Modeling Universal Instruction Selection

Gabriel Hjort Blindell[1], Roberto Castañeda Lozano[2,1], Mats Carlsson[2], and Christian Schulte[1,2]

[1] SCALE, School of ICT, KTH Royal Institute of Technology, Sweden
{ghb, cschulte}@kth.se
[2] SCALE, Swedish Institute of Computer Science, Sweden
{rcas, matsc}@sics.se

Abstract. Instruction selection implements a program under compilation by selecting processor instructions and has tremendous impact on the performance of the code generated by a compiler. This paper introduces a graph-based *universal* representation that unifies data and control flow for both programs and processor instructions. The representation is the essential prerequisite for a constraint model for instruction selection introduced in this paper. The model is demonstrated to be expressive in that it supports many processor features that are out of reach of state-of-the-art approaches, such as advanced branching instructions, multiple register banks, and SIMD instructions. The resulting model can be solved for small to medium size input programs and sophisticated processor instructions and is competitive with LLVM in code quality. Model and representation are significant due to their expressiveness and their potential to be combined with models for other code generation tasks.

1 Introduction

Instruction selection implements an input 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 over four decades (see [23] for a recent survey). It is typically decomposed into identifying the applicable instructions and selecting a combination of applicable instructions to meet the semantics of the input program. Combinations differ in efficiency and hence selecting one is naturally an optimization problem. Finding an efficient combination is crucial as efficiency might differ by up to two orders of magnitude [35].

Common approaches use graph-based techniques that operate on the data-flow graph of a program (nodes represent operations, edges describe data flow). However, state-of-the-art approaches are restricted to trees or DAGs to avoid NP-hard methods for general graphs. This restriction is severe: ad-hoc routines are needed for handling control flow; many instructions of modern processors, such as DSPs (digital signal processors), cannot be handled; and the scope of instruction selection is typically *local* to tiny parts of the input program and hence by design forsakes crucial optimization opportunities.

This paper introduces a *universal* representation based on general graphs. It is universal as it simultaneously captures both data and control flow for both

programs and processor instructions. By that it overcomes the restrictions of current approaches. In particular, the universal representation enables a simple treatment of *global code motion*, which lets the selection of instructions to be *global* for an entire function. The representation is compatible with state-of-the-art compilers; the paper uses LLVM [26] as compiler infrastructure.

However, the very reason of the proposed approach is that instruction selection can be expressed as a constraint model. Due to the expressiveness of the universal representation, the introduced model accurately reflects the interaction between control and data flow of both programs and processor instructions. The paper presents the model in detail and discusses how it supports processor features that are out of reach of state-of-the-art approaches, such as advanced branching instructions, multiple register banks, and SIMD instructions.

The paper shows that the described approach is feasible. The resulting model can be solved for small to medium size input programs and challenging SIMD processor instructions and is competitive with LLVM in code quality.

Model and representation are significant for two reasons. First, they are considerably more powerful than the state of the art for instruction selection and can capture common features in modern processors. Crucially, instruction selection with a universal representation is only feasible with an approach as expressive as a constraint model. Second, the paper's approach will be essential to integrate instruction selection with register allocation and instruction scheduling as the other code generation tasks that we have explored in previous work [11, 10]. It is only the combination of all three interdependent tasks that will enable generating optimal code for modern processors.

Paper outline. Section 2 introduces graph-based instruction selection and Sect. 3 introduces the representations that enable universal instruction selection. The corresponding constraint model is introduced in Sect. 4. Section 5 experimentally evaluates the paper's approach followed by a discussion of related work in Sect. 6. Section 7 concludes the paper.

2 Graph-based Instruction Selection

The most common approach for instruction selection is to apply graph-based methods. As is common, the unit of compilation is a single *program function*, which consists of a set of basic blocks. A *basic block*, or just *block*, is a sequence of *computations* (like an addition or memory load) and typically ends with a *control procedure* (like a jump or function return). Each block has a single entry point and a single exit point for execution. A program function has exactly one block as entry point, called *entry block*.

For a program function a *data-flow graph* called *program graph* is constructed, where each node represents a computation and each edge indicates that one computation uses the value produced by another computation. As a data-flow graph does not incorporate control-flow information, a single program function typically results in several program graphs (at least one per block). A *local*

instruction selector selects instructions for program graphs of a single block, whereas a *global* instruction selector does so for an entire function.

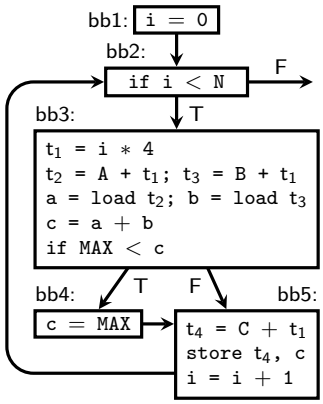
For each instruction of a given processor data-flow graphs called *pattern graphs* are also constructed. The set of all pattern graphs for a processor constitute a *pattern set*. Thus, the problem of identifying the applicable instructions is reduced to finding all instances where a pattern graph from the pattern set is subgraph isomorphic to the program graph. Such an instance is called a *match*.

A set of matches *covers* a program graph if each node in the program graph appears in exactly one match. By assigning a cost to each match that corresponds to the cost of the selected instruction, the problem of finding the best combination of instructions is thus reduced to finding a cover with least cost. It is well known that the subgraph isomorphism problem and the graph covering problem are NP-complete in general [8, 20], but can be solved optimally in linear time if the program and pattern graphs are trees [2].

Program trees are typically not expressive enough and hence modern compilers commonly use pattern trees and program DAGs, which are then covered using greedy heuristics. This, however, suffers from several limitations. First, pattern trees significantly limit the range of instructions that can be handled. For example, ad-hoc routines are required to handle even simple branch instructions as pattern trees cannot include control procedures. Second, program DAGs exclude handling of sophisticated instructions of modern processors that span multiple blocks. Third, more efficient code can be generated if certain computations can be moved across blocks, provided the program semantics is kept.

```
int i = 0;
while (i < N) {
  int c = A[i] + B[i];
  if (MAX < c)
    c = MAX;
  C[i] = c;
  i++;
}
```

(a) C code



(b) Corresponding IR and control-flow graph

Fig. 1: An example program that computes the saturated sums of two arrays, where `A`, `B`, and `C` are integer arrays of equal lengths and stored in memory, and `N` and `MAX` are integer constants representing the array length and the upper limit, respectively. An `int` value is assumed to be 4 bytes.

A Motivating Example. Assume the C program shown in Fig. 1a, which computes the saturated sums of two arrays. Saturation arithmetic "clamps" the computed value such that it stays within a specified range and does not wrap around in case of overflow, a property commonly desired in many digital signal processing applications. For simplicity the program in Fig. 1a only clamps on the upper bound, but the following argumentation can easily be extended to programs that clamp both bounds. Most compilers do not operate directly on source code but on the *internal representation (IR)* of a program, shown in Fig. 1b, which can be viewed as a high-level assembly language. Programs written in this form are typically portrayed as a *control-flow graph* where each node represents a block and each edge represents a potential jump from one block to another. Most modern compilers, however, employ a slightly different representation that will be discussed in Sect. 3.

Assume further that this program will be executed on a processor whose instruction set includes the following instructions: satadd computes the saturated sum of two integer values; repeat iteratively executes an instruction sequence a given number of times; and add4 can compute up to four ordinary integer sums simultaneously (a so-called *vector* or *SIMD (Single-Input Multiple-Data)* instruction). Clearly, this program would benefit from selecting the satadd instruction to compute the value c and from selecting the repeat instruction to implement the control of the loop consisting of blocks bb2 through bb5. What might be less obvious, however, is the opportunity to select the add4 instruction to compute values t_2 and t_3 together with t_4 and i. Since these additions all reside inside the loop and are independent from one another, they can be computed in parallel provided they can be performed in the same block. This notion of moving computations across blocks is referred to as *global code motion*.

But taking advantage of these instructions is difficult. First, describing the saturated sum involves computations and control procedures that span several blocks. However, the state of the art in instruction selection is limited to local instruction selection or cannot handle control procedures. Second, to maximize the utility of the add4 instruction the additions for producing values t_4 and i must be moved from bb5 to bb3. But it is not known how to perform global code motion in conjunction with instruction selection, and hence all existing representations that describe entire program functions inhibit moves by pre-placing each computation to a specific block. Third, for many processors the registers used by vector instructions are different from those of other instructions. Consequently, selecting the add4 instruction might necessitate further instructions for copying data between registers. These additional instructions can negate the gain of using the add4 instruction or, in the worst case, even degrade the quality of the generated code. Making judicious use of such instructions therefore requires that the instruction selector is aware of this overhead. Fourth, since the program graph must be covered exactly, the computation of the value i cannot be implemented by both the add4 and repeat instructions. Instruction selection must therefore evaluate which of the two will result in the most efficient code, which depends on their relative costs and the restrictions imposed by the processor.

```
bb1: int i1 = 0;
bb2: int i2 = φ(i1:bb1, i3:bb5);
     if (i2 < N) goto bb3; else goto end;
bb3: int c1 = A[i2] + B[i2];
     if (MAX < c1) goto bb4; else goto bb5;
bb4: int c2 = MAX;
bb5: int c3 = φ(c1:bb3, c2:bb4);
     C[i2] = c3;  int i3 = i2 + 1;
     goto bb2;
```

Fig. 2: The C program from Fig. 1a in SSA form.

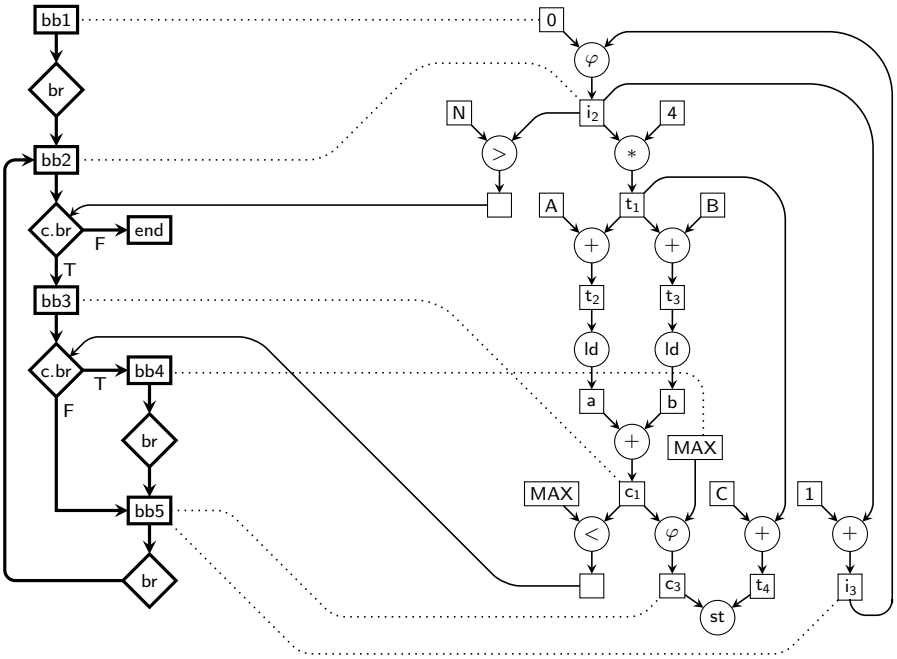
3 Representations for Universal Instruction Selection

This section introduces representations for both programs and instructions and how they can be used to express covering and global code motion.

Program Representation. The key idea is to combine both data flow and control flow in the very same representation. We start by modifying control-flow graphs such that the nodes representing blocks no longer contain any computations. We refer to these as *block nodes*. In addition, the control procedures previously found within the blocks now appear as separate *control nodes*, where each control node has exactly one inbound control-flow edge indicating to which block the procedure belongs. Consequently, the control-flow edges that previously originated from the block nodes now originate from the control nodes. An example will be provided shortly.

To capture the data flow of entire program functions as a data-flow graph we use the *SSA (Static Single Assignment) graph* constructed from programs in *SSA form*. SSA is a state-of-the-art program representation where each program variable must be defined only once [14]. When the definition depends on the control flow, SSA uses so-called *φ-functions* to disambiguate such cases by taking a value and the block from where it originates as arguments. Fig. 2 shows the C program from Fig. 1a in SSA form, and the corresponding control-flow and SSA graphs are shown in Fig. 3a and Fig. 3b, respectively. Originally the SSA graph only consists of nodes representing computations – called *computation nodes* – but we extend it such that each value is represented by a *value node*. Also note that copying of values is not represented as separate computation nodes in the SSA graph. Therefore, in Fig. 3b the program variables i_1 and c_2 have been replaced by 0 and MAX, respectively.

To unify the control-flow graph and the SSA graph, we first add data-flow edges from value nodes to the control nodes that make use of the corresponding values. For the control-flow graph shown in Fig. 3a this entails the c.br nodes, which represent conditional jumps. Note that the SSA graph does not indicate in which block a given computation should be performed. Although we want to keep such pre-placements to a minimum, having *no* pre-placements permits moves that violate the program semantics. For example, in Fig. 1b the assignment



(a) Control-flow graph (b) SSA graph

Fig. 3: The program graph constructed from the program shown in Fig. 2. 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.

`c = MAX` must be performed in block `bb4` as its execution depends on whether `MAX < c` holds. Fortunately, these cases can be detected whenever a φ-function appears in the program. The SSA-based program in Fig. 2, for example, has in block `bb2` a statement $\texttt{i}_2 = \varphi(\texttt{i}_1\texttt{:bb1, i}_3\texttt{:bb5})$. Thus, the program variable $\texttt{i}_2$ is assigned either the value $\texttt{i}_1$ or the value $\texttt{i}_3$ depending on whether the jump to `bb2` was made from block `bb1` or block `bb5`. These conditions can be ensured to hold in the generated code by requiring that (i) due to the arguments to the φ-function, the values $\texttt{i}_1$ and $\texttt{i}_3$ must be computed in blocks `bb1` and `bb5`, respectively; and (ii) due to the location of the φ-function in the program, the value $\texttt{i}_2$ must be assigned its value in block `bb2`. We encode these constraints into the program graph by introducing *definition edges* to signify that a certain value must be produced in a specific block. Hence, in case of the example above three definition edges are added: one from value node `0` to block node `bb1`, one from value node $\texttt{i}_3$ to block node `bb5`, and another from value node $\texttt{i}_2$ to block node `bb2`. Such edges are also added for the statement $\texttt{c}_3 = \varphi(\texttt{c}_1\texttt{:bb3, c}_2\texttt{:bb4})$, which results in the program graph shown in Fig. 3.

Instruction Representation. The procedure for constructing pattern graphs is almost identical to constructing program graphs. The only exception is that the control-flow graph of a pattern graph could be empty, which is the case for instructions whose result does not depend on any control flow. For example, Fig. 4 shows the pattern graph of satadd (introduced in Sect. 2), which has a substantial control-flow graph. In comparison, the pattern graph of a regular add instruction would only comprise an SSA graph, consisting of one computation node and three value nodes. Like with the program graph, it is assumed that all pattern graphs with a non-empty control-flow part have exactly one block node representing the instruction's entry point.

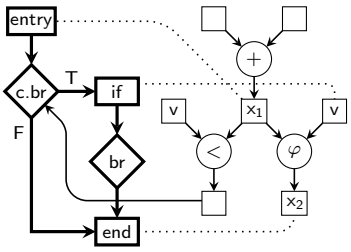


Fig. 4: satadd's pattern graph.

Since the program graph now consists of several kinds of nodes, we need to refine the notion of coverage. A program graph is *covered* by a set of matches if each *operation* in the program graph appears in exactly one match from the set, where an operation is either a computation or a control node. Likewise, a match *covers* the operations in the program graph corresponding to the operations in the pattern graph. Consequently, matches are allowed to partially overlap on the block and value nodes in the program graph. This property is deliberate as it enables several useful features, which will be seen shortly.

But not all covers of a given program graph yield valid solutions to the global code motion problem. For example, assume a cover of Fig. 3 where a match corresponding to the add4 instruction has been selected to cover the computation nodes that produce values t_2, t_3, and i_3. Because t_2 and t_3 are data-dependent on value i_2, which must be produced in block bb2 due to a definition edge, these values cannot be produced earlier than in bb2. Likewise, because value c_1 must be produced in block bb3 and is data-dependent on t_2 and t_3, these values cannot be produced later than in bb3. At the same time, i_3 must be produced in block bb5. Hence no single instruction that computes t_2, t_3, and i_3 can be placed in a block such that all conditions imposed by the program graph are fulfilled.

We use the above observation to formalize the global code motion problem as follows. If a *datum* refers to a particular value node in the program graph, a match m *defines* respectively *uses* a datum d if there exists an inbound respectively outbound data-flow edge to d in the pattern graph of m. Hence a datum can be both defined and used by the same match. We also refer to the data used but not defined by a match as its *input data* and to the data defined but not used as its *output data*. Next, a block b in the program graph *dominates* another block b' if every control-flow path from the program function's entry block to b' goes through b. By definition, a block always dominates itself. Using these notions, we define a placement of selected matches to blocks to be a solution to the global code motion problem if each datum d in the program graph is defined in some block b such that b dominates every block wherein d is used. Note that this definition excludes control procedures because moving such operations to another block rarely preserves the semantics of the program.

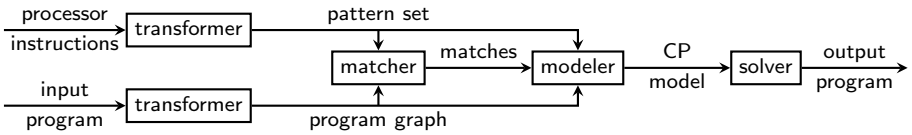


Fig. 5: Overview of our approach.

Some instructions impact both the covering and global code motion problems simultaneously. Assume for example a match in Fig. 3 of the pattern graph from Fig. 4, which corresponds to the satadd instruction. A match *spans* across the blocks in the program graph corresponding to the blocks appearing in the pattern graph. Hence, the match above spans across blocks bb3, bb4, and bb5. Of these, we note that the control procedures involving bb4 are all covered by this match. Consequently, the computations performed within this block must all be implemented by the satadd instruction. The match therefore *consumes* bb4, meaning no other matches can be placed in this block. Consequently, a universal instruction selector must take both the covering problem and the global code motion problem into account when selecting the matches.

4 A Constraint Model for Universal Instruction Selection

This section introduces a constraint model for universal instruction selection. An overview of the approach is shown in Fig. 5.

Match Identification. For a given program graph G and pattern set P, the matches are identified by finding all instances M where a pattern graph in P is subgraph isomorphic to G. Hence, separating identification from selection of matches is an optimality-preserving decomposition. We use an implementation of VF2 [13] to solve the subgraph isomorphism problem.

Depending on the pattern graph, certain matches may lead to cyclic data dependencies and must therefore be excluded from the set of identified matches. An example is a match of add4 in Fig. 3 defining t_2 or t_3 together with c_1. Such matches can be detected by finding all connected components in the match and checking whether a path exists between two components in the program graph.

Global Instruction Selection. The set of variables $\mathbf{sel}(m) \in \{0, 1\}$ models whether a match $m \in M$ is selected, where M denotes the set of identified matches. As explained in Sect. 3, each operation $o \in O$, where O denotes the set of operations in the program graph, must be covered by exactly one of the selected matches. Hence, if $\mathsf{covers}(m) \subseteq O$ denotes the set of operations covered by match m, then the condition can be expressed as:

$$\sum_{\substack{m \in M \text{ s.t.} \\ o \in \mathsf{covers}(m)}} \mathbf{sel}(m) = 1 \quad \forall o \in O \tag{1}$$

Global Code Motion. The set of variables $\mathbf{place}(m) \in B \cup \{b_{\text{null}}\}$ models in which block a match m is placed. B denotes the set of blocks in the program graph, and b_{null} denotes an additional block (not part of the program graph) in which non-selected matches are placed. In other words:

$$\mathbf{sel}(m) \Leftrightarrow \mathbf{place}(m) \neq b_{\text{null}} \quad \forall m \in M \tag{2}$$

where $\mathbf{sel}(m)$ abbreviates $\mathbf{sel}(m) = 1$.

Control procedures cannot be placed in another block than originally indicated in the program graph. Let $\mathsf{entry}(m) \in B$ denote the entry block of match m if the pattern graph of m has such a node, otherwise $\mathsf{entry}(m) \notin B$. This condition can then be expressed as:

$$\mathbf{sel}(m) \Rightarrow \mathbf{place}(m) = \mathsf{entry}(m) \quad \forall m \in M, \mathsf{entry}(m) \in B \tag{3}$$

The set of variables $\mathbf{def}(d) \in B$ models in which block a datum $d \in D$ is defined, where D denotes the set of data in the program graph. As explained in Sect. 3, each datum d must be defined in some block b such that b dominates every block wherein d is used. In addition, the conditions imposed by the definition edges must be maintained. Let $\mathsf{dom}(b) \subseteq B$ denote the set of blocks that dominate block b, where it is assumed that $\mathsf{dom}(b_{\text{null}}) = B$. Also let $\mathsf{uses}(m) \subseteq D$ denote the set of data used by match m, and let DE denotes the set of definition edges in the program graph. The conditions above can then be expressed as:

$$\mathbf{def}(d) \in \mathsf{dom}(\mathbf{place}(m)) \quad \forall m \in M, \forall d \in \mathsf{uses}(m) \tag{4}$$

$$\mathbf{def}(d) = b \quad \forall \{d, b\} \in DE \tag{5}$$

The $\mathbf{def}(\cdot)$ variables must be connected to the $\mathbf{place}(\cdot)$ variables. Intuitively, if a selected match m is placed in block b and defines a datum d, then d should also be defined in b. However, a direct encoding of this condition leads to an over-constrained model. Assume again a match in Fig. 3 of the pattern graph from Fig. 4, which thus defines the values c_1 and c_3. Due to the definition edges incurred by the φ-node, c_1 and c_3 must be defined in blocks $\mathsf{bb3}$ and $\mathsf{bb5}$, respectively. But if $\mathbf{def}(d) = \mathbf{place}(m)$ is enforced for every datum d defined by a match m, then the match above will never become eligible for selection because c_1 and c_3 must be defined in different blocks whereas a match can only be placed in a single block. In such cases it is sufficient to require that a datum is defined in any of the blocks spanned by the match. Let $\mathsf{spans}(m) \subseteq B$ denote the set of blocks spanned by match m and $\mathsf{defines}(m) \subseteq D$ denote the set of data defined by m. Then the condition can be relaxed by assigning $\mathbf{def}(d)$ to $\mathbf{place}(m)$ when $\mathsf{spans}(m) = \varnothing$, otherwise to any of the blocks in $\mathsf{spans}(m)$. Both these conditions can be combined into a single constraint:

$$\mathbf{sel}(m) \Rightarrow \mathbf{def}(d) \in \{\mathbf{place}(m)\} \cup \mathsf{spans}(m) \quad \forall m \in M, \forall d \in \mathsf{defines}(m) \tag{6}$$

Finally, matches cannot be placed in blocks that are consumed by some selected match. If $\mathsf{consumes}(m) \subseteq B$ denotes this set for a match m, then this condition can be expressed as:

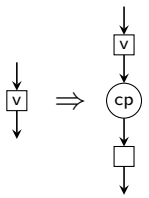
$$\mathbf{sel}(m) \Rightarrow \mathbf{place}(m') \neq b \quad \forall m, m' \in M, \forall b \in \mathsf{consumes}(m) \tag{7}$$

Data Copying. Instructions typically impose requirements on the values that they operate on, for example that its input and output data must be located in particular registers. Combinations of such instructions may therefore require additional copy instructions to be selected in order to fulfill these requirements.

The set of variables $\mathbf{loc}(d) \in L \cup \{l_{\text{null}}\}$ models in which location a datum d is available. L denotes the set of possible locations, and l_{null} denotes the location of a value computed 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 $\mathsf{stores}(m, d) \subset L$ denotes the locations for a datum d permitted by a match m – where an empty set means no restrictions are imposed – then such conditions can be imposed as:

$$\mathbf{sel}(m) \Rightarrow d \in \mathsf{stores}(m, d) \quad \forall m \in M, \forall d \in D \text{ s.t. } \mathsf{stores}(m, d) \neq \varnothing \tag{8}$$

This alone, however, may cause no solutions to be found for processors where there is little overlap between the locations – like in the case of multiple register banks – that can be used by the instructions. This problem is addressed using a method called *copy extension*. Prior to identifying the matches, the program graph is expanded such that every data usage is preceded by a special operation called a *copy*, represented by *copy nodes*. This expansion is done by inserting a copy node and value node along each data-flow edge that represents a use of data, as shown in the figure to the right. The definition edges are also moved such that they remain on the data adjacent to the φ-nodes. The same expansion is also applied to the pattern graphs except for the data-flow edges that represent use of input data.



Consequently, between the output datum of one match that is the input datum to another match, there will be a copy node that is not covered by either match. It is therefore assumed that a special *null-copy pattern*, consisting of a single copy node and two value nodes, is always included in the pattern set. A match derived from the null-copy pattern has zero cost but requires, if selected, that both data are available in the same location. This means that if the null-copy pattern can be used to cover some copy node, then the two matches connected by that copy node both use the same locations for its data. Hence there is no need for an additional instruction to implement this copy. If the locations are not compatible, however, then a match deriving the null-copy pattern cannot be selected as that would violate the requirement that the locations must be the same. An actual instruction is therefore necessary, and the restrictions on the $\mathbf{loc}(\cdot)$ variables ensure that the correct copy instruction is selected.

Fall-Through Branching. Branch instructions often impose constraints on the distance that can be jumped from the instruction to a particular block. For example, conditional branch instructions typically require that the FALSE block be located directly after the branch instruction, which is known as a *fall-through*. This condition may also be imposed by instructions that span multiple blocks.

Consequently, the order in which the blocks appear in the generated code is interconnected with the selection of certain matches.

The set of variables $\mathbf{succ}(b) \in B \cup \{b_{\text{null}}\}$ models the successor of block b. A valid block order then corresponds to a cycle formed by the $\mathbf{succ}(\cdot)$ variables. Using the global $\mathsf{circuit}$ constraint [27], this condition can be expressed as:

$$\mathsf{circuit}\left(\cup_{b \in B \cup \{b_{\text{null}}\}} \{\mathbf{succ}(b)\}\right) \tag{9}$$

$$\mathbf{succ}(b_{\text{null}}) = b_{\text{entry}} \tag{10}$$

Hence, if the instruction corresponding to a match m performs a fall-through to a block b, then this condition can be expressed as $\mathbf{sel}(m) \Rightarrow \mathbf{succ}(\mathbf{place}(m)) = b$. This approach can readily be extended to incorporate generic jump distances as required by some processors.

Objective Function. The objective function depends on the desired characteristics of the generated code. For example, if the compiler should maximize the performance of the program under compilation, then the execution time of each block should be minimized. In addition, the execution time should be weighted with the relative execution frequency of each block. Thus, if $\mathsf{freq}(b) \in \mathbb{N}_1$ denotes the relative execution frequency of block b, then the objective function to be minimized is:

$$\sum_{b \in B} \mathsf{freq}(b) \times \sum_{\substack{m \in M \text{ s.t.} \\ \mathbf{place}(m) = b}} \mathsf{cost}(m) \tag{11}$$

where $\mathsf{cost}(m) \in \mathbb{N}_0$ is the relative time it takes to execute the instruction corresponding to a match m. Note that non-selected matches are always placed in the b_{null} block, which is not part of the B set. If the compiler should optimize for code size, then the weight $\mathsf{freq}(b)$ is dropped from Eq. 11 and $\mathsf{cost}(m)$ is redefined as the size of the instruction corresponding to match m.

Implied Constraints. Similarly to Eq. 1, the set of data must be defined by the set of selected matches:

$$\sum_{\substack{m \in M \text{ s.t.} \\ d \in \mathsf{defines}(m)}} \mathbf{sel}(m) = 1 \quad \forall d \in D \tag{12}$$

For every block b in which a datum is defined, there must exist a selected match that either is placed in b or spans b:

$$\begin{gathered} \mathbf{def}(d) = b \Rightarrow \mathbf{sel}(m) \wedge b \in \{\mathbf{place}(m)\} \cup \mathsf{spans}(m) \\ \forall b \in B, \forall d \in D, \exists m \in M \end{gathered} \tag{13}$$

If two matches impose conflicting requirements on input or output data locations, or impose conflicting fall-through requirements, then at most one of these matches may be selected.

Dominance Constraints. By analyzing the constraints on the $\mathbf{loc}(\cdot)$ variables, one can identify subsets S of values such that any solution with $\mathbf{loc}(d) = v$ and $v \in S$ can be replaced by an equivalent solution with $\mathbf{loc}(d) = \max(S)$, for any $d \in D$. Consequently, all values in $S \setminus \{\max(S)\}$ can be a priori removed from the domains of all $\mathbf{loc}(\cdot)$ variables.

Suppose that there are two mutually exclusive matches m and m' with $\mathsf{cost}(m) \leq \mathsf{cost}(m')$ and the constraints imposed by m are compatible with and no stricter than the constraints imposed by m'. Then any solution that selects m' can be replaced by a solution of less or equal cost that selects m. Consequently, $\mathbf{sel}(m') = 0$ can be set a priori for all such m'. In case m and m' have identical cost and impose identical constraints, a lexicographic ordering rule is used.

Branching Strategy. Our branching strategy only concerns those $\mathbf{sel}(\cdot)$ variables of matches not corresponding to copy instructions. Let M' denote this set of matches. We branch on $\{\mathbf{sel}(m) \mid m \in M'\}$ ordered by non-increasing $|\,\mathsf{covers}(m)|$, trying $\mathbf{sel}(m) = 1$ before $\mathbf{sel}(m) = 0$. The intuition behind this is to eagerly cover the operations. The branching on the remaining decision variables is left to the discretion of the solver (see Sect. 5 for details).

Model Limitations. A constant value that has been loaded into a register for one use cannot be reused for other uses. Consequently, the number of selected matches may be higher than necessary. This problem is similar to spilling reused temporaries in [11] and can be addressed by adapting the idea of *alternative temporaries* introduced in [10].

For some processors, conditional jumps can be removed by predicating instructions with a Boolean variable that determines whether the instruction shall be executed [3]. For example, assume that the statement `c = MAX` in Fig. 1b is implemented using a copy instruction. If this instruction can be predicated with the Boolean value `MAX < c`, then the conditional jump to block `bb4` becomes superfluous. This is known as *if-conversion*. Such instructions can be described using two pattern graphs: one representing the predicated version, and another representing the non-predicated version. But because every operation must be covered by exactly one match, the predicated version can only be selected if the match implements all computations in the conditionally executed block.

5 Experimental Evaluation

The model is implemented in MiniZinc [31] and solved with CPX [1] 1.0.2, which supports FlatZinc 1.6. The experiments are run on a Linux machine with an Intel Core i7-2620M 2.70 GHz processor and 4 GB main memory using a single thread.

We use all functions (16 in total) from MediaBench [28] that have more than 5 LLVM IR instructions and do not contain function calls or memory computations (due to limitations in the toolchain but not in the constraint model). The size of their corresponding program graphs ranges between 34 and 203 nodes. The

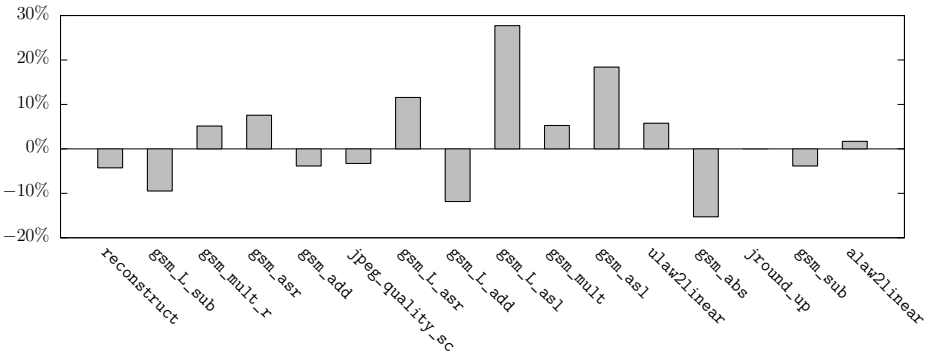


Fig. 6: Estimated execution speedup over LLVM for simple instructions.

functions are compiled and optimized into LLVM IR files using LLVM 3.4 [26] with the `-O3` flag (optimizes execution time). These files serve as input to our toolchain. However, as LLVM currently lacks a method re-entering its backend after instruction selection we cannot yet execute the code generated by our approach. Instead the execution time is estimated using LLVM's cost model.

In the following, the full runtimes comprising matching, flattening the MiniZinc model to FlatZinc, and solving are summarized. Thus the time for producing the LLVM IR files and transforming them into program graphs is not included. The time spent on matching is negligible (less than 1% of the full runtime), while the time spent on flattening is considerable (more than 84% of the full runtime). The solving time is measured until proof of optimality. All runtimes are averaged over 10 runs, for which the coefficient of variation is less than 6%.

Simple instructions. For proof of concept, the processor used is MIPS32 [33] since it is easy to implement and extend with additional instructions. The pattern graphs are manually derived from the LLVM instruction set description (to enable comparison), however the process could be easily automated as patterns are already described as trees in LLVM. As greedy heuristics already generate code of sufficient quality – in many cases even optimal – for MIPS32, we should not, in general, expect to see any dramatic execution speedup.

The shortest full runtime for the benchmarks is 0.3 seconds, the longest is 83.2 seconds, the average is 27.4 seconds, and the median is 10.5 seconds.

Fig. 6 shows the estimated execution speedup of our approach compared to LLVM. The geometric mean speedup is 1.4%; in average our approach is slightly better than LLVM. As the figure reveals, our approach has the potential to generate better code than the state of the art even for simple and regular architectures such as MIPS32, mainly due to its ability of moving computations to blocks with lesser execution frequency. The cases where our approach generates code that is worse than LLVM are due to the model limitations described in Sect. 4. Some of these cases are aggravated by the fact that LLVM's instruction selector is capable of, where appropriate, combining chains of binary φ-functions into a single

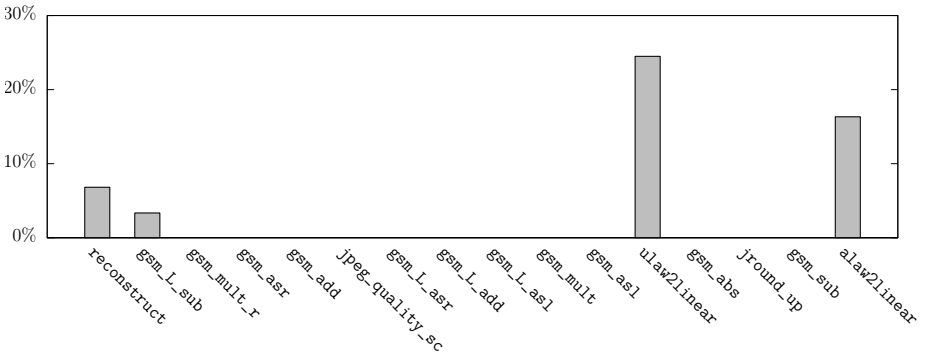


Fig. 7: Estimated execution speedup by adding SIMD instructions to MIPS32.

φ-function in order to reduce the number of branch operations. This feature has yet to be implemented in our toolchain.

An interesting aspect is that due to the branching described in Sect. 4 solving yields a very good first solution, which provides a tight upper bound. We conjecture that this together with the lazy clause learning in the CPX solver is the reason why we can prove optimality for all benchmarks.

SIMD instructions. In the following experiments we extend the MIPS32 architecture with SIMD instructions for addition, shifting, and Boolean logic operations as motivated by potential matches in the considered benchmark functions. The SIMD instructions use the same registers as the simple instructions.

The shortest full runtime for the benchmarks is 0.3 seconds, the longest is 146.8 seconds, the average is 44.2 seconds, and the median is 10.5 seconds.

Fig. 7 shows the estimated execution speedup for MIPS32 together with SIMD instructions. The geometric mean speedup compared to our approach for basic MIPS32 is 3%. The best cases correspond to functions that are computation-intensive (e.g. `ulaw2linear`) as more data parallelism can be exploited by the SIMD instructions. The worst cases (no improvement) correspond to control-intensive functions (e.g. `gsm_L_asr`), where there are either no matches for SIMD instructions or matches that are not profitable. In the latter case, the solver often discards SIMD matches that would require moving operations to *hot* blocks which are estimated to be executed often. A SIMD instruction in a hot block would be more costly than multiple primitive operations in the *colder* blocks where they originally reside. Our approach is unique in that it reflects this trade-off accurately. A traditional approach – vectorization first, then instruction selection – would greedily select the SIMD instruction and generate worse code. If a certain primitive operation must already be executed in a hot block then the solver will bring in operations of the same type from colder blocks to form a *speculative* SIMD instruction (this is the case e.g. for `alaw2linear`).

Hence our approach can exploit sophisticated instructions (such as SIMD) and has the potential to improve over traditional approaches since it accurately

reflects the trade-offs in instruction selection. We also expect to see further benefits when integrated with register allocation and instruction scheduling.

6 Related Work

Several linear time, optimal algorithms exist for local instruction selection on tree-based program and pattern graphs [2, 22, 32]. These have subsequently been extended to DAG-based program graphs [17, 18, 25], but at the loss of optimality. Together with instruction scheduling or register allocation, the problem has also been approached using integer programming (IP) [7, 21, 36] and constraint programming (CP) [6, 19, 30]. Far fewer methods exist for global instruction selection, which so far only has been approached as a partitioned Boolean quadratic problem [9, 15, 16]. Common among these techniques is that they are restricted to pattern trees or pattern DAGs.

Many methods exist for selecting vector instructions separately from instruction selection, but attempts have been made at combining these two tasks using IP [29, 34] and CP [4]. Of these, however, only [34] takes the cost of data copying into account, and none is global.

Global code motion has been solved both in isolation [12] as well as in integration with register allocation [5, 24]. Both [5] and [12] use a program representation that is similar to ours, but where the data are not explicitly represented as nodes. To the best of our knowledge, no previous attempt has been made in combining global code motion with instruction selection.

7 Conclusions and Future Work

This paper introduces a *universal* graph-based representation for programs and instructions that unifies control and data flow. From the representations a new constraint model for instruction selection is derived. The paper shows that the approach is more expressive and generates code of similar quality compared to LLVM as a state-of-the-art compiler. The constraint model is robust for small to medium-sized functions as well as expressive processor instructions and is competitive with LLVM in code quality.

One line of future work is to extend the model to address the limitations discussed in Sect. 4. Additional improvements include exploring more implied and dominance constraints and pre-solving techniques to increase the model's robustness. We intend to explore both larger input programs as well as more processor architectures with a more robust model.

Instruction selection is but one task in code generation. We will explore in detail how this paper's model can be integrated with a constraint model for register allocation and instruction scheduling introduced in [10, 11].

Acknowledgments. This research has been partially funded by LM Ericsson AB and the Swedish Research Council (VR 621-2011-6229). The authors are grateful for helpful discussions with Frej Drejhammar and Peter van Beek and for constructive feedback from the anonymous reviewers.

References

1. Opturion CPX user's guide: Version 1.0.2. Tech. rep., Opturion Pty Ltd (2013)
2. Aho, A.V., Ganapathi, M., Tjiang, S.W.K.: Code Generation Using Tree Matching and Dynamic Programming. Transactions on Programming Languages and Systems 11(4), 491–516 (1989)
3. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 177–189 (1983)
4. Arslan, M.A., Kuchcinski, K.: Instruction Selection and Scheduling for DSP Kernels on Custom Architectures. In: EUROMICRO Conference on Digital System Design (2013)
5. Barany, G., Krall, A.: Optimal and Heuristic Global Code Motion for Minimal Spilling. In: Compiler Construction, pp. 21–40 (2013)
6. Bashford, S., Leupers, R.: Constraint Driven Code Selection for Fixed-Point DSPs. In: ACM/IEEE Design Automation Conference. pp. 817–822 (1999)
7. Bednarski, A., Kessler, C.W.: Optimal Integrated VLIW Code Generation with Integer Linear Programming. In: International Euro-Par Conference, vol. 4128, pp. 461–472 (2006)
8. Bruno, J., Sethi, R.: Code Generation for a One-Register Machine. Journal of the ACM 23(3), 502–510 (1976)
9. Buchwald, S., Zwinkau, A.: Instruction Selection by Graph Transformation. In: International Conference on Compilers, Architectures and Synthesis for Embedded Systems. pp. 31–40 (2010)
10. Castañeda Lozano, R., Carlsson, M., Blindell, G.H., Schulte, C.: Combinatorial spill code optimization and ultimate coalescing. In: Kulkarni, P. (ed.) Languages, Compilers, Tools and Theory for Embedded Systems. pp. 23–32. ACM Press, Edinburgh, UK (Jun 2014)
11. Castañeda Lozano, R., Carlsson, M., Drejhammar, F., Schulte, C.: Constraint-based register allocation and instruction scheduling. In: Milano, M. (ed.) Eighteenth International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 7514, pp. 750–766. Springer-Verlag, Québec City, Canada (Oct 2012)
12. Click, C.: Global Code Motion/Global Value Numbering. In: ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. pp. 246–257 (1995)
13. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10), 1367–1372 (Oct 2004)
14. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM TOPLAS 13(4), 451–490 (Oct 1991)
15. Ebner, D., Brandner, F., Scholz, B., Krall, A., Wiedermann, P., Kadlec, A.: Generalized Instruction Selection Using SSA-Graphs. In: ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 31–40 (2008)
16. Eckstein, E., König, O., Scholz, B.: Code Instruction Selection Based on SSA-Graphs. In: International Workshop on Software and Compilers for Embedded Systems. pp. 49–65 (2003)

17. Ertl, M.A.: Optimal Code Selection in DAGs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 242–249 (1999)
18. Ertl, M.A., Casey, K., Gregg, D.: Fast and Flexible Instruction Selection with On-Demand Tree-Parsing Automata. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 52–60 (2006)
19. Floch, A., Wolinski, C., Kuchcinski, K.: Combined Scheduling and Instruction Selection for Processors with Reconfigurable Cell Fabric. In: International Conference on Application-Specific Systems, Architectures and Processors. pp. 167–174 (2010)
20. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman (1979)
21. Gebotys, C.H.: An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In: International Symposium on System Synthesis. pp. 41–47 (1997)
22. Glanville, R.S., Graham, S.L.: A New Method for Compiler Code Generation. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 231–254 (1978)
23. Hjort Blindell, G.: Survey on Instruction Selection: An Extensive and Modern Literature Study. Tech. Rep. KTH/ICT/ECS/R-13/17-SE, KTH Royal Institute of Technology, Sweden (Oct 2013)
24. Johnson, N., Mycroft, A.: Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In: International Conference of Compiler Construction. pp. 1–16 (2003)
25. Koes, D.R., Goldstein, S.C.: Near-Optimal Instruction Selection on DAGs. In: IEEE/ACM International Symposium on Code Generation and Optimization. pp. 45–54 (2008)
26. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: IEEE/ACM International Symposium on Code Generation and Optimization (2004)
27. Laurière, J.L.: A Language and a Program for Stating and Solving Combinatorial Problems. Artificial Intelligence 10(1), 29–127 (1978)
28. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In: IEEE MICRO-30. pp. 330–335 (1997)
29. Leupers, R.: Code Selection for Media Processors with SIMD Instructions. In: Conference on Design, Automation and Test in Europe. pp. 4–8 (2000)
30. Martin, K., Wolinski, C., Kuchcinski, K., Floch, A., Charot, F.: Constraint-Driven Instructions Selection and Application Scheduling in the DURASE System. In: International Conference on Application-Specific Systems, Architectures and Processors. pp. 145–152 (2009)
31. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Thirteenth International Conference on Principles and Practice of Constraint Programming. vol. 4741, pp. 529–543. Springer-Verlag (2007)
32. Pelegrí-Llopart, E., Graham, S.L.: Optimal Code Generation for Expression Trees: An Application of BURS Theory. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 294–308 (1988)
33. Sweetman, D.: See MIPS Run, Second Edition. Morgan Kaufmann (2006)
34. Tanaka, H., Kobayashi, S., Takeuchi, Y., Sakanushi, K., Imai, M.: A Code Selection Method for SIMD Processors with PACK Instructions. In: International Workshop on Software and Compilers for Embedded Systems. pp. 66–80

35. Živojnović, V., Martínez Velarde, J., Schläger, C., Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology. In: Conference on Signal Processing Applications and Technology. pp. 715–720 (1994)
36. Wilson, T., Grewal, G., Halley, B., Banerji, D.: An Integrated Approach to Retargetable Code Generation. In: International Symposium on High-Level Synthesis. pp. 70–75 (1994)