Complete and Practical Universal Instruction Selection

GABRIEL HJORT BLINDELL, KTH Royal Institute of Technology and RISE SICS
MATS CARLSSON, RISE SICS
ROBERTO CASTAÑEDA LOZANO, RISE SICS and KTH Royal Institute of Technology
CHRISTIAN SCHULTE, KTH Royal Institute of Technology and RISE SICS

In code generation, instruction selection chooses processor instructions to implement a program under compilation where code quality crucially depends on the choice of instructions. Using methods from combinatorial optimization, this paper proposes an expressive model that integrates global instruction selection with global code motion. The model introduces (1) handling of memory computations and function calls, (2) a method for inserting additional jump instructions where necessary, (3) a dependency-based technique to ensure correct combinations of instructions, (4) value reuse to improve code quality, and (5) an objective function that reduces compilation time and increases scalability by exploiting bounding techniques. The approach is demonstrated to be complete and practical, competitive with LLVM, and potentially optimal (w.r.t. the model) for medium-sized functions. The results show that combinatorial optimization for instruction selection is well-suited to exploit the potential of modern processors in embedded systems.

CCS Concepts: • **Software and its engineering → Compilers**; **Constraint and logic languages**; • **Hardware → Emerging languages and compilers**;

Additional Key Words and Phrases: instruction selection; code generation; constraint programming; combinatorial optimization

ACM Reference format:
Gabriel Hjort Blindell, Mats Carlsson, Roberto Castañeda Lozano, and Christian Schulte. 2017. Complete and Practical Universal Instruction Selection. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (July 2017), 18 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Instruction selection is the task of implementing a program under compilation by selecting instructions from a given processor. It is a crucial part of code generation in a compiler and has been actively researched for the past four decades (see [28] for a recent survey). The task is typically decomposed into identifying the applicable instructions and selecting a combination that meets the semantics of the program under compilation. Instruction selection is an optimization problem as combinations differ in their efficiency – in certain cases, by up to two orders of magnitude [52].

Most state-of-the-art approaches apply graph-based methods to data-flow graphs only and, to avoid NP-hard methods, are restricted to trees or DAGs. These restrictions are severe: ad-hoc routines are needed for handling control flow; the scope of instruction selection is typically limited

This work is supported by the Swedish Research Council (VR 621-2011-6229) and LM Ericsson AB. The article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2017 and appears as part of the ESWEEK-TECS special issue.

DOI: 10.1145/nnnnnnn.nnnnnnn

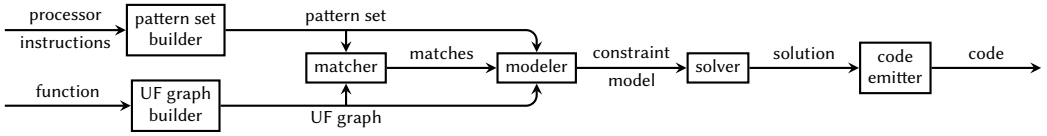


Fig. 1. Overview of the approach.

to single basic blocks, hence forsaking crucial optimization opportunities by design; and many instructions of modern processors, such as DSPs (digital signal processors), are not supported [28]. Moreover, other code generation problems cannot readily be integrated with instruction selection.

Approach. In this paper, we use a new approach called *universal instruction selection* [29] that addresses many of the limitations mentioned above. The approach (see Fig. 1 for an overview) is based on constraint programming, where problems of combinatorial optimization are expressed as constraint models. Our constraint model captures many aspects of instruction selection. It accurately reflects the interaction between control and data flow of both programs and processor instructions, enabling uniform treatment of data and control flow and selecting instructions globally for an entire function. These features jointly enable support for complex instructions, such as SIMD (single input, multiple data) and hardware-loop instructions, which often appear in embedded systems. This is especially important for processors where instructions can be tailored for a specific program and therefore can be arbitrarily complex. The model also integrates global code motion which allows computations to be moved across block boundaries, thereby advancing use of complex instructions. Moreover, as the model can potentially be solved to optimality, our approach provides a complement to the greedy heuristics applied in state-of-the-art compilers, which are suitable for day-to-day development (where short compilation times are imperative) but suboptimal for deployment (where longer compilation times can be tolerated in favor of higher code quality).

The constraint model is based on *universal function graphs*, a graph-oriented representation that also enables global code motion. The representation was recently introduced in [29], together with a basic constraint model for universal instruction selection. The basic model has several limitations: it fails to handle memory computations and function calls; it is unable to produce code in cases where additional jump instructions must be inserted; it can potentially produce incorrect code in certain cases where complex memory instructions are available; it may select more instructions than necessary, which limits code quality; it lacks scalability; and it has been evaluated on a limited set of small benchmark programs on a hypothetical processor architecture.

Contributions. We extend the approach in [29] to overcome the limitations above and make several crucial improvements to the constraint model, thus making universal instruction selection *complete* and *practical*. More precisely, we make the following contributions:

- We describe a complete instruction selector that supports memory computations and function calls, and can insert additional jump instructions where necessary.
- We achieve correctness by detecting and prohibiting illegal instruction combinations that could potentially result in cyclic data dependencies.
- We increase code quality by introducing *alternative values* to enable value reuse, thereby avoiding redundant recomputation.
- We decrease compilation time and increase scalability by refining the objective function and applying bounding techniques, where we exploit LLVM to provide a first solution.
- We evaluate the approach using medium-sized benchmark programs and real hardware (Hexagon V5 [48]), which are typical for embedded systems.

These contributions are significant as they turn a basic model for universal instruction selection into a *complete* and *practical* model.

Plan of the paper. The required *background* is introduced in Sect. 2. The following sections discuss *memory computations and function calls* (Sect. 3), *jump insertion* (Sect. 4), *cyclic data dependencies* (Sect. 5), and *value reuse* (Sect. 6), which are crucial for the *constraint model* introduced in Sect. 7. The approach is *evaluated* in Sect. 8. *Related work* is discussed in Sect. 9 and *conclusions and future work* in Sect. 10.

2 BACKGROUND

Graph-based Instruction Selection. The prevailing approach to instruction selection is to apply graph-based methods. As is common, the unit of compilation is a single function consisting of a set of basic blocks (henceforth referred to simply as *blocks*), where exactly one block represents the function's *entry block*. From the function one or more data-flow graphs are constructed, which are directed graphs where *computation nodes* represent computations in the function and edges represent data dependencies between computations. Data-flow graphs confined to a single block are called *block graphs*, and data-flow graphs that capture the data flow of entire functions are called *function graphs*. A *local* instruction selector operates on block graphs whereas a *global* instruction selector operates on function graphs.

Likewise, a data-flow graph called *pattern graph* (or simply *pattern*) is constructed for each instruction of a given processor. The set of all patterns for a processor constitutes a *pattern set*. The problem of identifying all applicable instructions for a given function is thus reduced to finding all instances where a pattern in the pattern set is subgraph isomorphic to the block or function graph. Such an instance is called a *match*.

A set of matches *covers* a block or function graph g if each computation node in g appears in exactly one match. By assigning a cost to each match that corresponds to the cost of selecting the instruction, the problem of finding the best combination of instructions is reduced to finding a cover with least cost. It is well known that the subgraph-isomorphism and optimal graph-covering problems are NP-complete in general, but for block graphs both problems can be solved optimally in linear time if the block and pattern graphs are limited to trees [28].

Constraint Programming. Constraint programming [49] is a combinatorial optimization technique for solving computationally hard problems. Instead of solving a problem directly using problem-specific heuristics – which typically yield suboptimal solutions – in constraint programming one first *models* the problem, and then *solves* the model.

A *constraint model* consists of a set of *variables* with finite domain (typically integers), a set of *constraints* expressing relations among the variables, and an *objective function*. A *solution* is a variable assignment that satisfies all constraints, and a solution is *optimal* if it maximizes (or minimizes) the value of the objective function. A *model instance* is a constraint model that has been instantiated with the parameters for a specific instance of the problem.

A *constraint solver* (or just *solver*) finds solutions to a given model instance by performing *propagation* and *search*. Propagation removes values for variables which are in conflict with a constraint and thus cannot appear in any solution. If any variable remains unassigned and no further propagation is possible, search attempts several value choices on which propagation and search is interleaved. Propagation is essential for reducing the search space.

Many problems have recurring structural patterns that constraint programming can exploit through *global constraints*. A global constraint expresses problem-specific relations among several

variables, offering two benefits: it eases modeling, and it enables strong propagation which drastically reduces the search space.

Static Single Assignment Form. An intermediate representation (IR) based on static single assignment (SSA) is a representation where each program variable is defined exactly once [17]. For example, the program shown in Fig. 2a is not in SSA form as the variables n and f are redefined within the loop. To disambiguate cases where the definition depends on control flow, SSA uses so-called *φ-functions* which take a value and the block from where the value originates as arguments. By splitting n and f into several variables and merging their values using φ-functions, we get the program shown in Fig. 2b, which is in SSA form.

Given a function represented in SSA form, a data-flow graph called *SSA graph* can be constructed [26]. By including *φ-nodes*, the SSA graph can capture data flow of entire functions, which is an enabler for global instruction selection. Matches derived from patterns that only cover φ-nodes are called *φ-matches*.

Universal Function Graphs. We use *universal function graphs* (or *UF graphs* for short) to refer to the program representation introduced in [29] (see Fig. 2c for an example). A UF graph is essentially a combination of two graphs: a control-flow graph, consisting of *block nodes* and control-flow edges, which is extended with *control nodes* to represent jumps; and an SSA graph, which is extended with *value nodes* to represent values. An important invariant is that every control node and value node has exactly one inbound control-flow edge and data-flow edge, respectively. Consequently, value nodes not produced through computation – such as constants and function arguments – have a data-flow edge from the node representing the function's entry block.

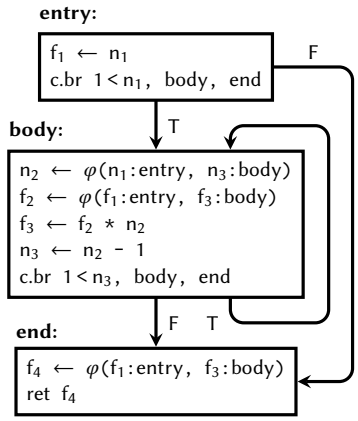
As with SSA graphs, the UF graph does not dictate in which block a given computation should be executed, which is crucial for enabling global code motion. Having no restraints at all, however, may lead to code that violates the program semantics. Consequently, the UF graph contains *definition edges* to indicate that a given value must be produced within a specific block. The concerned values are exactly those connected to φ-nodes, which is sufficient to preserve the program semantics.

Since the UF graph contains several kinds of nodes, a refined notion of coverage is given in [29]: a set of matches *covers* a UF graph if each *operation* appears in exactly one match, where an operation refers to either a computation node or a control node in the UF graph. We will also use the term *blocks* to refer to block nodes in the UF graph.

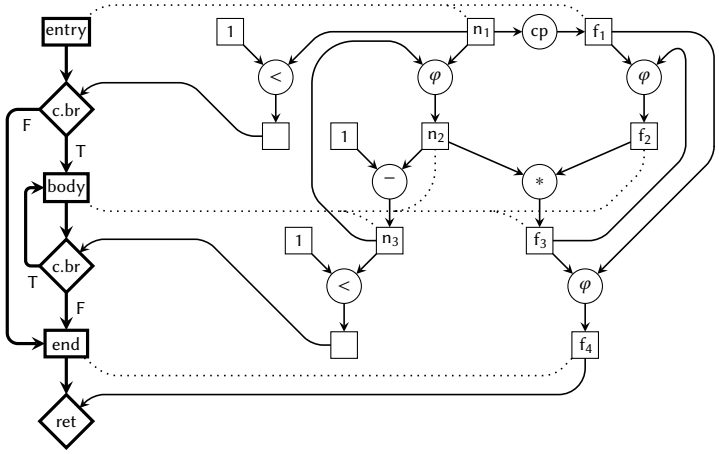
Global Code Motion. Global code motion is the task of moving computations from one block to another in order to increase code quality. We formalize the global code motion problem by reiterating the definitions given in [29]. If a *datum d* refers to a value node in the UF graph, then a match *m defines* respectively *uses d* if there exists an inbound respectively outbound data-flow edge from *d* in the pattern of *m*. Hence a datum can be both defined and used by the same match. Next, a block *b dominates* another block *c* if every control-flow path in the UF graph that goes from the function's entry block to *c* also goes through *b*. By definition, a block always dominates itself. Hence a placement of selected matches into blocks is a solution to the global code motion problem if each datum *d* is defined in some block *b* such that *b* dominates every block wherein *d* is used, excluding uses made by φ-matches. The last clause is needed as blocks appearing as arguments to φ-functions do not necessarily dominate the block wherein the φ-function is placed. For example, in Fig. 2b the φ-function in the end block demands that value f_3 is defined in the body block, which clearly does not dominate end. This is the reason for applying definition edges to enforce that f_1 and f_3 are defined in the correct blocks. Likewise, a definition edge is also applied for f_4 to enforce correct placement of the φ-function.

```
int fact(int n) {
    int f = n;
    for (; 1 < n; n--)
        f = f * n;
    return f;
}
```

(a) C code.



(b) Corresponding SSA IR and control-flow graph.



(c) Corresponding UF graph. Thick-lined diamonds, boxes, and arrows represent control nodes, block nodes, and control-flow edges, respectively. Thin-lined circles, boxes, and arrows represent computation nodes, value nodes, and data-flow edges, respectively. Dotted lines represent definition edges. For readability, the value node with value 1 has been duplicated and the data-flow edges from the entry block to the value nodes representing constants and function arguments have been removed.

Fig. 2. The factorial function written in C, transformed into an SSA-based IR, and then into a UF graph.

Some complex instructions impact both instruction selection and global code motion. Assume for example a match m of the pattern shown in Fig. 3, which models an instruction that assigns a register one of two values depending on whether a given predicate is true or false. Assume further that the entry, if, and end blocks in the pattern are mapped to blocks in a UF graph labeled A, B, and C, respectively. A match *spans* the blocks in the UF graph mapped to blocks in the pattern. Hence m spans A, B, and C. Of these, m *consumes* B as the instruction of m assumes full control of the control flow to and from B. Consequently, no other computations may be placed in blocks consumed by some selected match.

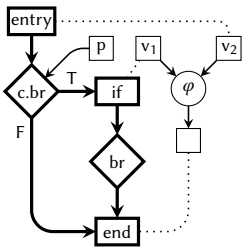


Fig. 3. A conditional assignment instruction

As demonstrated in Sect. 8, solving global instruction selection and global code motion in unison enables use of complex instructions that would otherwise not have been selectable.

3 MEMORY COMPUTATIONS AND FUNCTION CALLS

As with arithmetic computations, we represent memory computations and function calls as computation nodes (see Fig. 4 for an example). Nodes representing memory loads and stores have a fixed number of inbound and outbound data-flow edges, while nodes representing function calls may have an arbitrary number of data-flow edges (depending on the function being called). This also enables the calling convention of the target machine – that is, how to pass the arguments and return value – to be implemented through patterns as these can apply location requirements on the data being used or defined (see Eq. 11 in Sect. 7).

Memory computations and function calls may have side effects which are not observable from the data and control flow. As the order in which these are executed may affect the program semantics, the order given in the function must be preserved. We introduce *state nodes* to capture this order, and refine the definition of *data* to also include state nodes. As shown in Fig. 4, each memory computation and function call takes exactly one state node as input and produces another state node as output, thus forming state dependencies which exactly capture the necessary order between such operations. Each state flow is limited to a single block, where the first and last state node in the chain has a data-flow edge and a definition edge, respectively, from the block node. This is to forbid memory computations and function calls from being moved to another block as well as to comply with the invariant that every datum has exactly one inbound data-flow edge.

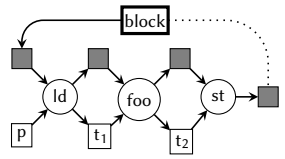
4 JUMP INSERTION

For certain combinations of functions and target machines, the produced code may need to include more jumps than originally specified in the function. In most architectures, a branch instruction jumps to a given label if the condition holds, otherwise the execution continues to the next instruction (known as *fall-through*). Hence the block representing the false label must follow the branch instruction in the code. However, in Fig. 2b we see that the end block is the false label of two conditional jumps, residing in blocks entry and body. Since end cannot follow both entry and body, an unconditional jump to end must be inserted after one of the conditional jumps.[1] We address this by extending the pattern set with new patterns, called *dual-target branch patterns*, that insert additional jump instructions if selected.

[1]In this particular case, the problem could also be solved by negating the condition and swapping the false and true labels in one of the conditional jumps. However, there exist cases where this is not sufficient (for example, consider two loops that share the same exit block).

```
block:
  t1 ← ld p
  t2 ← call foo, t1
  st q ← t2
```

(a) IR.

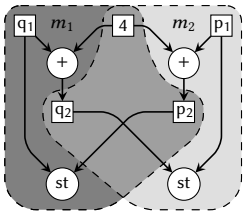


(b) UF graph. Gray boxes represent state nodes.

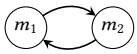
Fig. 4. An example illustrating how to capture memory computations and function calls in the UF graph.

```
p2 ← p1 + 4
st q1 ← p2
q2 ← q1 + 4
st p1 ← q2
```

(a) IR.



(b) UF graph, covered using two matches derived from an auto-increment store instruction.



(c) Dependency graph.

Fig. 5. A program at risk of cyclic data dependency. For brevity, state nodes are not included in the UF graph.

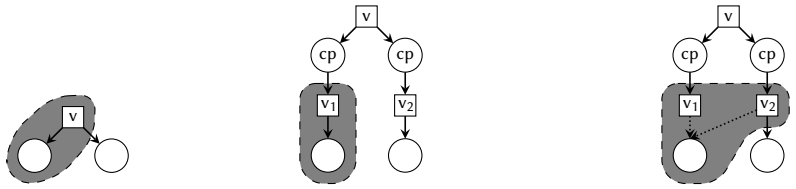
The idea works as follows. Given a pattern set S, find the pattern $g_{\text{br}} \in S$ that corresponds to an unconditional jump (it is reasonable to assume such a pattern exists). Then, for each pattern $g \in S$ that corresponds to a conditional jump with a fall-through condition to block b, add to S a new pattern h that has the same graph structure as g and emits the instruction represented by g followed by the instruction represented by g_{br}. Because h has no fall-through condition, h represents a conditional jump targeting two blocks (hence its name). The new pattern set containing all dual-target branch patterns is denoted S_{ext}.

This idea of extending the pattern set can also be generalized to include branch patterns that flip the predicate used in conditional jumps, which is applied in techniques like trace scheduling [22] to alleviate block reordering and jump elimination.

5 CYCLIC DATA DEPENDENCIES

Modern processors often contain complex instructions that could potentially lead to cyclic data dependencies. As pointed out by Ebner et al. [18], this can happen with memory instructions that automatically increment the address register after a load or store. See for example Fig. 5. If both matches are selected, then either value p_2 or value q_2 will be used before it is available (depending on the instruction order), thus resulting in incorrect code. Consequently, such combinations must be identified and forbidden.

We detect such combinations – which could involve more than two matches – by constructing a *dependency graph* and finding all cycles (we use Johnson's algorithm [30]). A node in the dependency graph represents a match m, which has a directed edge if m produces data used by another match. For example, in the dependency graph shown in Fig. 5, m_1 has an edge to m_2 due to value q_2, and m_2 has an edge to m_1 due to value p_2. φ-matches are not included as they will always yield cycles that are not due to true cyclic data dependencies, and neither are *null matches* (matches that emit nothing if selected) as they will never be the cause of such dependencies.



(a) UF graph. (b) After copy extension. (c) With alternative values.

Fig. 6. A UF graph covered by a match with two alternative values. Dotted edges indicate potential use of such values.

6 VALUE REUSE

Code quality can be increased if instructions are allowed to reuse copies of values, which is a crucial feature to be expected in the code emitted by any modern instruction selector. See for example Fig. 6a, in which a value v is used by two computation nodes and one of them is covered by a match. After copy extension – a transformation described in [29] used to handle architectures with instructions that require data to reside in specific locations – the computations will instead use the two newly introduced values v_1 and v_2, respectively, as shown in Fig. 6b. Assume now that v resides in a location where it must first be copied elsewhere before used. It may for example reside on the memory stack, which is a common calling convention in architectures with few registers. Without value reuse, a copy would be performed for each such use. But because v_1 and v_2 are copies of the same value, one copy might be sufficient (depending on the instructions using the values). We say that v_1 and v_2 are *copy-related* and therefore interchangeable, meaning the match is free to use either of values v_1 and v_2, as shown in Fig. 6c. Consequently, if a match m uses but does not define a value v_1, and v_1 is copy-related with values $v_2, \ldots, v_k$, then $v_1, \ldots, v_k$ are *alternative values* to m.

We utilize alternative values by allowing a value node in a pattern to map to multiple (copy-related) value nodes in the UF graph. We do this by first finding all matches and then expanding each match with alternative value node mappings. Another approach is to first expand the UF graph with additional data-flow edges and then merge matches that differ only in value node mappings, but that complicates the matching algorithm and incurs an unnecessary computational overhead. We then extend the constraint model by essentially adding an extra level of indirection wherever a constraint refers to a particular datum, which is detailed below.

7 CONSTRAINT MODEL

Match Identification. For a given UF graph G and a pattern set S_{ext}, the match set M contains all instances where a pattern in S_{ext} is subgraph isomorphic to G. Like in [29], we compute M using VF2 [16] – a well-known subgraph isomorphism algorithm – and apply the same method for removing illegal matches.

Operation Coverage. The two sets of variables $\mathbf{sel}[m] \in \{0, 1\}$ and $\mathbf{omatch}[o] \in M_o$ model whether match $m \in M$ is selected respectively which selected match covers operation $o \in O$, where O denotes the set of operations in G, and $M_o \subseteq M$ denotes the set of matches that can cover o. The condition that each operation must be covered by exactly one selected match can then be modeled as

$$\begin{gathered} \mathbf{omatch}[o] = m \Leftrightarrow \mathbf{sel}[m], \\ \forall o \in O, \forall m \in M_o, \end{gathered} \tag{1}$$

where $\mathbf{sel}[m]$ abbreviates $\mathbf{sel}[m] = 1$ (likewise, for a variable $\mathbf{x} \in \{0, 1\}$, $\neg\mathbf{x}$ abbreviates $\mathbf{x} = 0$).

Data Definitions. Similarly, each datum must be defined by exactly one selected match. The set of variables $\mathbf{dmatch}[d] \in M_d$ models which selected match defines datum $d \in D$, where $M_d \subseteq M$ denotes the set of matches that can define d, and D denotes the set of data in G. The condition can then be modeled as

$$\begin{gathered}\mathbf{dmatch}[d] = m \Leftrightarrow \mathbf{sel}[m], \\ \forall d \in D, \forall m \in M_d.\end{gathered} \tag{2}$$

Global Code Motion. The two sets of variables $\mathbf{oplace}[o] \in B$ and $\mathbf{dplace}[d] \in B$ model in which block operation o respectively the definition of datum d is placed, where B denotes the set of blocks in G. In order to decide in which block to place a selected match m, all operations covered by m are required to be placed in the same block. If $\mathit{covers}(m) \subseteq O$ denotes the set of operations covered by match m, then the condition can be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o_1] = \mathbf{oplace}[o_2], \\ \forall m \in M, \forall o_1, o_2 \in \mathit{covers}(m).\end{gathered} \tag{3}$$

Jump operations are forbidden to be placed in another block than originally indicated in the UF graph as that would most likely break the program semantics. Let $\mathit{entry}(m) \in B$ denote the entry block of match m if the pattern of m has such a node, otherwise $\mathit{entry}(m) = \varnothing$. The condition can then be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] = b, \\ \forall m \in M, \forall o \in \mathit{covers}(m), \forall b \in \mathit{entry}(m).\end{gathered} \tag{4}$$

Each datum d must be defined in some block $b \in B$ such that b dominates every block wherein d is used (with the exception of φ-matches). In addition, we want to make use of alternative values. Assume therefore that, instead of defining and using data, a match defines and uses *operands*. Every definition or usage in a match incurs a new operand, which will be mapped to a datum. Hence the set of variables $\mathbf{alt}[p] \in D_p$ models to which datum operand $p \in P$ is mapped, where $D_p \subseteq D$ denotes the alternative values available for p, and P denotes the set of operands applied in M. Let $\mathit{defines}(m) \subseteq P$ and $\mathit{uses}(m) \subseteq P$ and denote the set of operands defined respectively used by match m. Let also $\mathit{dom}(b) \subseteq B$ denote the set of blocks that dominate block b. The condition can then be modeled as

$$\begin{gathered}\mathbf{dplace}[\mathbf{alt}[p]] \in \mathit{dom}(\mathbf{oplace}[o]), \\ \forall m \in M_{\bar{\varphi}}, \forall p \in \mathit{uses}(m), \forall o \in \mathit{covers}(m),\end{gathered} \tag{5}$$

where $M_{\bar{\varphi}}$ denotes the set M excluding φ-matches.

The conditions imposed by the definition edges are modeled as

$$\begin{gathered}\mathbf{dplace}[d] = b, \\ \forall \langle d, b \rangle \in E,\end{gathered} \tag{6}$$

where E denotes the set of definition edges in G.

The **oplace** and **dplace** variables are connected as follows: if a match m spans any blocks then every datum d defined by m may be defined in any of the spanned blocks, otherwise d must be defined in the same block wherein m is placed. This is because there exist cases where the constraint model would become over-constrained if only the latter clause was applied. Assume for example a match m of the pattern shown in Fig. 3. Due to Eq. 4 the operations covered by m must be placed in the block b in G corresponding to the pattern's entry block. Due to the definition edges, however, the datum corresponding to value v_1 in the pattern must be defined in a block different from b. If $\mathit{spans}(m) \subseteq B$ denotes the set of blocks spanned by match m, then the conditions can jointly be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{dplace}[\mathbf{alt}[p]] \in \{\mathbf{oplace}[o]\} \cup \mathit{spans}(m), \\ \forall m \in M, \forall p \in \mathit{defines}(m), \forall o \in \mathit{covers}(m).\end{gathered} \tag{7}$$

Lastly, no operations must be placed inside blocks consumed by a selected match. Hence, if $\mathit{consumes}(m) \subseteq B$ denotes the set of blocks consumed by match m, then the condition can be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{oplace}[o] \neq b, \\ \forall o \in O, \forall m \in M, \forall b \in \mathit{consumes}(m).\end{gathered} \tag{8}$$

Inactive Data. Due to Eq. 5 there may be data that is not used by any match, but Eq. 2 requires that every datum must still be defined. This is addressed by extending the pattern set with a *kill pattern*, consisting of a single copy node which uses one datum and defines another datum. Matches derived from kill patterns are called *kill matches*, which are also null matches. A datum is *inactive* if it is defined by a kill match, and non-kill matches are forbidden to use inactive data.

With the set of variables $\mathbf{inactive}[d] \in \{0, 1\}$, which models whether a datum d is active, the condition can be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Leftrightarrow \mathbf{inactive}[\mathbf{alt}[p]], \\ \forall m \in M_{\times}, \forall p \in \mathit{defines}(m),\end{gathered} \tag{9}$$

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \neg\mathbf{inactive}[\mathbf{alt}[p]], \\ \forall m \in M_{\bar{\times}}, \forall p \in \mathit{uses}(m),\end{gathered} \tag{10}$$

where $M_{\times}$ and $M_{\bar{\times}}$ denote the set of kill matches respectively the set M excluding kill matches.

Data Copying. The set of variables $\mathbf{loc}[d] \subseteq L \cup \{l_{\text{null}}\}$ models in which location a datum d is available, where L denotes the set of possible locations, and l_{null} denotes the location of an intermediate value produced by an instruction that can only be accessed by this very instruction. For example, the address computed by a memory load instruction with a sophisticated addressing mode cannot be reused by other instructions. Thus, if $\mathit{stores}(m, p) \subset L$ denotes the set of permitted locations for an operand p defined or used by match m – where an empty set means no restrictions are imposed – then the condition can be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p]] \in \mathit{stores}(m, p), \\ \forall m \in M, \forall p \in P \text{ s.t. } \mathit{stores}(m, p) \neq \varnothing.\end{gathered} \tag{11}$$

The condition that forbids reuse of intermediate values can be modeled as

$$\begin{gathered}\mathbf{sel}[m] \Rightarrow \mathbf{loc}[\mathbf{alt}[p]] = l_{\text{null}}, \\ \forall m \in M, \forall p \in \mathit{intvalues}(m),\end{gathered} \tag{12}$$

where $\mathit{intvalues}(m) \subseteq P$ denotes the set of intermediate values produced by match m. It is assumed that copy extension (see [29]) has been performed on the UF graph.

Fall-through Branching. The set of variables $\mathbf{succ}[b] \subseteq B$ models the successor of block b, and a valid block order corresponds to a cycle formed by the **succ** variables. Using the global *circuit* constraint [37], the condition can be modeled as

$$\mathit{circuit}(\cup_{b \in B}\{\mathbf{succ}[b]\}). \tag{13}$$

If a match m is derived from an instruction that performs a fall-through to block b, then the condition can naively be modeled as $\mathbf{sel}[m] \Rightarrow \mathbf{succ}[\mathit{entry}(m)] = b$. However, this constraint is too limiting as it is not uncommon that an empty block is placed between $\mathit{entry}(m)$ and b. Hence we extend the constraint to allow fall-through via an empty block. Let J denote a set of tuples, consisting of a match and a block to which the match falls through, and let $M_{\perp}$ denote the set of null matches. The constraint can then be modeled as

$$\begin{gathered}\big(\mathbf{succ}[\mathbf{succ}[\mathit{entry}(m)]] = b \wedge \mathit{empty}(\mathbf{succ}[\mathit{entry}(m)])\big) \vee \mathbf{succ}[\mathit{entry}(m)] = b, \\ \forall \langle m, b \rangle \in J,\end{gathered} \tag{14}$$

where $\mathit{empty}(b) = \mathbf{oplace}[o] \neq b \vee \mathbf{omatch}[o] \in M_{\perp}, \forall o \in O$.

Cyclic Data Dependencies. As explained in Sect. 5, combinations of matches that will result in cyclic data dependencies must be forbidden. If $F \subset 2^M$ denotes the set of such combinations, then the condition can be modeled as

$$\sum_{m \in f} \mathbf{sel}[m] < |f|, \forall f \in F. \tag{15}$$

Objective Function. The objective function reflects the desired characteristics of the produced code. For example, if the compiler should maximize performance then the execution time of each block should be minimized. As is common, the execution time should also be weighted with the relative execution frequency of each block, which is typically estimated through static analysis. A straightforward objective function can be modeled as

$$\sum_{m \in M} \mathbf{sel}[m] \times \mathit{cost}(m) \times \mathit{freq}(\mathit{place}(m)), \tag{16}$$

where $\mathit{cost}(m) \in \mathbb{N}$ denotes the cost of match m (typically the instruction latency), $\mathit{freq}(b) \in \mathbb{N}$ denotes the execution frequency of block b, and $\mathit{place}(m) \in B$ denotes the block wherein m is placed.

However, the objective function in Eq. 16 is naive as it yields poor propagation, resulting in long compilation times. A better approach is to reason about the cost incurred by each operation (which must always be covered) instead of each match (which may or may not be selected).

The set of variables $\mathbf{ocost}[o] \in \mathbb{N}$ thus models the cost incurred by operation o. The cost per operation is computed as follows. For each match m, evenly divide the cost of m among the operations covered by m. Let $\mathit{cost}(m, o) \in \mathbb{N}$ denote this cost. Then, for each operation o covered by m and block b wherein o could be placed, multiply $\mathit{cost}(m, o)$ by the execution frequency of b. Hence $\sum_{o \in \mathit{covers}(m)} \mathit{freq}(b) \times \mathit{cost}(m, o) = \mathit{freq}(b) \times \mathit{cost}(m)$. By representing this information as a matrix

$$\mathbf{C} = \left[\left\langle o, m, b, \mathit{freq}(b) \times \mathit{cost}(m, o) \right\rangle \;\middle|\; \begin{array}{c} m \in M, \\ o \in \mathit{covers}(m), \\ b \in B \end{array} \right] \tag{17}$$

one can connect the **omatch**, **oplace**, and **ocost** variables by requiring that, for each operation o, the tuple $\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o] \rangle$ must appear as a row in $\mathbf{C}$. This can be enforced using the global *table* constraint [39], thus allowing the condition to be modeled as

$$\begin{array}{c} \mathit{table}\left(\langle o, \mathbf{omatch}[o], \mathbf{oplace}[o], \mathbf{ocost}[o] \rangle, \mathbf{C}\right), \\ \forall o \in O. \end{array} \tag{18}$$

Lastly, the objective function to minimize is modeled as

$$\mathbf{cost} = \sum_{o \in O} \mathbf{ocost}[o], \tag{19}$$

where $\mathbf{cost} \in \mathbb{N}$ models the total cost of the solution.

Bounding the Cost Variable. To find the optimal solution – that is, the solution which minimizes or maximizes the objective function – most solvers apply a method known as *branch and bound* [34]. The idea is to first find some initial solution with cost c and then enforce that all later found solutions must have a cost strictly less than c, thereby avoiding the parts in the search space that only contain inferior solutions. This is repeated until no more solutions are found, where the last found solution is optimal.

The solving time can be decreased by bounding the cost variable, which reduces the search space. This is particularly important for proving optimality. We obtain a lower bound by solving a

relaxation of the problem which we model using variables **omatch**, **opcosts**, **sel**, **succ**, and **totalcost** together with Eq. 1 and modified versions of Eq. 13, 14, and 17–19. The cost of an optimal solution to the relaxed model is always less than or equal to the cost of any solution to the full model. We obtain an upper bound by taking the cost of the solution found by LLVM, which only takes a tiny fraction of the time it takes to solve the model itself. If $C_{\text{relaxed}} \in \mathbb{N}$ and $C_{\text{LLVM}} \in \mathbb{N}$ denote the costs found using the relaxed model and LLVM, respectively, then the bounding conditions can jointly be modeled as

$$C_{\text{relaxed}} \leq \textbf{cost} < C_{\text{LLVM}}. \tag{20}$$

If no solution exists when using C_{LLVM} as lower bound, then it means no code better than that produced by LLVM can be produced by the constraint model.

Implied Constraints and Dominance Breaking. Simply modeling a problem as a constraint model is in most cases not sufficient for solving the problem in practice, typically due to too weak propagation and too large a search space. Consequently, a constraint model must often be extended with *implied* (or *redundant*) *constraints* (to strengthen propagation) and *dominance breaking constraints* (to remove solutions that are no better than other solutions). Our constraint model is no exception, but due to space restrictions we only briefly discuss a few of them in this paper.

Several implied constraints are based on the fact that for every operation, a covering match must be selected, and for every datum, a defining match must be selected. For example:

- Suppose that for a given non-ϕ-node operation o and data u and d, where all matches covering o use u, define d and span no blocks. Then o must be placed in the block defining d, which must be dominated by the block defining u.
- Suppose that for a given operation o and block b, b is the entry block of all matches that cover o. Then o must be placed in b.
- Suppose that for a given non-state datum d, its defining matches do not include any kill matches, and they all define d accessible by other instructions. Then the location of d cannot be l_{null}.

Dominance breaking constraints include the following:

- The location of all state data is fixed to l_{null}.
- For any match that is not selected, its **alt** variables are fixed to their minimal values.
- Any kill match whose operation can be covered by a null match, or whose datum it defines has no copy-related values, is redundant and thus fixed as not selected.
- Copy-related values cause *value symmetries* in the solution space. Let $V = \{v_1, \ldots, v_n\}$ be a set of copy-related values. Then any solution can be converted to an equivalent solution simply by picking two members v_i and v_j from V and swapping all their occurrences. We break such symmetries with *value precedence constraints* [38], which state that any occurrence of v_j in the solution must be preceded by an occurrence of v_{j-1}, $\forall 1 < j \leq n$.

Limitations. In some cases, it is beneficial to *recompute* the same value instead of reusing it. In terms of graph-based instruction selection, this means letting an operation be covered by more than one match. However, Eq. 1 and 2 enforce that every operation and datum must be covered respectively defined by exactly one selected match, and several other constraints depend on this assumption. Consequently, a solution that is optimal with respect to the constraint model may still be inferior to that produced by an instruction selector which supports recomputation.

Another limitation is that fall-through branching across more than one empty block will still result in redundant jump instructions.

Model Differences. Compared to the constraint model introduced in [29], only the **loc** and **succ** variables remain unchanged. Variables **alt**, **cost**, **dmatch**, **dplace**, **inactive**, **ocost**, **omatch**, **oplace** are new, where **dplace** and **oplace** respectively replace the **def** and **place** variables in [29].

Regarding the constraints, only Eq. 6 is identical to Eq. 5 in [29]. Equations 1, 4, 5, 7, 8, 11, and 13 are respective reformulations of Eq. 1, 3, 4, 6, 7, 8, and 9 in [29]. Equations 17, 18, and 19 are jointly a reformulation of Eq. 11 in [29]. Equations 2, 3, 9, 10, 12, 14, 15, and 20 are new constraints. Equations 2 and 10 in [29] become obsolete in this constraint model due to lack of null blocks.

8 EXPERIMENTAL EVALUATION

We implemented the constraint model in MiniZinc 2.1.3 [45] – a high-level constraint modeling language – and solved the instances using Chuffed [14] – a lazy clause learning constraint solver which is included in MiniZinc. The experiments were run using a single thread on a Linux machine with an Intel Xeon E5-2680 v3 at 2.50 GHz and 64 GB main memory. Since we are solving optimal graph covering – which is an NP-complete problem – we limit solving time with a timeout.

As input for the experiments we use functions extracted from the MediaBench suite [40] and processor instructions from Hexagon V5 [48] – a DSP with a rich instruction set that appears in many embedded systems. The functions (in LLVM IR) are generated with LLVM 3.8 [36] using the `-O3` optimization level. 284 candidate functions are filtered out of an initial pool of 6313 functions. A function is a candidate if it is medium-sized (between 50 and 200 LLVM IR instructions, yielding function graphs between 189 and 1524 nodes) and does not contain non-scalar or floating-point LLVM instructions (which are not yet supported due to limitations in our tool chain but not in the approach). Finally, 20 functions are selected by sampling out of the candidate pool. The purpose of sampling is to make the experiments feasible (so that they can be reproduced in the order of hours) while ensuring a certain diversity in the selected function graphs. The sampling procedure splits the candidate functions into 20 clusters with a k-means clustering algorithm and randomly selects a function out of each cluster [47]. Functions are clustered by size (in number of LLVM IR instructions), number of memory instructions (one of the main sources of complexity for instruction sets with rich addressing modes such as Hexagon's), and original application (to improve the spread across MediaBench).

Alternative Values. To evaluate the impact of using alternative values, we implemented two versions of the constraint model: one using alternative values; and one without alternative values to serve as baseline, thereby mimicking the capabilities of the basic model introduced in [29]. All instances of the baseline model were solved to optimality using C_{relaxed} as lower cost bound and no upper bound, with the exception of the `build_ycc_rgb_t` and `gpk_open` functions where the solver timed out even after half an hour. To reduce experiment runtime, we used the costs of the optimal solutions to the baseline model as upper bounds for the instances of the other model, which were then solved with a timeout of 1800 s. Since the intention of alternative values is to increase code quality, we expect to see an overall execution speedup over the baseline model.

Fig. 7 shows the estimated execution speedup of using alternative values. The geometric mean speedup is 5.4%, and for every function the constraint model with alternative values produced code that is better or equally good as that produced by the baseline model. Hence alternative values yield significant and consistent code quality improvements over the baseline model.

Alternative values primarily yield speedup when the function is data-bound. For example, the `gl_init_lists` function – which initializes a large data structure – consists mainly of instructions that write constants to memory. Many of these constants are used more than once and are often too large to fit as immediates to the instructions. This leads to many redundant copy instructions emitted by the baseline model which are not needed when making use of alternative values. In

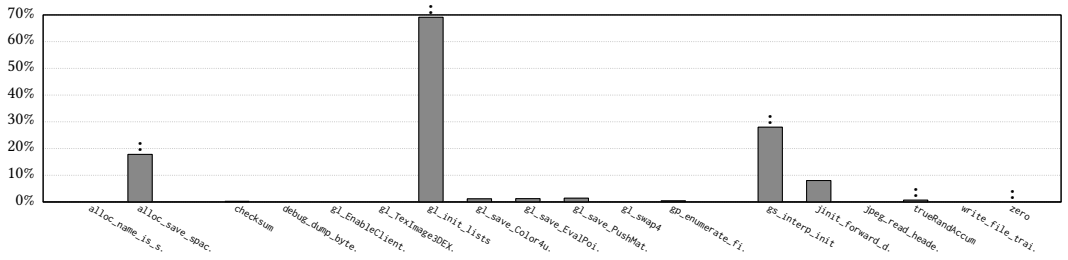


Fig. 7. Estimated execution speedup with alternative values. Bars marked with dots indicate solver timeouts.

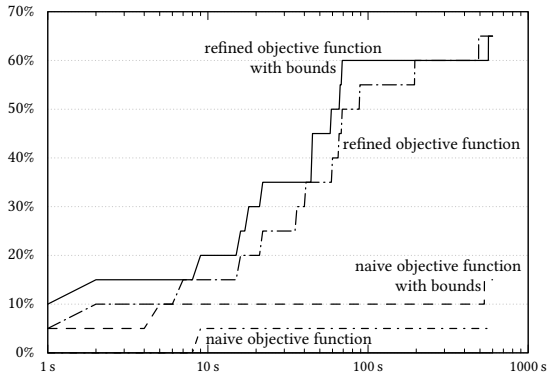


Fig. 8. Cumulative number of optimality proofs found over time using four constraint models.

contrast, no speedup is gained for the `alloc_name_is_s` function – which consists of a long chain of *if-then-else* checks – where most of its values are only used once throughout the entire function.

Compilation Time. To evaluate the impact of using the refined objective function and cost bounding, we implemented four more versions of the constraint model: one using the naive objective function given in Eq. 16; one using the refined objective function given in Eq. 17–19; one combining the naive function with bounds; and one combining the refined function with bounds. All models used alternative values and C_{relaxed} and C_{LLVM} as the lower and upper bound, respectively. We then produced code for the same functions and target as in the previous experiment, with a timeout of 60 s for computing the lower bound and 600 s for solving the model, and measured the solving time for proving optimality for each model. We expect the first model to be outperformed by the second and third models, which in turn are expected to both be outperformed by the fourth model. All solving times were averaged over 10 runs, for which the maximum coefficient of variation was 9.6%. The time spent on pattern matching was negligible (less than 2% of the total runtime, which was dominated by the solving time).

Fig. 8 shows the cumulative frequency distribution of optimality proofs found over time, which includes lower bound computation time and solving time. We see clearly a significant increase in scalability from using the refined objective function, which manages to prove optimality for UF graphs with up to 1415 nodes, compared to the naive function as used in [29], which only scales to 203 nodes (the increase is not due to more memory as no function required more than 4 GB during solving, and the hardware used in this experiment was only 19% faster [1]). We also observe that bounding has an overall positive impact for both objective functions, with the exception of one long-running case where cost bounding seems to have a negative impact on the solver.

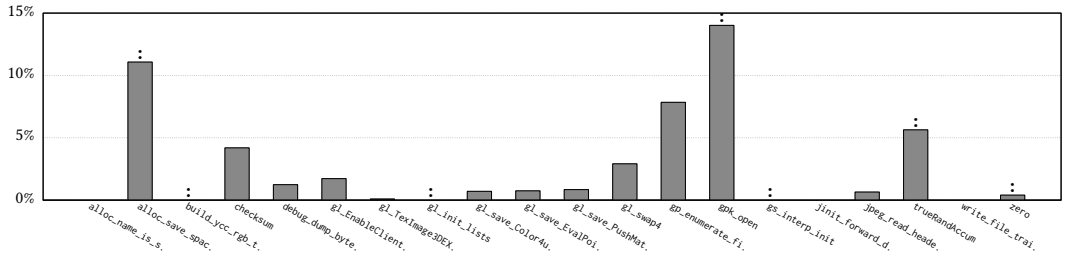


Fig. 9. Estimated execution speedup over LLVM. Bars marked with dots indicate solver timeouts.

Comparison with LLVM. To evaluate how our approach compares to the state of the art, we compared the quality of the code emitted by our combined model implementation against the code emitted by the instruction selector in LLVM, which is a greedy heuristic that mimics *maximum munch* (always pick largest match) [28]. We used the same functions and target as in the alternative values and compilation time experiments, with a timeout of 600 s for the solver, using no upper bound. All estimated execution cycle counts were averaged over 10 runs, for which there was no variation. As LLVM is limited to tree-shaped patterns, is unable to move operations across blocks, and relies on local-based, greedy heuristics – thereby limiting use of complex instructions – we expect to see an overall improvement.

Fig. 9 shows the estimated execution speedup over LLVM. The geometric mean speedup is 2.5%, hence our approach is competitive with LLVM. The cases showing a speedup are predominantly due to global code motion and block ordering. In the `alloc_save_spac`, `gp_enumerate_fi`, and `gpk_open` functions, for example, our approach is able to lift operations – constant loads in particular – out of blocks with high execution frequency into blocks with lower frequency, thereby reducing the cost. In the `jpeg_read_header`, `gl_TexImage3DEX`, and `gl_EnableClient` functions, our approach rearranges the blocks to remove one to two jump instructions. Also, in `alloc_save_spac` our approach is able to move an addition and memory operation into the same block and implement both using a single auto-increment memory instruction, whereas LLVM must implement these operations using two instructions. Improvements such as these become possible only when integrating global instruction selection with global code motion.

In three cases (`gl_swap4`, `gp_enumerate_fi`, and `zero`), LLVM is able to recompute memory addresses as part of the memory instruction and eliminate common subexpressions. Although none of these features is directly supported in our approach, we could work around these limitations by adding special patterns that exactly cover the operations involved in the recomputations or common subexpression eliminations. Without these special patterns, our approach would produce worse code than LLVM for the three cases above.

9 RELATED WORK

Several local, linear-time, least-cost covering algorithms exist for tree-based block and pattern graphs [2, 24, 46], which subsequently have been extended to DAG-based block graphs at the loss of optimality [20, 21, 33]. Together with instruction scheduling and register allocation, instruction selection has also been approached using integer programming [8, 25, 54] and constraint programming [7, 23, 44]. Far fewer methods exist for global instruction selection, which so far only has been approached as a partitioned Boolean quadratic problem [11, 18, 19] or using constraint programming [29]. Common among these techniques is that they are restricted to tree-based and DAG-based patterns, whereas our approach can operate on full-fledged pattern graphs.

Global code motion has been solved both in isolation [15, 22] as well as in integration with register allocation [6, 31]. Moreover, attempts at exploiting fine-grained instruction level parallelism using SIMD instructions [35, 42] suggest that the selection of such instructions should be done together with global code motion. Although many methods exist for selecting SIMD instructions in isolation, with a few that combine it with instruction selection [4, 41, 51], but to the best of our knowledge only our approach performs these three tasks in unison.

Several attempts have been made to reduce the number of jump instructions through block reordering [9, 22]. Common among these approaches is that they apply tailored algorithms for finding, replacing, and removing conditional jump instructions, whereas our approach rely on the selection of special patterns to perform these actions. Jump instructions can also be removed through *if conversions* (in some architectures, instructions can be predicated with a Boolean register that decide whether the instruction should be executed) [3, 27, 32, 53], which can be supported by our approach if the pattern set is extended with the appropriate patterns.

An idea similar to alternative values is introduced in [13] to address a related problem that appears in register allocation and instruction scheduling, where spilled values may be reused by multiple instructions when they are loaded back from memory.

The problem of cyclic data dependencies has been considered in several instruction selection techniques [18, 50]. In [18] such dependencies are found by identifying strongly connected components in the function graph and then forbidding conflicting decisions via a cost matrix. In [50] the approach instead relies on a dependency graph but, unlike our approach, applies iterative and greedy methods for breaking cycles.

10 CONCLUSIONS AND FUTURE WORK

This paper introduces a *complete* and *practical* approach to universal instruction selection: it handles memory computations and function calls; it inserts additional jump instructions where necessary; it ensures correct combinations of instructions; it improves code quality; and it increases scalability. The approach is demonstrated to be competitive with LLVM and potentially optimal (w.r.t. the model) for medium-sized functions targeting hardware which is typical for embedded systems.

Future work includes addressing the model limitations and increasing robustness and scalability by strengthening propagation with more implied and dominance breaking constraints and presolving techniques. We also intend to integrate existing models for register allocation and instruction scheduling [12, 13].

In addition, many peephole optimizations can be readily modeled as patterns, and several tools exist for automatic discovery and verification of such patterns [5, 10, 43]. We intend to exploit these tools to automatically extend the pattern set with these patterns in order to further improve code quality.

ACKNOWLEDGMENTS

The authors wish to thank Sebastian Buchwald for bringing cyclic data dependencies to our attention and the anonymous reviewers for their valuable comments and helpful suggestions.

REFERENCES

[1] CPU Benchmarks – Single Thread Performance. PassMark Software. URL: http://www.cpubenchmark.net/singleThread.html, updated June 2, 2017.

[2] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL '83*, pages 177–189, 1983.

[4] M. A. Arslan and K. Kuchcinski. Instruction Selection and Scheduling for DSP Kernels on Custom Architectures. In *DSD'13*. IEEE, 2013.

[5] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *ASPLOS'06*, pages 394–403. ACM, 2006.

[6] G. Barany and A. Krall. Optimal and Heuristic Global Code Motion for Minimal Spilling. In *CC'13*, pages 21–40. Springer, 2013.

[7] S. Bashford and R. Leupers. Constraint Driven Code Selection for Fixed-Point DSPs. In *DAC'99*, pages 817–822. ACM/IEEE, 1999.

[8] A. Bednarski and C. W. Kessler. Optimal Integrated VLIW Code Generation with Integer Linear Programming, 2006.

[9] J. Boender and C. Sacerdoti Coen. On the Correctness of a Branch Displacement Algorithm. In *TACAS'14*, pages 605–619. Springer, 2014.

[10] S. Buchwald. Optgen: A Generator for Local Optimizations. In *CC'15*, pages 171–189. Springer, 2015.

[11] S. Buchwald and A. Zwinkau. Instruction Selection by Graph Transformation. In *CASES'10*, pages 31–40, 2010.

[12] R. Castañeda Lozano, M. Carlsson, F. Drejhammar, and C. Schulte. Constraint-based Register Allocation and Instruction Scheduling. In *CP'12*, pages 750–766. Springer.

[13] R. Castañeda Lozano, M. Carlsson, G. Hjort Blindell, and C. Schulte. Combinatorial Spill Code Optimization and Ultimate Coalescing. In *LCTES'14*, pages 23–32. ACM, 2014.

[14] G. G. Chu. *Improving Combinatorial Optimization.* PhD thesis, The University of Melbourne, Australia, 2011.

[15] C. Click. Global Code Motion/Global Value Numbering. In *PLDI'95*, pages 246–257. ACM, 1995.

[16] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[18] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized Instruction Selection Using SSA-Graphs. In *LCTES'08*, pages 31–40. ACM, 2008.

[19] E. Eckstein, O. König, and B. Scholz. Code Instruction Selection Based on SSA-Graphs. In *SCOPES'03*, pages 49–65. ACM, 2003.

[20] M. A. Ertl. Optimal Code Selection in DAGs. In *POPL'99*, pages 242–249. ACM, 1999.

[21] M. A. Ertl, K. Casey, and D. Gregg. Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata. In *PLDI'06*, pages 52–60. ACM, 2006.

[22] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7): 478–490, 1981.

[23] A. Floch, C. Wolinski, and K. Kuchcinski. Combined Scheduling and Instruction Selection for Processors with Reconfigurable Cell Fabric. In *ASAP'10*, pages 167–174. IEEE, 2010.

[24] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

[25] C. H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *ISSS'97*, pages 41–47. IEEE, 1997.

[26] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-driven SSA Form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995.

[27] T. Granlund and R. Kenner. Eliminating Branches Using a Superoptimizer and the GNU C Compiler. In *PLDI'92*, pages 341–352. ACM, 1992.

[28] G. Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications.* Springer, 2016. ISBN 978-3-319-34017-3.

[29] G. Hjort Blindell, R. Castañeda Lozano, M. Carlsson, and C. Schulte. Modeling Universal Instruction Selection. In *CP'15*, pages 609–626. Springer, 2015.

[30] D. B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[31] N. Johnson and A. Mycroft. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In *CC'03*, pages 1–16. Springer, 2003.

[32] A. Jordan, N. Kim, and A. Krall. IR-level Versus Machine-level If-conversion for Predicated Architectures. In *ODES'13*, pages 3–10. ACM, 2013.

[33] D. R. Koes and S. C. Goldstein. Near-Optimal Instruction Selection on DAGs. In *CGO'08*, pages 45–54. IEEE/ACM, 2008.

[34] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[35] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *PLDI'00*, pages 145–156. ACM, 2000.

[36] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.
[37] J.-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1): 29–127, 1978.
[38] Y. Law and J. Lee. Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction. *Constraints*, 11 (2-3):221–267, 2006.
[39] C. Lecoutre and R. Szymanek. Generalized Arc Consistency for Positive Table Constraints. In *CP'06*, pages 284–298. Springer, 2006.
[40] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *MICRO'97*, pages 330–335. IEEE, 1997.
[41] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *DATE'00*, pages 4–8. IEEE, 2000.
[42] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. Kandemir. A Compiler Framework for Extracting Superword Level Parallelism. In *PLDI'12*, pages 347–358. ACM, 2012.
[43] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably Correct Peephole Optimizations with Alive. In *PLDI'15*, pages 22–32. ACM, 2015.
[44] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. Constraint-Driven Instructions Selection and Application Scheduling in the DURASE System. In *ASAP'09*, pages 145–152. IEEE, 2009.
[45] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP'07*, pages 529–543. Springer, 2007.
[46] E. Pelegrí-Llopart and S. L. Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. In *POPL'88*, pages 294–308. ACM, 1988.
[47] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites. In *ISPASS'05*, pages 10–20. IEEE, 2005.
[48] *Hexagon V5/V55 Programmer's Reference Manual.* Qualcomm Technologies Inc., Aug 2013. 80-N2040-8 Rev. A.
[49] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., 2006. ISBN 0-444-52726-5.
[50] V. Sarkar, M. J. Serrano, and B. B. Simons. Register-sensitive Selection, Duplication, and Sequencing of Instructions. In *ICS'01*, pages 277–288. ACM, 2001.
[51] H. Tanaka, S. Kobayashi, Y. Takeuchi, K. Sakanushi, and M. Imai. A Code Selection Method for SIMD Processors with PACK Instructions. In *SCOPES'03*, pages 66–80. Springer, 2013.
[52] V. Živojnović, J. Martínez Velarde, C. Schläger, and H. Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *ICSPAT'94*, pages 715–720. Miller Freeman, 1994.
[53] N. J. Warter, S. A. Mahlke, W.-M. W. Hwu, and B. R. Rau. Reverse If-Conversion. In *PLDI'93*, pages 290–299. ACM, 1993.
[54] T. Wilson, G. Grewal, B. Halley, and D. Banerji. An Integrated Approach to Retargetable Code Generation. In *ISSS'94*, pages 70–75. IEEE, 1994.