IMPROVED ISE IDENTIFICATION UNDER HARDWARE CONSTRAINT

Daniel Shapiro, Miodrag Bolic

School of Information Technology and Engineering
Computer Architecture Research Group, University of Ottawa
{ dshap092, mbolic } @site.uottawa.ca

ABSTRACT

The three Instruction Set Extension (ISE) enumeration algorithms described in this paper are Subgraph Enumeration (SE), Subgraph Removal (SR), and Lucky Subgraph Removal (LSR). SE exhaustively enumerates all convex subgraphs of a dataflow graph. SR iteratively finds the highest gain subgraph and then locks the related nodes out of the solution space for the next iteration of the search. Finally, LSR represents our tunable approach where both SE and SR are used to trade compiler execution time for solution quality in a hardware constrained design space. In this paper we present the mechanics behind these three ISE enumeration algorithms, and an instruction selection algorithm compatible with all three approaches.

Index Terms— Instruction set extension, instruction enumeration, instruction selection, configurable processor, ASIP, ISE

1. INTRODUCTION

Hardware/software codesign compilers are commonly used in the literature to perform research into ASIP design. A good ASIP compiler should have the capability to analyze legacy source code and representative execution profiles for the code, and then partition the design into hardware and software based upon the overall design constraints such as the available hardware area, and the bandwidth between the registers and the processor's execution stage. Fast compilation time is important for codesign compilers because a system designer will want to evaluate many different scenarios before arriving at a final design, and it is common practice to incrementally compile source code as it is being developed.

ISE identification is the overall algorithm for deciding upon an extended instruction set for an ASIP. ISE enumeration is the part of the ISE identification algorithm where candidate instructions are listed by exploring the program source code at the basic block level. From the enumerated list of ISEs, a subset must be selected for implementation. This is where the instruction selection algorithm comes into play. The instruction selection algorithm maximizes the speedup resulting for the selected ISEs while observing hardware size constrains for the ASIP.

As we will see later on, SE is an exhaustive search algorithm, SR is a greedy algorithm, and LSR is a parameterized combination of SE and SR with a depth cutoff for searching. Section 2 covers the ISE enumeration details for these approaches, and Section 3 presents the instruction selection algorithm. Section 4 sums up the presented algorithms.

2. ISE ENUMERATION

An ISE enumeration algorithm is used to list the candidate custom instructions (ISEs) which can be used to cover all, or part of, a basic block. Each basic block is one node in a (usually cyclic if loops are present and are not fully unrolled) control flow graph such as Fig. 1. A Directed Acyclic Graph (DAG) G(V,E) is used to represent the dataflow graph of a basic block as shown in Fig. 2. G is the graph, V represents the set of nodes and E represents the set of the data dependencies. Unresolved memory access nodes, and nodes involved in non-ALU operations are marked as "forbidden" so that they cannot be included as part of an ISE [1].

The user can specify at compile time which ISE enumeration approach to follow. The enumeration is achieved by SR, SE or LSR by repeatedly solving an Integer Linear Program (ILP) which models the design space. The enumeration of subgraphs is performed by maximizing the speedup in the model to identify a candidate ISE. Because of this, the ISEs are enumerated by SE, SR, and LSR from largest time saved to smallest. Note that in this context the time saved does not yet take into account the frequency of basic block execution. At the end of each ISE enumeration iteration an action must be taken to update the model for the next iteration of the design space exploration. If the user has specified the SE algorithm should be used, then the pattern of nodes of the identified ISE are disabled as a group, and if the user has specified that SR should be used then each node from the identified ISE is marked as forbidden. The process of executing and then updating the model is repeated until no valid solution is found. There is also a user accessible setting for deciding the maximum number of inputs and outputs allowed

Thanks to NSERC for funding this research.

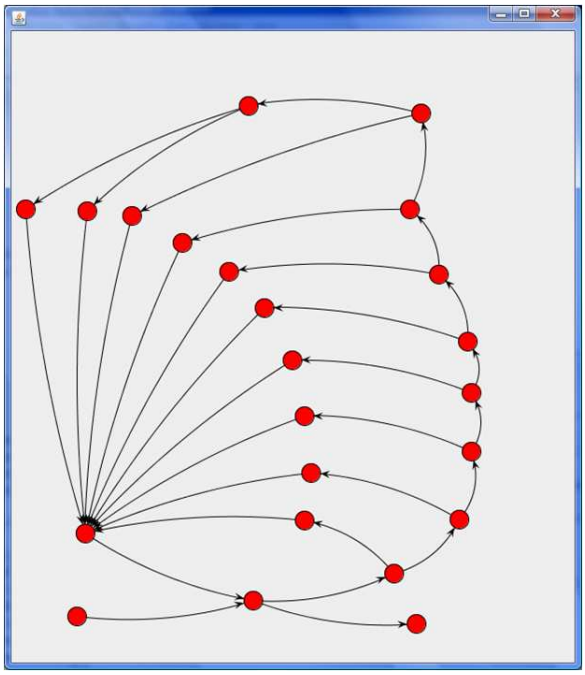

Fig. 1. Control flow graph example.

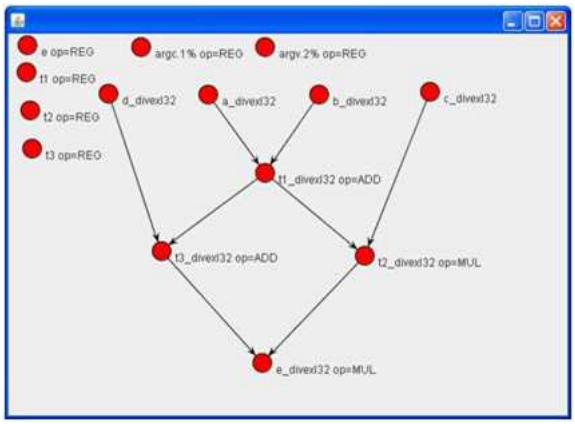

Fig. 2. Dataflow graph example.

in an ISE because that information is specific to the target processor, and higher I/O constraints are directly proportional to compiler execution time.

Algorithms 1, 2, and 3 below explain the ISE enumeration algorithm for SE, SR and LSR. All three algorithms begin with declarations of the variables that will be used. The constants $maxInputs$, $maxOutputs$, and $maxHW$ are the user specified I/O constraint and maximum available hardware respectively. For each algorithm a list is used to mark any constants so that they are not treated as inputs to a candidate ISE, and so that the parsing does not treat the node as a variable. The $candidate\_ISEs$ vector is used to store the list of candidate ISEs during the algorithm execution.

In Algorithm 1 each identified candidate ISE is disabled as a pattern. Any ISE added to the $disabled\_ISEs$ vector on line 11 of Algorithm 1 is disabled from being enumerated again by the ILP model. In Algorithm 2 the list $forbidden\_nodes$ on line 11 is used to track the list of nodes which cannot or should not be used as part of an identified ISE. For example, memory accesses cannot be implemented in an ISE using our models, whereas nodes of an identified ISE should not be used in additional ISEs when the SR algorithm is followed.

Algorithm 1 ISE Enumeration using SE

1: $maxInputs, maxOutputs, maxHW$ are fixed design space constraints set at compile time
2: $blkHashCode \leftarrow$ Unique Basic Block ID
3: Build dataflow graph $G < V, E >$ from basic block
4: Note the operation on each node, the latency of the node in HW and SW, the hardware size of the node
5: Note when an operand is a constant
6: $disabled\_ISEs = \{\emptyset\}$;
7: **repeat**
8: $WriteModelToFile(arcs, nodes, forbidden\_nodes, disabled\_ISEs, maxInputs, maxOutputs, maxHW, latency\_of\_node\_SW, latency\_of\_node\_HW, is\_constant, hw\_size)$
9: Execute external solver $SOLVE(blkHashCode, nodes)$
10: NEW_ISE = Read in solution for identified ISE
11: $disabled\_ISEs.add(NEW\_ISE)$
12: $candidateISEs.add(NEW\_ISE)$
13: **until** (No ISE was found)
14: **return** $candidateISEs$

Algorithm 2 ISE Enumeration using SR

1: $maxInputs, maxOutputs, maxHW$ are fixed design space constraints set at compile time
2: $blkHashCode \leftarrow$ Unique Basic Block ID
3: Build dataflow graph $G < V, E >$ from basic block
4: Note the operation on each node, the latency of the node in HW and SW, the hardware size of the node
5: Note when an operand is a constant
6: $forbidden\_nodes = \{\emptyset\}$;
7: **repeat**
8: $WriteModelToFile(arcs, nodes, forbidden\_nodes, maxInputs, maxOutputs, maxHW, latency\_of\_node\_SW, latency\_of\_node\_HW, is\_constant, hw\_size)$
9: Execute external solver $SOLVE(blkHashCode, nodes)$
10: NEW_ISE = Read in solution for identified ISE
11: $forbidden\_nodes = forbidden\_nodes \cup (NEW\_ISE \rightarrow nodes)$
12: $candidateISEs.add(NEW\_ISE)$
13: **until** (No ISE was found)
14: **return** $candidateISEs$

In Algorithm 3, LSR_SIZE indicates the number of SE iterations to perform during LSR. In other words we will run the algorithms as LSR(LSR_SIZE). When performing an SE step of LSR, the candidate ISE is added to the SE buffer for later use and the pattern is disabled from being identified again. We verify that each disabled ISE does not consist completely of forbidden nodes. For any case where a disabled ISE consists entirely of forbidden nodes, we remove that ISE from the list of disabled ISEs because it will be automatically disabled by the fact that all of its nodes are forbidden (lines 35-39). When switching from an SR step back to an SE step, the SE buffer must be checked to verify that each identified

ISE was not in fact a step in the SR algorithm. Specifically the buffer must be checked in order from oldest to newest ISE (lines 14-20), ensuring that all nodes of the candidate ISE are either mutually exclusive with the forbidden nodes (therefore really an SR step), or not mutually exclusive with the set of disabled nodes. Finally, if the nodes of the ISE from the SR step are still mutually exclusive then its nodes are disabled, and otherwise it is added to the SE buffer (lines 22-27). There are many fewer ISEs enumerated when the nodes of each identified ISE are collapsed to a single forbidden node because the design space exploration is cut short. LSR enumerates far more ISEs, but not nearly as many as subgraph enumeration.

Algorithm 3 ISE Enumeration using LSR

1: $LSR\_SIZE, maxInputs, maxOutputs, maxHW$ are fixed design space constraints set at compile time
2: $blkHashCode \leftarrow$ Unique Basic Block ID
3: Build dataflow graph $G < V, E >$ from basic block
4: Note the operation on each node, the latency of the node in HW and SW, the hardware size of the node
5: Note when an operand is a constant
6: $forbidden\_nodes = \{\emptyset\}$; $disabled\_ISEs = \{\emptyset\}$; $SE\_Buffer = \{\emptyset\}$
7: $iterations = 0$
8: **repeat**
9: $WriteModelToFile(arcs, nodes, forbidden\_nodes, disabled\_ISEs, maxInputs, maxOutputs, maxHW, latency\_of\_node\_SW, latency\_of\_node\_HW, is\_constant, hw\_size)$
10: Execute external solver $SOLVE(blkHashCode, nodes)$
11: NEW_ISE = Read in solution for identified ISE
12: **if** $((iterations\%LSR\_SIZE) == 1)$ **then**
13: **if** $SE\_Buffer.size() > 0$ **then**
14: **for** $(\forall ise \in SE\_Buffer)$ **do**
15: **if** $\nexists node \in (ise \rightarrow nodes), node \in forbidden\_nodes$ **then**
16: $SE\_Buffer.remove(ise)$
17: $disabled\_ISEs.remove(ise)$
18: $forbidden\_nodes = forbidden\_nodes \cup (ise \rightarrow nodes)$
19: **end if**
20: **end for**
21: clear SE_Buffer
22: **if** $(\nexists node \in (NEW\_ISE \rightarrow nodes), node \in forbidden\_nodes)$ **then**
23: $forbidden\_nodes = forbidden\_nodes \cup (NEW\_ISE \rightarrow nodes)$
24: **else**
25: $SE\_Buffer.add(NEW\_ISE)$
26: $disabled\_ISEs.add(NEW\_ISE)$
27: **end if**
28: **else**
29: $forbidden\_nodes = forbidden\_nodes \cup (NEW\_ISE \rightarrow nodes)$
30: **end if**
31: **else**
32: $SE\_Buffer.add(NEW\_ISE)$
33: $disabled\_ISEs.add(NEW\_ISE)$
34: **end if**
35: **for** $(ise \in disabled\_ISEs)$ **do**
36: **if** $(forbidden\_nodes \cup (ise \rightarrow nodes)) \equiv forbidden\_nodes$ **then**
37: $disabled\_ISEs.remove(ise)$
38: **end if**
39: **end for**
40: $candidateISEs.add(NEW\_ISE)$
41: $iterations++$
42: **until** (No ISE was found)
43: **return** $candidateISEs$

The LINGO and lp_solve math programming languages were used to model the instruction enumeration and selection problems [2], [3]. Note that all three algorithms are deterministic and produce the same result for a given program no matter how many times the algorithm is tested. Such consistent behavior is important for a compiler pass.

In the ILP model for each algorithm the set of binary variables x_i contains one variable for each node in the dataflow graph. The nodes of an identified ISE are marked by the corresponding x_i being set to 1. Once an ISE has been identified it must be added to a list of ISEs that will be handed off to the instruction selection phase for ranking and selection. However, in the more immediate term, the current ISE is used to get the next best ISE.

For SE, every already identified ISE has two sets of binary variables created for it in the model. First, a set for the first ISE found will be called $ISE0$, for the second identified ISE it will be called $ISE1$, and so on. Each of these sets is defined alongside a set DIF_{ji} that acts as a binary counter of differences between the identified ISE and the current candidate ISE. If 10 ISE patterns have been disabled, then $DIF0$, $DIF1$, ..., $DIF9$, $ISE0$, $ISE1$, ..., $ISE9$ will be declared as sets of binary variables in the model. Each set has as many elements as there are nodes in the dataflow graph. As long as there is one difference in the x_i values of each disabled ISE pattern and the candidate ISE, the candidate ISE can be the solution for the current iteration. Otherwise it is not a valid solution. The graph patterns of two ISEs can be exactly the same but the nodes forming the ISEs cannot be the same exact nodes. We define this formally as follows:

I_1: indicies for nodes in a basic block
I_2: indicies for ISEs already enumerated

$$x_i \in \{0,1\}, i \in I_1 \quad (1)$$

$$x_i - ISE_{ji} \leq DIF_{ji}, i \in I_1, j \in I_2 \quad (2)$$

$$ISE_{ji} - x_i \leq DIF_{ji}, i \in I_1, j \in I_2 \quad (3)$$

$$ISE_{ji} + x_i \geq DIF_{ji}, i \in I_1, j \in I_2 \quad (4)$$

$$2 - ISE_{ji} - x_i \geq DIF_{ji}, i \in I_1, j \in I_2 \quad (5)$$

$$\sum_{i \in I_1} DIF_{ji} \geq 1, \forall j \in I_2 \quad (6)$$

The five constraints above are all required in order to disable the nodes of an enumerated ISE in the graph. Each DIF_{ji} set is forced to contain a 1 if there is an index which differs from the x_i set. In the first two constraints above an index of the DIF_{ji} set is forced to 1 if x_i and ISE_{ji} are not the same. In the third and fourth constraint a DIF_{ji} element with index "I" is forced to 0 if x_i and ISE_{ji} are the same. In the final constraint, the number of differences between the ISE_{ji} and x_i sets must be greater than 1. Therefore the sum of these differences represented by DIF_{ji} must be greater than or equal to 1. The truth table for these rules is presented

below. We can see that the DIF_{ji} variable counts differences in node selection between an identified ISE and the current candidate ISE.

Table 1. Truth table for the rules which compare disabled patterns to the candidate ISE

x_i	ISE_{ji}	DIF_{ji}
0	0	0
0	1	1
1	0	1
1	1	0

2.1. ISE Enumeration Examples: SR and LSR

Now that we have seen some detail on the ISE enumeration algorithms, let us look at a simplified example without considering the actual graph shape. In Table 3 we see a sequence of steps by SR to identify ISEs and then remove the associated nodes from a basic block. In Table 4 we see the same basic block is processed by LSR.

Steps 1, 2, 5, and 7 of LSR (Table 2) find the same ISEs as SR, and the ISEs are found in the same order. LSR also finds ISEs in Steps 3, 4, and 6 which overlap previously identified ISEs not considered by SR. The SE steps find ISEs which require a check for mutual exclusivity, which occurs during a switch from SE to SR. This check is first performed in Step 4, after the fourth ISE is found. LSR checks the list of disabled ISEs (resulting from Steps 2 and 3), and finds that the ISE from Step 2 (n1, n2, n3) is mutually exclusive from the list of forbidden nodes. LSR adds these nodes (n1, n2, n3) to the list of forbidden nodes, and evaluates the next ISE in the list of disabled ISEs. The instruction (n1, n2, n3, n4) overlaps the new list of forbidden nodes, and remains disabled. After all instructions in the SE buffer have been checked, the buffer is flushed. The ISE found in Step 4 (n1, n2, n4, n20) now requires a check, as new nodes were forbidden during the check of ISEs found during the SE steps (n1, n2, n3). The check finds that ISE (n1, n2, n4, n20) is no longer mutually exclusive from the list of forbidden nodes as it overlaps nodes n1 and n2. LSR adds this ISE to the SE buffer and proceeds to step 5.

Steps 5 and 6 both find ISEs, and add them to the SE buffer, and to the list of disabled ISEs. At the end of Step 7, the ISEs found in Steps 5 and 6 are checked, and ISE (n4, n17, n18, n19, n20) is found to be mutually exclusive. Those nodes are added to the list of forbidden nodes. As both (n1, n2, n3, n4) and (n1, n2, n4, n20) now have all their nodes in the list of forbidden nodes, they can be removed from the list of disabled ISEs as there is no chance that they will be implemented. This ensures that the number of ISEs needed to be checked against in each SE step is small, which decreases the amount of time needed to find an ISE. The check then looks at ISE (n14, n15, n17, n18, n19, n20), and noting that some of its nodes are not forbidden, keeps it in the list of disabled ISEs. Finally, it checks ISE (n8, n9) as new nodes were added to the list of forbidden nodes. As (n8, n9) is mutually exclusive, its nodes are forbidden, and the algorithm exits.

For simplicity assume that this problem (basic block) represents the entire program, and the block is executed only once. We observe that the SR approach resulted in 8+6+5+1=20 cycles saved. Meanwhile the LSR algorithm resulted in the same answer of 20 saved cycles. This is because LSR only does better than SR when a hardware constraint is present. Consider the small example shown in Fig. 3. Assume for this example that all variables are integers, the area constraint is 3200 LEs, and the I/O constraint is (6,3). The hardware sizes and latencies of the nodes are assumed as listed in Table 2 for this example only. For this example we assume that software and hardware latencies are the same, and that the speedup is obtained through parallel implementation of the hardware blocks.

Table 2. Assumed hardware sizes and latencies for example shown in Fig. 3

Instruction	Hardware size (LEs)	Latency (cycles)
BXOR	32	1
SUB	49	2
MUL	1598	3

r1 = x1 $\oplus$ x2;
r2 = x2 $\oplus$ x3;
r3 = x3 - x4;
r4 = x5 - x6;
r5 = x6 * x7;
r6 = x7 * x8;

Fig. 3. Simple motivating example program. All variables are integer. It is assumed that all hardware and software latencies are equal for this list of instructions.

If SR is selected as the ISE enumeration algorithm, then the ISEs enumerated by SR will be first (r4,r3,r2,r1) which saves 4 cycles, and second (r5,r6) which saves 3 cycles. In the instruction selection pass of the ISE identification algorithm the ISE (r4,r3,r2,r1) will be selected because it has the highest gain among the various alternatives and it does not exceed the hardware area constraint. However, if LSR is selected as the ISE enumeration algorithm, then several additional instructions can be enumerated, as follows: first (r4,r3,r2,r1) will be enumerated, and then (r5,r6). Next the enumeration algorithm will find the ISE (r6,r4,r2) and then (r5,r3,r1), both of which which overlap the first two ISEs enumerated and save 3 cycles. In the instruction selection pass of the ISE identifi-

cation algorithm, the ISEs (r6,r4,r2) and (r5,r3,r1), which are equivalent, are selected because they save 6 cycles. For this example the application of SR saved 4 cycles whereas LSR saved 6 cycles.

3. INSTRUCTION SELECTION

The instruction selection algorithm is in fact an ILP model which must select a subset of the candidate ISEs for implementation. The selection of ISEs must maximize the speedup of the application without exceeding the limit on the amount of the available hardware or the I/O constraint. This problem can be seen as a knapsack problem and it is programmed in that style. The designed chip can be seen as a knapsack of limited size and the candidate ISEs as goods which we would like to include in the knapsack. Since we may not be able to include all instructions in the chip, we attempt to add the best set of instructions that will fit.

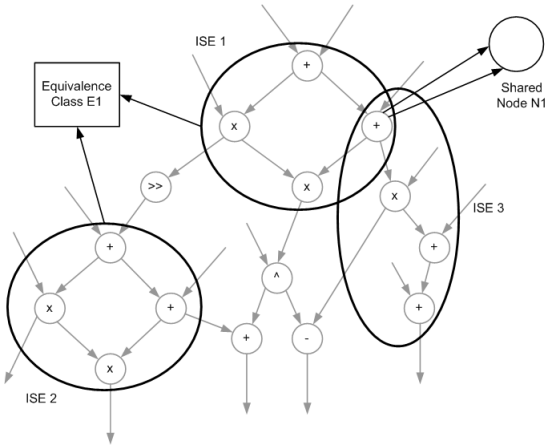

Fig. 4. Association of ISEs to nodes and classes

Graph isomorphism is performed on the list of identified ISEs and each equivalent ISE is added to an equivalence class. As shown in Fig. 4, equivalent ISEs $ISE1$ and $ISE2$ are associated to the equivalence class E1. In Fig. 4 note that the same output can be used multiple times as input to other nodes. Two arcs coming from a node does not imply that the node has two outputs, but instead that a single output is being shared by two other nodes.

Equivalence classes have two metrics that characterize them: IS_USED and $EQUIV\_HWCOST$. IS_USED is non-zero when any member of the equivalence class is selected, and $EQUIV\_HWCOST$ represents the cost of implementing the equivalent ISE in terms of hardware area. Information on equivalence classes and shared nodes such as the node $N1$ shared between $ISE1$ and $ISE3$ is used by the instruction selection ILP model.

Our instruction selection model is defined by the 7 equations below. Equation 13 is the model objective while Equations 7 through 12 are constraints. The letters I and J are used to index into arrays of variables. ISEs are associated to equivalence classes using the association set $EQUIV(ISE, EQUIV\_CLASS)$. ISEs are associated with nodes via the association set $ISE\_USES(ISE, NODE)$. In Fig. 4 the ISEs $ISE1$ and $ISE3$ are mutually exclusive because they both contain the node $N1$, and therefore they are both associated with that node $N1$ in the instruction selection model. The model will select either $ISE1$ or $ISE3$ but not both. Only nodes that are associated with multiple ISEs are included in the model in order to improve the runtime of the model. Each node that is included in the model can be used by only one ISE. Although several ISEs may share a node during the instruction enumeration, two overlapping instructions could not be scheduled after instruction selection. Therefore each node in the model has a variable $TIMES\_USED$ which tracks the number of times that a node is used by a selected ISE. Of course each node cannot have $TIMES\_USED$ greater than 1, and each ISE can only use a node once. These constraints are encoded into the instruction selection model. Out of all of the ISEs capable of using a given node in it's datapath, only one of them can succeed in being selected. Equations 7 and 8 represent these ideas in the model.

E_{ic}: indicies for $EQUIV_{ic}$ where i is a candidate ISE and c is an equivalence class
I_2: indices for candidate ISEs
N_1: indices for nodes which are part of more than one candidate ISE
C_1: indices for equavenance classes

$$TIMES\_USED_k = \sum_{l \in N_1} (INCLUDE_{kl})), k \in I_2 \quad (7)$$

$$TIMES\_USED_k \leq 1, k \in I_2 \quad (8)$$

$$UNLISTED\_COST = \sum_{i \in I_2} (INCLUDE_i \cdot HWCOST_i \cdot UNLISTED_i), \quad (9)$$

$$LISTED\_COST = \sum_{c \in C_1} (IS\_USED_c \cdot EQUIV\_HWCOST_c) \quad (10)$$

$$IS\_USED_c \geq INCLUDE_i, \{i, c\} \in E_{ic} \quad (11)$$

$$UNLISTED\_COST + LISTED\_COST \leq KNAPSACK\_CAPACITY \quad (12)$$

$$MAX = \sum_{i \in I_2} (SAVEDCYCLES_i \cdot INCLUDE_i) \quad (13)$$

Every ISE is modeled in the instruction selection ILP with four variables: $INCLUDE$, $HWCOST$, $SAVEDCYCLES$, and $UNLISTED$. Each element of the $INCLUDE$ set is a binary variable used to decide if the ISE is selected for implementation. $HWCOST$ is the projected area cost of implementing the ISE. For an ISE that is part of an equivalence class, $HWCOST$ is always equal to $EQUIV\_HWCOST$. We can see in Fig. 4 that the $EQUIV\_HWCOST$ is available via an association described in Equation 10 which pays once for hardware cost

of the entire equivalence class. $UNLISTED$ is set to 0 for all equivalent ISEs, and set to 1 for all non-equivalent ISEs. Therefore $LISTED\_COST$ is equal to the cost of the nodes contained in selected isomorphic ISEs as shown in Equation 12, and $UNLISTED\_COST$ is equal to the cost of the nodes contained in selected non-isomorphic ISEs. $SAVEDCYCLES$ is the projected time saved by selecting an ISE.

When exploring the design space we may encounter equivalent custom instructions. Graph isomorphism is the problem of proving that two graphs are topologically equivalent and functionally equivalent. In ISE identification it is important to know when two ISEs are equivalent so that in such a case they are not both implemented as hardware, wasting area on the chip. Of course there is no need to implement two identical ISEs because they cannot execute in parallel when implemented into the processor, and therefore only one copy of the instruction is needed. One may make the mistake of thinking that an implementation of two copies of an instruction in parallel must be considered as part of the design space exploration at the instruction selection stage, and therefore modeled in the graph isomorphism testing. In fact, the instruction enumeration stage has already considered this problem, and parallel implementations of a given ISE may show up as another ISE on its own during the instruction enumeration stage. Instead of implementing several copies of the same ISE, we should express in the instruction selection model the possibility that one instruction implemented in hardware can be used to execute two equivalent graphs. The graph isomorphism problem is known to be very hard to solve, but there are tools such as nauty and JGraphT which can be used to perform graph isomorphism testing [4],[5]. We selected JGraphT because it is Java based and was easily customized to perform directed labeled graph isomorphism testing. We also used JGraphT to print graphical representations of ISEs.

Let us now consider the calculation of the hardware cost of equivalent ISEs. We may not want all possible members of an equivalence class to be targeted for execution on one shared ISE hardware unit. Even though implementing one instance of the equivalence class creates hardware for executing all of the members of that class, it may be advantageous to execute some members of the class as part of other ISEs. Put another way, selecting all members of an equivalence class of ISEs locks down the associated nodes that are in conflict with other ISEs, preventing the selection of ISEs overlapping a node in any member of the equivalence class. An example would be a case where the ith instance of ISE equivalence class EQ1 shares resources with a highly beneficial ISE . By completely implementing all instances of EQ1, the highly beneficial ISE is no longer implementable (as the nodes shared with the ith instances are forbidden). Instead, it is more beneficial to implement all instances of EQ1 except for the ith instance, allowing the highly beneficial ISE to be implemented as well. Therefore we give the model a chance to weigh the benefit of selecting each equivalent copy of the ISE with all other selectable ISEs. The number of uses of an equivalent ISE does not matter in terms of implementation cost as long as it is used at least once. This is due to the fact that only one instruction is implemented. The cost of implementing equivalent ISEs is defined in Equations 9 and 10.

We use the $LISTED\_COST$ and $UNLISTED\_COST$ to constrain the total cost in hardware area of the selected instructions. The instruction selection model accepts a user parameter to define the available hardware area for instruction implementation we call $KNAPSACK\_CAPACITY$ as defined in Equation 12. Equation 13 represents the objective of the model.

3.1. Hardware Size Assumptions

Since LSR can only be more useful than SR when there is a binding hardware size contraint, it is important to mention here the hardware size assumptions made in both the instruction enumeration and the instruction selection algorithms. We define hardware area in an abstract way by using integers to represent hardware size in LEs. Hardware size assumptions for each instruction are presented in Table 5. There are a few improvements possible for the information in Tables 6 and 5 which were not attempted. First, MOD may be replaced by wire mapping in the case where the modulus is a base 2 constant. Second, barrel_shifter may be replaced when shifting by wire mapping in case where the shift is a constant number of bits. In this case the hardware size cost and the execution time should both be counted as 0. Third, some instructions in Tables 6 and 5 were not implemented, and instead were assumed to have the same hardware size and latency as a similar component. For example the signed right shift RSHS was assumed to have the same hardware size and latency as the unsigned right shift RSHU.

4. CONCLUSIONS

ISE enumeration algorithms were presented for SE (an exhaustive algorithm), SR (a greedy algorithm) and LSR (a new configurable algorithm which can trade compiler execution time for solution quality when a hardware constraint is binding in the design space). After presenting ISE enumeration algorithm details, examples of SR and LSR in action were provided in order to give the reader a feel for how these algorithms operate, and to show how LSR can exceed the speedup of SR under hardware constrained conditions. We proceeded to present an ISE selection algorithm which can handle isomorphic ISEs.

5. REFERENCES

[1] K. Atasu, G. Dundar, and C. Ozturan, "An integer linear programming approach for identifying instruction-set extensions," in *Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2005, pp. 172–177.

[2] Linus Schrage, *Optimization Modeling with LINGO*, Lindo Systems Inc., Chicago, IL., 6th edition, 2006.

[3] "lp_solve," http://sourceforge.net/projects/lpsolve.

[4] Brendan D. McKay, "Nauty," Australia National University, Canberra, Australia.

[5] Barak Naveh, "Jgrapht," El Granada, CA, The Eigenbase Project.

Table 3. Example SR solution

Iteration	Graph nodes in identified ISE	Time saved by using this ISE	Nodes forbidden during this iteration
1	(n5, n6, n7, n11, n12, n13)	8	-
2	(n1, n2, n3)	6	(n5, n6, n7, n11, n12, n13)
3	(n4, n17, n18, n19, n20)	5	(n1, n2, n3, n5, n6, n7, n11, n12, n13)
4	(n8, n9)	1	(n1, n2, n3, n4, n5, n6, n7, n11, n12, n13, n17, n18, n19, n20)
5	-	0	(n1, n2, n3, n4, n5, n6, n7, n8, n9, n11, n12, n13, n17, n18, n19, n20)

Table 4. Example LSR solution

Iteration	Step type	Graph nodes in identified ISE	Time saved by using this ISE	Nodes forbidden during this iteration	ISEs disabled during this iteration
1	SR	(n5, n6, n7, n11, n12, n13)	8	-	(n5, n6, n7, n11, n12, n13)
2	SE	(n1, n2, n3)	6	(n5, n6, n7, n11, n12, n13)	-
3	SE	(n1, n2, n3, n4)	5	(n5, n6, n7, n11, n12, n13)	(n1, n2, n3)
4	SR	(n1, n2, n4, n20)	5	(n5, n6, n7, n11, n12, n13)	(n1, n2, n3) (n1, n2, n3, n4)
5	SE	(n4, n17, n18, n19, n20)	5	(n1, n2, n3, n5, n6, n7, n11, n12, n13)	(n1, n2, n3, n4), (n1, n2, n4, n20)
6	SE	(n14, n15, n17, n18, n19, n20)	5	(n1, n2, n3, n5, n6, n7, n11, n12, n13)	(n1, n2, n3, n4) (n1, n2, n4, n20) (n4, n17, n18, n19, n20)
7	SR	(n8, n9)	1	(n1, n2, n3, n5, n6, n7, n11, n12, n13)	(n1, n2, n3, n4) (n1, n2, n4, n20), (n4, n17, n18, n19, n20) (n14, n15, n17, n18, n19, n20)
8	SE	-	0	(n1, n2, n3, n4, n5, n6, n7, n8, n9, n11, n12, n13, n17, n18, n19, n20)	(n14, n15, n17, n18, n19, n20)

Table 5. Hardware size assumptions for 32-bit floating point operations.

Instruction	Hardware Size (LEs)	Latency (cycles)	Altera Component Name	Assumptions
ADD	975	14	altfp_add_sub	
CONVFI	457	1	altfp_convert	
CONVFS	457	1	altfp_convert	
CONVFT	689	1	altfp_convert	assumed 64 to 32-bit
CONVFX	84	1	altfp_convert	assumed 32 to 64-bit
CONVSF	342	1	altfp_convert	
CONVUF	342	1	altfp_convert	assumed values
DIVS	5941	33	altfp_div	
DIVU	5941	33	altfp_div	assumed values
FLOATCONST	0	0	-	
INTCONST	0	0	-	
MUL	926	5	altfp_mult	
NEG	2897	20	altfp_inv	
SUB	975	14	altfp_add_sub	
TSTEQ	58	3	altfp_compare	
TSTGES	97	3	altfp_compare	
TSTGEU	97	3	altfp_compare	assumed values
TSTGTS	98	3	altfp_compare	
TSTGTU	98	3	altfp_compare	assumed values
TSTLES	97	3	altfp_compare	
TSTLEU	97	3	altfp_compare	assumed values
TSTLTS	97	3	altfp_compare	
TSTLTU	97	3	altfp_compare	assumed values
TSTNE	58	3	altfp_compare	
ZEROS	0	0	-	

Table 6. Hardware size assumptions for 32-bit integer operations. Entries in the latency column marked with a * were counted as 1 cycle for software execution and 0 cycles for hardware implementation.

Instruction	Hardware Size (LEs)	Latency (cycles)	Altera Component Name	Assumptions
ADD	49	1	lpm_add_sub	
BAND	32	0*	-	
BNOT	0	0*	-	it appears that LUTs of input LEs are adjusted to perform the binary invert without the need for LEs
BOR	32	0*	lpm_or	
BXOR	32	0*	lpm_xor	
CONVFI	457	1	altfp_convert	
CONVFS	457	1	altfp_convert	
CONVIT	342	1	altfp_convert	assumed values
CONVSF	342	1	altfp_convert	
CONVSX	0	0*	-	sign extend does not require LEs
CONVUF	342	1	altfp_convert	assumed values
CONVZX	0	0*	-	zero extend does not require LEs
DIVS	1343	7	lpm_divide	
DIVU	1550	7	lpm_divide	
INTCONST	0	0	-	
LSHS	384	1		assumed values
LSHU	384	1	barrel_shifter	
MODS	1426	0*	MODS	
MODU	1114	0*	MODU	
MUL	1598	7	lpm_mult	
NEG	50	1	lpm_add_sub	subtraction with 0 as first input
RSHS	384	1		assumed values
RSHU	384	1	barrel_shifter	
SUB	53	1	lpm_add_sub	
SUBREG	0	0*	-	routing wires does not require LEs
TSTEQ	21	0*	lpm_compare	
TSTGES	54	0*	lpm_compare	
TSTGEU	54	0*	lpm_compare	
TSTGTS	32	0*	lpm_compare	
TSTGTU	32	0*	lpm_compare	
TSTLES	54	0*	lpm_compare	
TSTLEU	54	0*	lpm_compare	
TSTLTS	32	0*	lpm_compare	
TSTLTU	32	0*	lpm_compare	
TSTNE	21	0*	lpm_compare	
ZEROS	0	0	-	