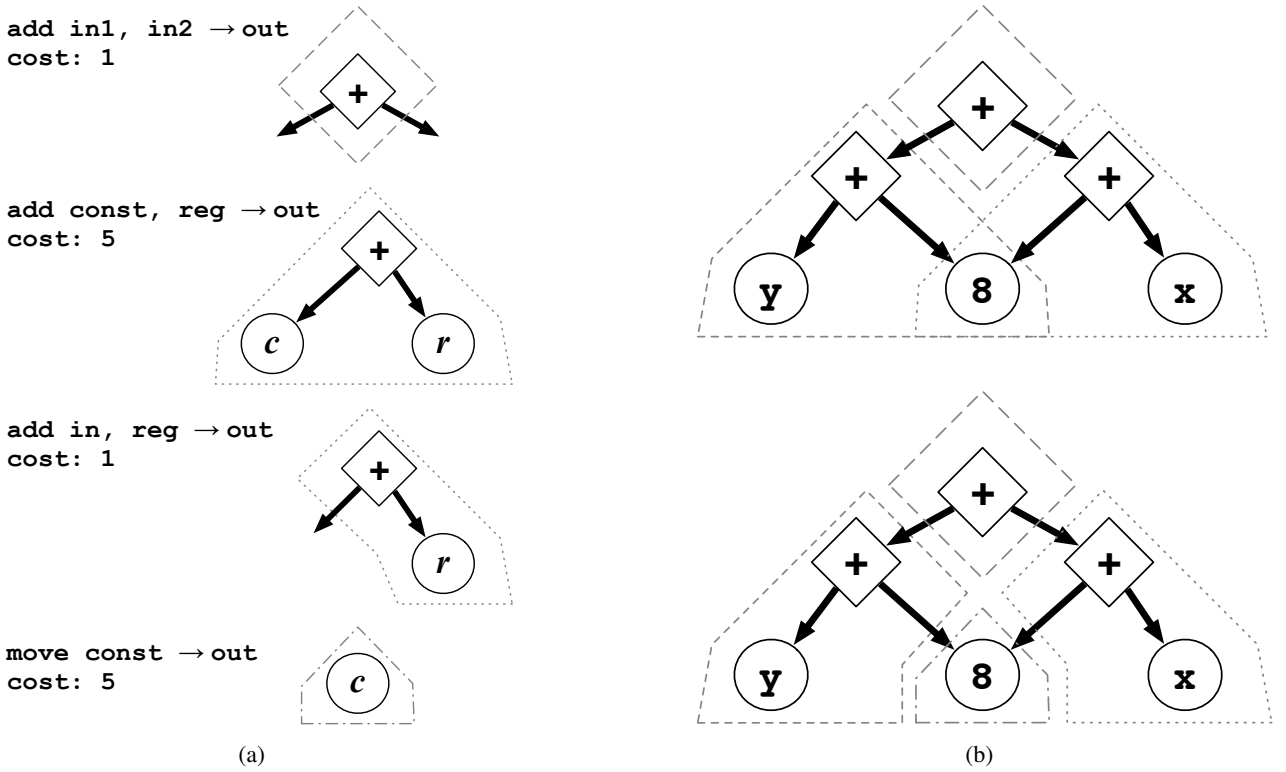

Figure 1. An example of instruction selection on a DAG. (a) The tile set used (commutative tiles are omitted). (b) Two possible tilings. In a simple cost model where every tile has a unit cost the top tiling would be optimal, but with the cost model shown the lower tiling is optimal.

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 [12, 21] 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 register[1]. Similarly, the

[1] These are unallocated temporary, note actual hard registers.

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 $o_v : ch(v) \rightarrow \{1, 2, ...|ch(v)|\}$; 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 $k(v_i) \in K \bigcup \{\circ\}$ 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 which 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 $\forall v \in V$:

$$t_i \in f(v) \implies v \; matches \; t_i$$

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

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

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

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

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

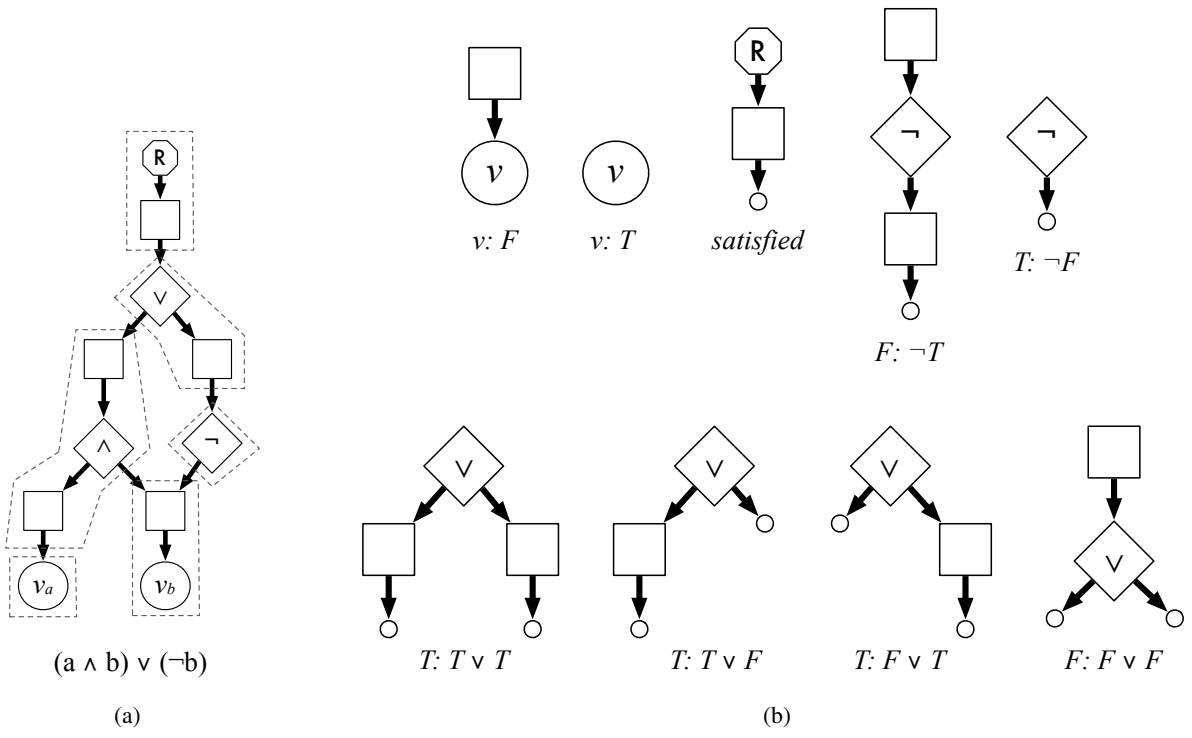

Figure 2. (a) An example of an expression DAG that represents a Boolean expression. (b) The tiles used to cover such an expression DAG. Each tile has unit cost. The tiles representing $\wedge$ are omitted, but are similar to the $\vee$ tiles with the two middle tiles having an additional box node at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

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 [19]. 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 $\{\vee, \wedge, \neg, \Box, R, v\}$. We refer to nodes with kind $\Box$ 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 $k(u_1) = \Box$ 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 $k(op_1) = \Box$ and $k(op_2)$ is the corresponding operation. Next, for every operation $aopb$ create edges $(op_2 \rightarrow a_1)$ and $(op_2 \rightarrow b_1)$ where $k(a_1) = k(b_1) = \Box$ (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 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.

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

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

```
1:  DAG : expression DAG
2:  bestChoiceForNode : Node → (Tile × int)
3:  fixedNodes : set of Node
4:  matchedTiles : set of Tile
5:  coveringTiles : Node → set of Tile

6:  procedure SELECT
7:      fixedNodes ← {}
8:      BOTTOMUPDP()                          ▷ initializes bestChoiceForNode
9:      TOPDOWNSELECT()                       ▷ initializes coveringTiles
10:     IMPROVECSEDECISIONS()                 ▷ initializes fixedNodes
11:     BOTTOMUPDP()                          ▷ uses fixedNodes
12:     TOPDOWNSELECT()                       ▷ leaves final tiling in matchedTiles

13: procedure BOTTOMUPDP
14:     for n ∈ reverseTopologicalSort(DAG) do
15:         bestChoiceForNode[n].cost ← ∞
16:         for t_n ∈ matchingTiles(n) do
17:             if ¬hasInteriorFixedNode(t_n, fixedNodes) then
18:                 val ← cost(t) + Σ_{n'∈edgeNodes(t_n)} bestChoiceForNode[n'].cost
19:                 if val < bestChoiceForNode[n].cost then
20:                     bestChoiceForNode[n].cost ← val
21:                     bestChoiceForNode[n].tile ← t_n

22: procedure TOPDOWNSELECT
23:     matchedTiles.clear()
24:     coveringTiles.clear()
25:     q.push(roots(DAG))
26:     while ¬q.empty() do
27:         n ← q.pop()
28:         bestTile ← bestChoiceForNode[n].tile
29:         matchedTiles.add(bestTile)
30:         for every node n_t covered by bestTile do
31:             coveringTiles[n_t].add(bestTile)
32:         for n' ∈ edgeNodes(bestTile) do
33:             q.push(n')
```

shared nodes using the procedure BOTTOMUPDP shown in 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

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 ∈ sharedNodes(DAG) do
3:         if coveringTiles[n].size() > 1 then                    ▷ has overlap
4:             overlapCost ← getOverlapCost(n, coveringTiles)
5:             cseCost ← bestChoiceForNode[n].cost
6:             for t_n ∈ coveringTiles[n] do
7:                 cseCost ← cseCost + getTileCutCost(t_n, 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 ← 0
3:     seen ← {}
4:     for t ∈ coveringTiles[n] do
5:         q.push(t)
6:         seen.add(t)
7:     while ¬q.empty() do
8:         t ← q.pop()
9:         cost ← cost + cost(tile)
10:        for n′ ∈ edgeNodes(t) do
11:            if n′ is reachable from n then
12:                t′ ← bestChoiceForNode[t′].tile
13:                if coveringTiles[n′].size() = 1 then
14:                    cost ← cost + bestChoiceForNode[n′].cost
15:                else if t′ ∉ seen then
16:                    seen.add(t′)
17:                    q.push(t′)
18:    return cost
```

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.

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 ← ∞
3:      r ← root(tile)
4:      for t' ∈ matchingTiles(r) do
5:          if n ∈ edgeNodes(t') then
6:              cost ← cost(t')
7:              for n' ∈ edgeNodes(t') ∧ n' ≠ n do
8:                  cost ← cost + bestChoiceForNode[n'].cost
9:              if cost < bestCost then
10:                 bestCost ← cost
11:     for n' ∈ edgeNodes(t) do                ▷ Subtract edge costs of original tile
12:         if path r ⇝ n' ∈ t does not contain n then
13:             bestCost ← bestCost − bestChoiceForNode[n'].cost
14:     return bestCost
```

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 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.

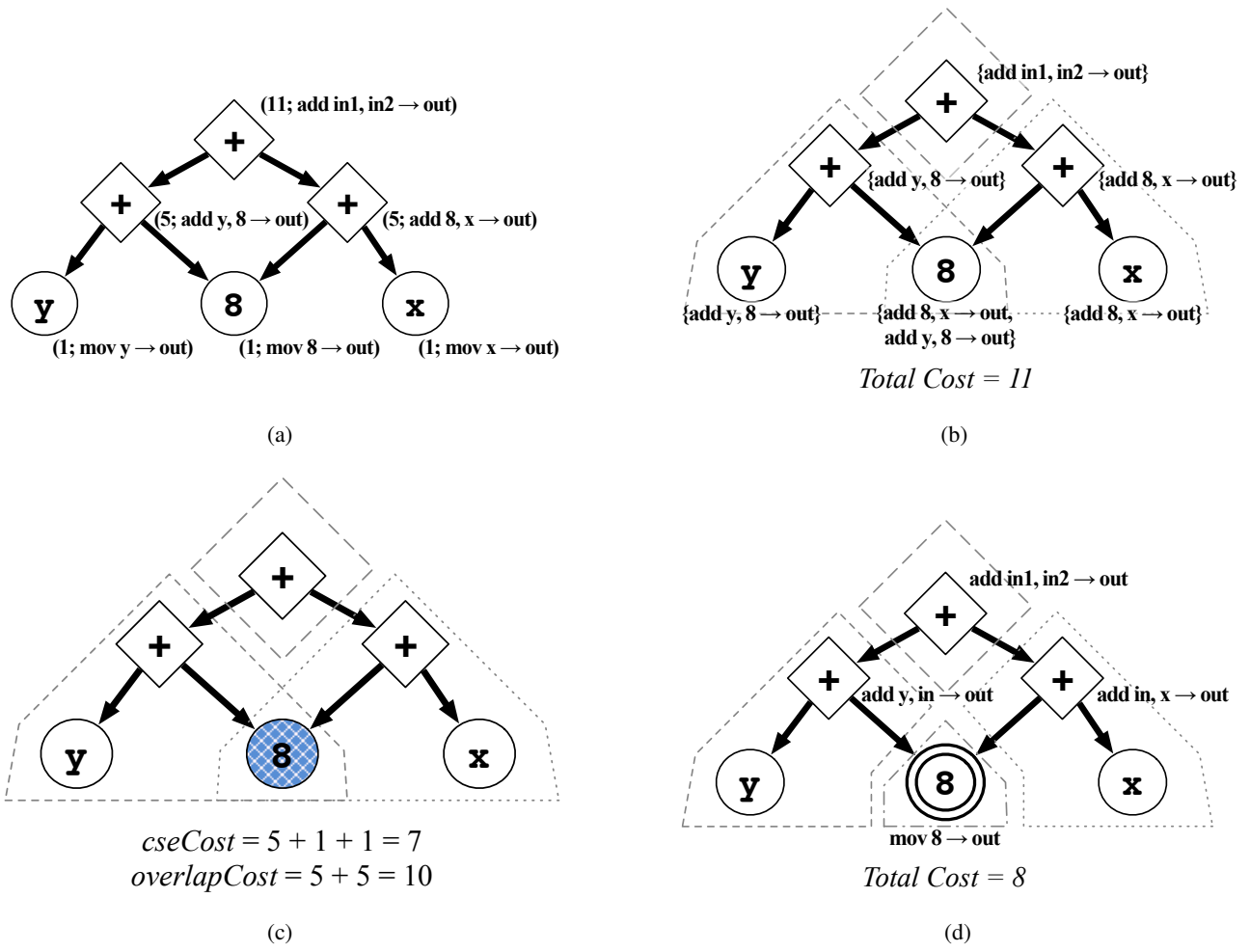

Figure 3. The application of the NOLTIS algorithm to the example from Figure 1. (a) BOTTOMPUPDP computes the dynamic programming solution for the DAG, initializing $bestChoiceForNode$. (b) TOPDOWNSELECT determines a tiling from the dynamic programming solution, initializing $coveringTiles$. (c) IMPROVECSEDECISIONS 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.

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 $edgeNodes(i, j)$ be the nodes at the edge of tile j when rooted at node i, then the optimal instruction tiling problem is:

$$\min \sum_{i,j} cost_j M_{i,j}$$

subject to

$$\forall_{i \in roots} \sum_{j} M_{i,j} >= 1 \tag{1}$$

$$\forall_{i,j} \forall_{i' \in edgeNodes(i,j)} M_{i,j} - \sum_{j'} M_{i',j'} \leq 0 \tag{2}$$

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 2.0 [29] compiler infrastructure targeting the Intel x86 architecture on the Apple Mac OS X 10.4 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].

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 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).

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 detail[2], 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 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[3] of the SPEC CPU2006 [37] benchmark suite and observing both the immediate, post-selection cost of the tiling and the final code size of the benchmark. This results in a thorough evaluation as these benchmarks include nearly half a million basic blocks ranging in size from a single instruction to thousands of instructions. We evaluate the optimality of the NOLTIS algorithm, demonstrate its superiority compared to existing heuristics, and investigate its impact on the code size of fully compiled code.

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 [22]. In order to solve the nearly half million tiling problems, we utilized a cluster of Pentium 4 machines ranging in speed from 2.8Ghz to 3.0Ghz. 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

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

[3] Three benchmarks, 400.perlbench, 453.povray, and 471.omnetpp do not execute properly due to issues unrelated to the instruction selector. If a newer release of LLVM does not fix these issues prior to publication, these benchmarks will be removed from the final paper.

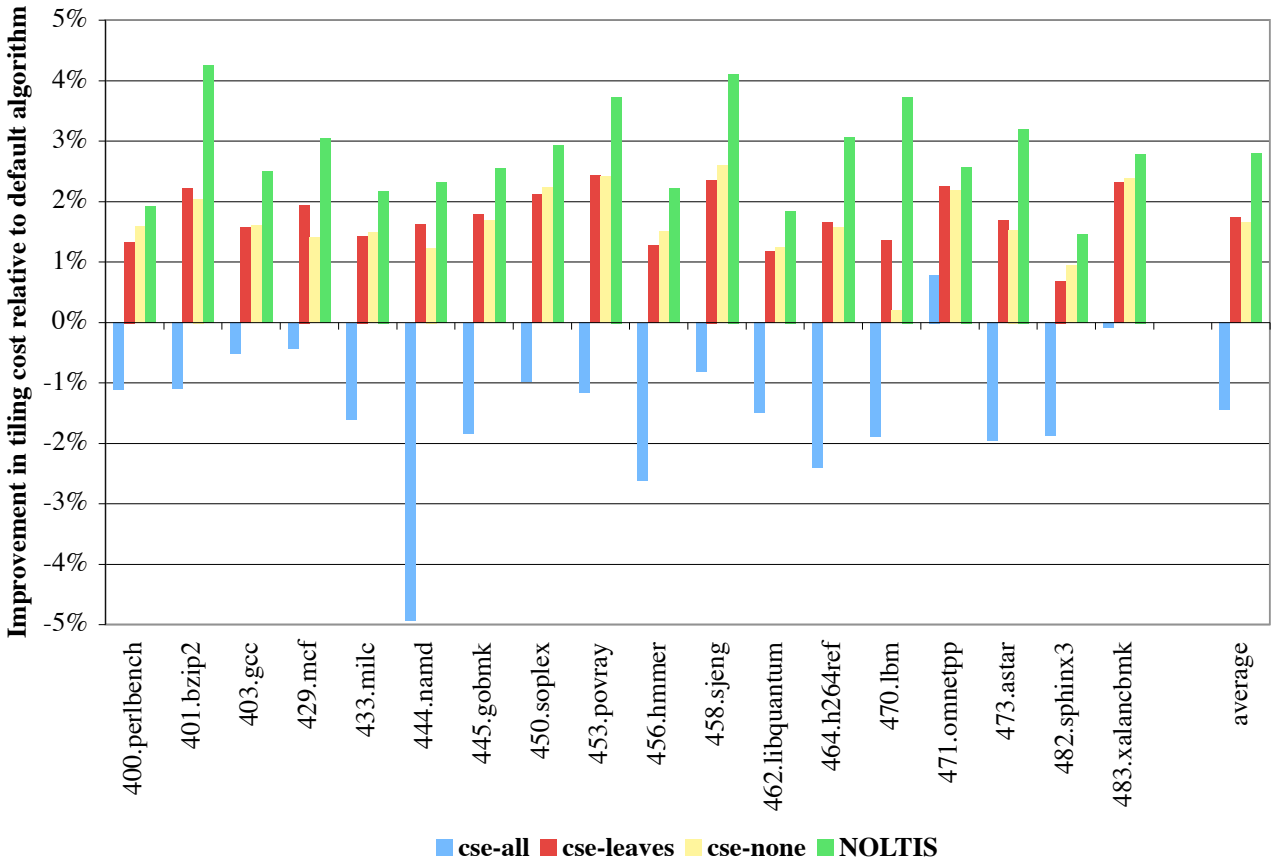

Figure 4. The improvement in tiling cost relative to the default maximal munch algorithm. The average improvements of the *cse-all*, *cse-leaves*, *cse-none*, and NOLTIS algorithms are -1.44%, 1.73%, 1.66%, and 2.80% respectively.

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 as shown in Figure 4. 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. Merely allowing overlap of leaf expressions results in a significant average improvement of 1.73% over the greedy heuristic. Interestingly,

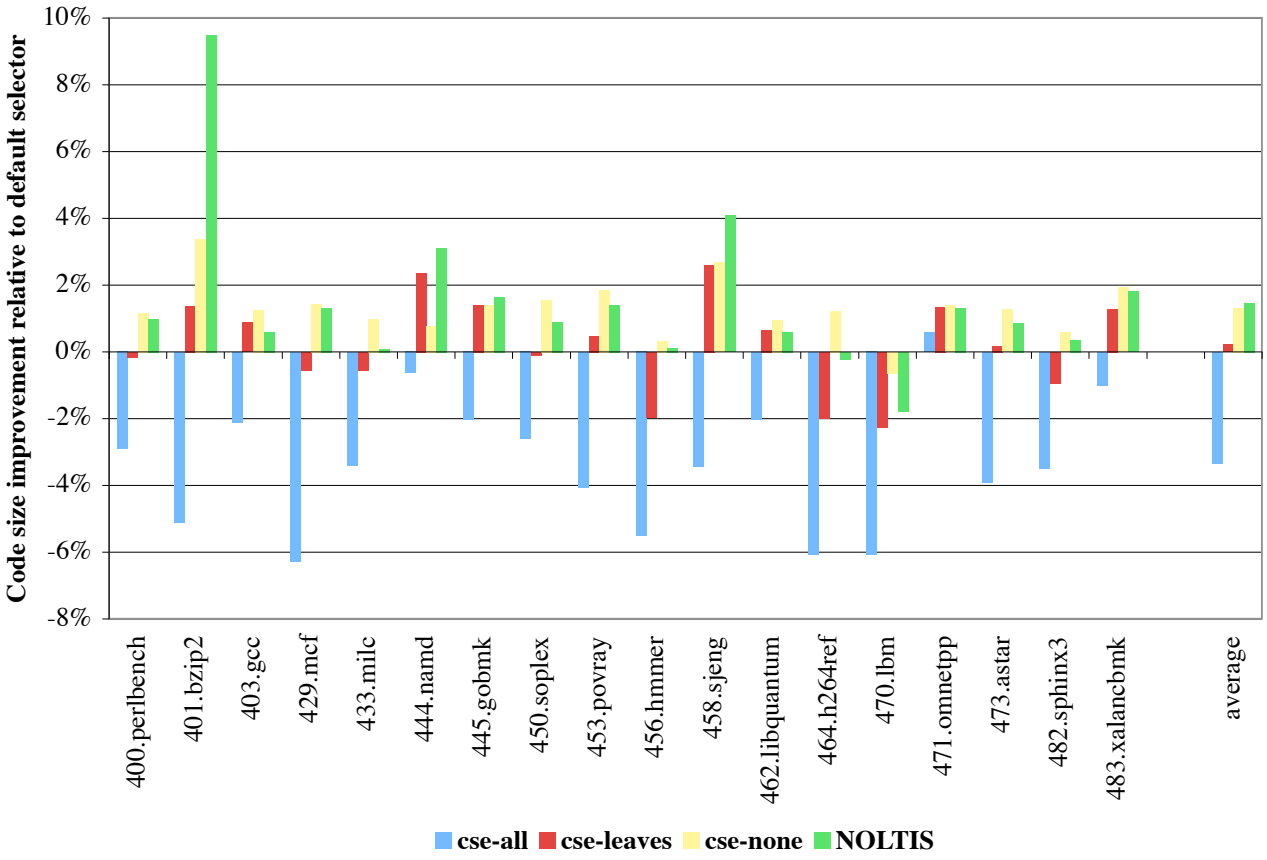

Figure 5. The final code size improvement exhibited by each algorithm relative to the default. The average improvements of the *cse-all*, *cse-leaves*, *cse-none*, and NOLTIS algorithms are -3.33%, 0.23%, 1.31%, and 1.48% respectively.

the *cse-none* algorithm also performs well even though this algorithm does not attempt to limit the amount of overlap. Unsurprisingly, the NOLTIS algorithm performs the best in every benchmark with a 2.80% average improvement.

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 resulting in the mixed code size results shown in Figure 5. Although the NOLTIS algorithm results in slightly better code size on average, the *cse-none* algorithm results in smaller code on more benchmarks. These discrepancies appear 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 hence more values must be spilled to memory. However, allowing unlimited overlap can also have a negative effect on register allocation

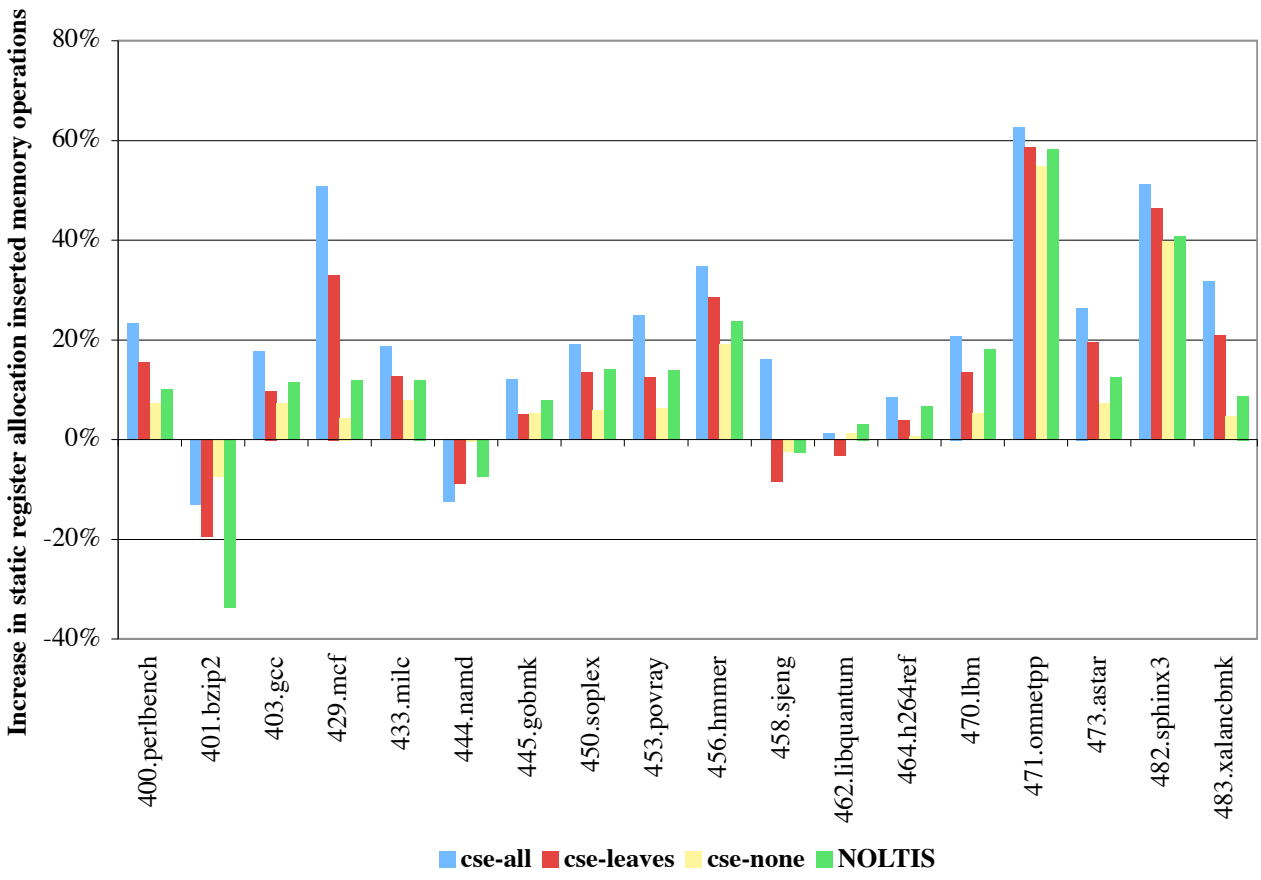

Figure 6. The impact of each instruction tiling algorithm on register allocation relative to the default algorithm. Larger values mean that the register allocator had to insert more loads and stores.

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.

The overall impact on register allocation of all the instruction tiling algorithms for each benchmark is shown in Figure 6. In most cases, all the tiling algorithms have an adverse effect upon register allocation relative to the default algorithm. This is likely because the greedy maximal munch algorithm selects the largest possible tiles resulting in the fewest number of temporaries. Note that the three benchmarks where NOLTIS has a beneficial effect on register allocation (401.bzip, 444.namd, and 458.sjeng) are the three benchmarks with the best final code size improvement. As expected, the *cse-all* algorithm does particularly poorly with an average improvement of -3.33%. The *cse-none* algorithm tends to have the least adverse effect on register allocation, likely because of its relatively larger tiles (tiles are never cut at shared nodes). This favorable interaction with register allocation appears to be largely responsible for the mixed results of Figure 5. However, there

is no guarantee that the *cse-none* algorithm will have the most favorable interaction with the register allocator. For example, in the 401.bzip2 benchmark, the NOLTIS algorithm does substantially better than the *cse-none* algorithm, in large part because the register allocator generates two-thirds as many memory operations for the NOLTIS tiling as for the *cse-none* tiling. The improved register allocation behavior combined with an already superior tiling results in an impressive 9.49% code size improvement.

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.

8. Limitations and Future Work

Instruction selection algorithms have been used successfully to solve the technology mapping problem in the automated circuit design 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 [2, 16, 32, 35] 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 allocation 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 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. We have also provided an NP-completeness proof of the DAG tiling problem and described a 0-1 integer programming formulation of the problem. Although the interaction between instruction selection and register allocation bears further study, we have shown that NOLTIS significantly improves code size compared to existing techniques.

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: Princiles, 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 *ACM '86: Proceedings of 1986 ACM Fall joint computer conference* (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] DAVIDSON, J. W., AND FRASER, C. W. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst. 6*, 4 (1984), 505–526.

[12] DE MICHELI, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994, ch. 10.

[13] EMMELMANN, H., SCHRÖER, F.-W., AND LANDWEHR, L. Beg: a generation for efficient back ends.

In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation* (New York, NY, USA, 1989), ACM Press, pp. 227–237.

[14] ERTL, M. A. Optimal code selection in dags. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM Press, pp. 242–249.

[15] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst. 1*, 3 (1992), 213–226.

[16] 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.

[17] FRASER, C. W., AND WENDT, A. L. Integrating code generation and optimization. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction* (New York, NY, USA, 1986), ACM Press, pp. 242–248.

[18] FRASER, C. W., AND WENDT, A. L. Automatic generation of fast optimizing code generators. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM Press, pp. 79–84.

[19] 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.

[20] GLANVILLE, R. S., AND GRAHAM, S. L. A new method for compiler code generation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1978), ACM Press, pp. 231–254.

[21] HASSOUN, S., AND SASAO, T., Eds. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[22] ILOG CPLEX. `http://www.ilog.com/products/cplex`.

[23] KESSLER, C., AND BEDNARSKI, A. A dynamic programming approach to optimal integrated code generation. In *LCTES '01: Proceedings 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 *SIGPLAN '84: Proceedings of the 1984 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 *DAC '87: Proceedings of the 24th ACM/IEEE conference 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 *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference 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 *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (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 *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (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] SPEC CPU2006 benchmark suite. `http://www.spec.org`.