Computer Architecture Research Group: carg.eecs.uottawa.ca
University of Ottawa

Tunable Instruction Set Extension Identification

Daniel Shapiro · Michael Montcalm · Jonathan Parri · Miodrag Bolic

Abstract In this work a tunable algorithm for instruction set extension identification is presented. The goal is to find a set of extensions to the instruction set which reduces application execution time. This novel approach enables the user to trade application speedup for compiler execution time. This approach shows benefit only when a binding hardware area constraint is applied to the design space. Several experiments are presented. An average improvement in application speedup of 4.5% over the state of the art approach was observed, and some instances of our approach were as much as 25.8% better than the state of the art. The aforementioned results are merely samples in a large design space, and we conclude that our novel algorithm can provide valuable advantages over state of the art approaches.

Keywords Instruction Set Extension · Design Space Exploration · ASIP

1 Introduction

A custom processor, called an Application Specific Instruction-set Processor (ASIP), can be designed using automated tools or by careful analysis of the target application. Some ASIPs such as Tensilica's Xtensa can be configured and extended automatically by a compiler toolchain [1]. Currently, Instruction Set Extension (ISE) identification algorithms may be performed off-line by a third-party such as Tensilica.

In order to design high performance embedded processors, efficient algorithms are needed for the identification of extensions to processor instruction sets. In this paper, we investigate improvements to one such technique, called

The authors are with the School of Information Technology and Engineering,
University of Ottawa, Ottawa, ON, Canada.
Tel.: +1-613-562-5800 x 2192
Fax: +1-613-691-1169
E-mail: {dshapiro, mmontcalm, jparri, mbolic}@uottawa.ca

ISE identification, which is used to discover the ISEs for an ASIP by analyzing the software that will be executed. Another way of looking at ISE identification is through the lens of hardware/software partitioning, where the instruction set of the ASIP is the hardware. The result of ISE identification is an instruction set for an ASIP, and a compiled program that can utilize this new customized processor. The partition is based upon analysis of the program that will be executed on the ASIP and profiling information.

We have implemented in the COINS compiler two ISE enumeration algorithms as Integer Linear Programs (ILPs). The first is called Subgraph Removal (SR), and follows the approach of [2]. The second is called Subgraph Enumeration (SE) and follows the approach of [3] but with graph isomorphism testing [4]. The SE implementation in this article replaces the tree traversal approach in [3], so that all models can execute in a common framework. The novel algorithm in this work is called Lucky Subgraph Removal (LSR), and combines the advantages of both SE and SR in order to reach a solution. SR is ruling-out patterns in flow graphs based on forbidden nodes, i.e., all graph patterns containing a forbidden node are rejected, while SE relies on forbidden (disabled) graph patterns, i.e., all graph patterns that match one of the forbidden patterns are rejected. The LSR approach uses both, forbidden nodes and forbidden patterns. However, the set of forbidden nodes is not updated after every iteration.

Our instruction selection model is flexible enough to understand each of the different enumeration algorithms that may be used, and so only one instruction selection model is needed regardless of the enumeration algorithm specified. A basic block is a sequence of instructions in a program without branch or jump statements. Basic blocks are the nodes in a control-flow graph representation of a program. The algorithms in this paper explore the dataflow graphs of basic blocks, and does not consider memory accesses for inclusion in custom instructions.

We have developed an algorithm (LSR) which improves the application speedup of designs constrained by a limit on the amount of hardware in the embedded system. The hardware constraint is expressed in terms of Logic Elements (LEs) found in an FPGA. A further contribution of this work is the tunable nature of our ISE enumeration algorithm which allows the user to specify the amount of design space exploration to perform. This novel approach enables the user to trade application speedup for compiler execution time. The presented algorithm contains an instruction selection component which can explore the trade-off between the enumerated ISEs. The algorithm can also understand isomorphic custom instructions, and instruction reuse. Given various hardware size constraints and basic block sizes, the speedup and compilation time of optimal (SE), greedy (SR) and tunable (LSR) ISE identification algorithms were characterized.

1.1 ISE Enumeration Algorithms

Although there are many possible stochastic solutions for identifying processor extensions, we focus on a deterministic and model-based approach. The three ISE enumeration algorithms described in this paper are SR, SE, and LSR. We use SE and SR as analogies for existing approaches [3], [5], and LSR represents our tunable approach. SE is an exhaustive search algorithms that produces a list of all valid ISEs for the dataflow graph of a basic block. The listed ISEs may overlap on the dataflow graph, or conflict with one another, as two or more enumerated ISEs may require the same graph nodes. Similar to [6], our ILP models avoid cyclic dependencies within ISEs using convexity constraints, whereas inter-ISE cyclic dependencies are not prevented.

SR enumerates only mutually exclusive ISEs by finding the ISE with the largest difference in hardware and software execution times (the gain), and then marks the associated nodes as forbidden, preventing them from being included in any subsequently listed ISEs. This process of finding the ISE with the highest gain and marking the nodes as forbidden is repeated until no more valid ISEs are found within the basic block. SR is called "most profitable subgraph first" in [5].

In the case of LSR, SE, and SR, the guide function for design space exploration formulated as a cost function in an ILP model, and each solution to the model produces one ISE. Between iterations of the ISE enumeration algorithm, where one ISE is found, the basic block model is modified to account for the most recently identified ISE. Next, the model is executed again to produce another ISE, or the stopping condition occurs where no solution was found.

The selected ISE enumeration algorithm is specified at compile time. Various ISE identification algorithms can be added to a compiler which accepts a design space specification (e.g. hardware area constraint, register bank I/O constraints), an execution profile, and source code as inputs. The maximum bandwidth of the register file expressed as the number of inputs and outputs are the I/O constraints. The compiler generates a customized embedded system composed of hardware and software as its output. The tunable algorithm (LSR) described in this paper can perform a deeper search than prior tractable approaches, and prune the search space at a specified depth. The instruction selection model used for this paper understands the selection of graph isomorphic ISEs, but does not support the use of ISEs that are only partially isomorphic. The implication is that hardware is not shared between the instructions added to the ASIP. We look for isomorphic ISEs because the hardware needed to implement all isomorphic instances of an ISE is only the hardware required by one instance of that ISE.

In the following sections we introduce our work in more detail. We begin with Section 2 where some of the prior art related to this work is presented. The key concept behind our ISE identification algorithm is explained in Section 3 by way of an example.

Section 4 contains observations about the performance of our LSR approach as compared to SE and SR. Experiments are described and results are compared to the prior art using data sets and benchmarks. Our conclusions and a discussion of future work are contained in Section 5.

2 Related Work

In this section, concepts in ISE identification are reviewed. First a general introduction to existing approaches is presented, followed by an examination of the limitations of ISE identification. Next, the flexibility of various approaches is examined in terms of generality and legacy code. Some approaches, such as our own, are able to enumerate ISEs which are mutually exclusive on the program dataflow graph. We review the literature relating to this concept. Also discussed is the impact of an area constraint on the model of the design space, and the impacts of memory hierarchy, loop handling, and compiler optimizations on the compilation time and speedup of the ISE identification algorithm. We conclude with an analysis of the compiler execution time.

ISE identification has several steps. First, a profile is generated for a program. Next, a set of candidate instructions is enumerated based upon the source code and profiling data. Finally, a selection pass solves an optimization problem which selects a subset of the enumerated instructions to be implemented in the processor. Selected ISEs can be added to the Arithmetic Logic Unit (ALU) of a processor as additional hardware, resulting in a faster execution of the program, smaller code size, and consequently a larger hardware area for the processor.

ISE identification has been applied to hardware/software codesign using a variety of tools and techniques as described in [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. Many approaches for solving the ISE identification problem use constraints to limit the size of the solution space. Current ISE identification algorithms are either exact with worst-case exponential runtimes such as [3], [17], or heuristic with lower worst-case runtimes [8], [2], [6], [18], [19], [20], [21], [22], [23], [24]. The heuristic algorithms often have polynomial worst-case runtimes. The approach in [25] proves that their algorithm for the ISE enumeration problem has a polynomial worst-case complexity. In our view, the best approach so far is the recent work using dynamic programming [8].

Many of the heuristic ISE identification algorithms listed above have good computational runtimes for the average programs they were tested with. However, they may fail when there is a binding hardware constraint [2], a low number of dependencies in the basic block dataflow graph [17], and/or more than several hundred nodes [3], [2]. For example, the non-heuristic algorithm of [3] cannot compile AES within several days as noted in [2]. The approach in [2] is a follow-up to the work in [3] and shows an exponential runtime curve. The best results in [17] were in cases where there are many dependencies. A large number of dependencies between nodes in a dataflow graph reduces the number of possible ISEs, because the dependencies cause many candidate so-

lutions to lie outside of the register file bandwidth, hardware area, or convexity constraints in the enumeration model.

Some approaches to ISE identification use an iterative application of ILP models such as [2], while others use iterative graph traversal algorithms such as [3], [8]. The latter use a recursive algorithm for subgraph enumeration, while the former follows the approach of fine-grained graph analysis using mathematical modeling tools. The approach in [3] finds all convex subgraphs of a basic block dataflow graph whereas [2] aimed to decrease compiler execution time while enumerating the ISEs which contribute most to accelerate the program. Using the algorithm of [3], many realistic problems do not find a solution due to the exponential runtime complexity of the algorithm. The work of [8] brings us much closer to our goal, but still hangs for a long time on difficult benchmarks such as AES. Our approach (LSR) achieves a tractable runtime by incorporating iterations of graph traversal similar to [3] into iterations of a greedy best-first approach similar to [2]. What separates [8] from [3] is several new rules to prune the search space, which further allow to use dynamic programming.

Several papers, including [26], [19], [21], [22], [27], [15] describe algorithms for the enumeration of ISEs using several multiple-input single-output subgraphs in combination. Others such as [28] utilize a clustering method on enumerated ISEs to find larger convex multiple-input multiple-output ISE candidates. We do not consider low-effort minimal exploration heuristics such as [18] and other maximal subgraph approaches to be an adequate solution to our automated system design problem because they dedicate a large portion of the hardware area to code that is unlikely to be found in any other location in the application. We also do not consider the approach of [28] to be adequate, as the ISEs found by that algorithm would most likely exceed the limited hardware space alloted to ISEs, and the approach does not limit the number of inputs and outputs when searching for an ISE. In addition, the clustering and partitioning detailed in [28] does not allow for the discovery and exploitation of overlapping ISEs. An improvement in this line of thinking is provided by [13], where the I/O of the enumerated ISEs is strictly limited, dramatically improving algorithm runtime. The ISE enumeration algorithm should expose small instructions as well as large ones, and we have observed that the MAXMIMO instructions of a large basic block have many subgraphs which may not be considered when taking only maximal subgraphs. The work in [18] presented a very fast graph traversal algorithm for enumerating maximal subgraphs (large instructions) as ISEs using incompatibility graphs for each basic block. Algorithms like the one proposed in [18] often generate a very large datapath without really deeply exploring the design space for instruction reuse. The work of [15] involves selecting the enumerated ISEs for implementation using a graph covering technique and modeling the design space constraints such as convexity, forbidden nodes, and register bandwidth.

Efficient implementation of ISEs by combining dataphaths has been studied in the literature [29], [30], [19], [31], [11], [10], [14]. The ISE enumeration algorithm presented in [19] allows for resource sharing between selected ISEs.

Their method also allows for implementation of two overlapping ISEs through hardware duplication. This would allow for all instances of two sets of isomorphic ISE patterns to be implemented. Our approach disallows one of the two patterns in instances where the two ISEs overlap. For our implementation we did allow identical patterns in a program to use the same instruction hardware (isomorphism). In [31] isomorphic ISEs are found using pattern matching at the beginning of the ISE selection phase. They utilize isomorphic equivalence classes in much the same way that we do in order to find the maximum gain from isomorphic instructions. However, [31] used a greedy heuristic for solving the selection problem, whereas we have employed an ILP-based knapsack model to find a solution to the selection problem under the defined hardware constraints. The work in [19] shows that several instructions can be implemented using a single common datapath, whereas our work does not have this capability. We do not foresee that these approaches to instruction selection are mutually exclusive, but we do not explore the possibility of combining these methods in this paper. In terms of optimality, we rely upon the results of the ILP solver when providing the user with a guarantee that the given solution is at a global maximum. In [14], the design space is reduced by applying bounds to the number of functional units of each type, and further applying a hardware area constraint. Our instruction selection model is similar.

Another method to reduce hardware consumption by implementing ISEs that utilize some common hardware resources in a shared datapath is described in By comparing the tradeoff between the time saved, the space required and the number of ISEs using specific resources, they found that multiple ISEs that share resources can outperform a single ISE that provides better results than the multiple ISEs when measured individually. The work in [29] may be compatible with our approach to instruction selection. The knapsack problem could be extended to include modeling of the merger of ISEs to pack even more custom instructions into the same amount of hardware.

Similar to the work of [19] and [31],the ability to save hardware area during instruction enumeration and selection to meet an area requirement was explored in [32]. I/O constraints are applied to the enumeration problem, forcing the solution to exclude high cost ISEs. In addition to I/O constraints, a limitation on the depth of enumerated ISEs is imposed to prevent deep ISEs from being enumerated. This depth constraint reduces the size of the design space, which accelerates the enumeration algorithm. A limit on the number of high area operations in each ISE was imposed in order to prevent the enumeration of high cost ISEs. Unlike [31], [32] uses a branch and bound method to achieve optimal ISE selection, however a smaller number of ISEs is considered. Their approach begins by first establishing a good lower bound on the branch and bound algorithm (for ISE enumeration) by using dynamic programming to discover the minimum number of sequential nodes required to cover each enumerated ISE. The lower bound is calculated by finding exact covers of Single-Input Single-Output (SISO) fragments of each ISE. This information helps their selection algorithm to prune large portions of the design space.

They achieve runtimes on the order of microseconds for problems with fewer than 100 nodes.

In [16] a trade-off is presented to the user between hardware area and application execution time. Similarly, in this paper we present an algorithm where the user can specify a trade-off between compiler execution time and application speedup.

The approach of [33] is one of creating ISEs that cover entire loops. Their approach creates pipelined ISEs by selecting an entire loop as an ISE, while we find isomorphic ISEs by finding all of the instances of a loop. Our approach is to use a fine-grained search, as we can potentially find many good ISEs inside the loop which their approach cannot. A heuristic method for ISE enumeration was presented in [34], along with an approach for designing output constraints which accelerate model execution. In this work we did not allow inter-basic block ISEs. Such ISEs can be formed as explained in [4].

Several approaches exist for consideration of the memory hierarchy of an embedded processor in the enumeration or selection of ISEs. An approach is presented in [17] to reduce the need for parallel reading from the register file of the processor into the ISE. They accomplish this by transferring data to the ISEs in the ALU over several cycles as the data is needed. The cost of transferring data between the register file of the processor and the ISEs in the ALU was expressed in the model of [6]. Furthermore, [35] and [36] present an ISE identification algorithm which includes vector and register memory elements, and thus reduces pressure on the register bank. In [37] the memory hierarchy is considered during instruction selection, and [38] used the cache hierarchy to communicate data directly to the ISEs, increasing the available bandwidth and reducing the power consumption.

Another important aspect to the ISE identification problem is the relationship between software optimizations and the effectiveness of ISE identification. It was demonstrated in [4] and [39] that compiler optimizations transform the code in a way that heavily influences the effectiveness of an ISE identification pass. Selecting software optimizations in a smart way can lead to improvements in ISE identification performance as high as 55% [4]. In this paper we opted to turn on the optimizations in the COINS compiler as discussed in later sections. Our goal in this work was to use software optimizations as much as possible before adding hardware to the ASIP. Ideally, the software optimization would be so good that custom hardware would not be required.

The models which we discuss in this paper are binary linear programs. A relaxation of the Integer ILP is solved using Simplex, and then a branch-and-bound search tries to find a valid integer solution. Attempting to describe the worst-case complexity of this class of algorithms is difficult or impossible as explained in [40], that "the running time complexity of pure branch-and-bound algorithms usually cannot be described explicitly since the actual number of nodes in the branch and bound tree cannot be bounded a priori. Indeed there are particular instances of the [Knapsack Problem] which are extremely time consuming to solve for any branch-and-bound algorithm." We observe in the results of Fig. 5 that the execution time of the proposed algorithm LSR(6)

does show a runtime following the polynomial complexity curve $n^{3.5}$, where n is the number of nodes in the basic block. We have found in our experiments that the vast majority of the compilation time is spent in the ISE enumeration algorithm.

3 Multi-algorithm Instruction Set Extension

3.1 Overview

LSR is a refined version of SR which can perform a variable amount of design space exploration. The amount of design space exploration attempted by LSR is specified by combining SE iterations into the SR algorithm. The number of SE steps used in LSR is represented by a number in parentheses. LSR(6) would be a configuration of LSR where there are six iterations of SE for every iteration of SR, whereas LSR(2) would have only two SE iterations. By combining SE and SR we take advantage of SRs ability to quickly find and remove large ISEs from a dataflow graph, and SEs ability to stumble upon high speedup isomorphic instructions. After each execution of SR on the dataflow graph we allow a varying number of executions of SE, as specified by the user (e.g. LSR(6)). The focus of SR is solely on absolute speedup. This is a key point and should be emphasized: LSR is a deterministic and repeatable algorithm which explores more of the design space than SR, and in less time than SE, although it does not intelligently search the extra design space it encounters. Therefore, it may find a better solution than SR or it may not. We posit that LSR has the same focus on speedup as SE and SR but allows for a higher probability to find isomorphic instructions.

Algorithm details for SE, SR, and LSR are provided in [41]. We present the algorithms here in brief. 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 ← 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 = {∅}; disabled_ISEs = {∅}; SE_Buffer = {∅}
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 (∀ise ∈ SE_Buffer) do
15:            if ∄node ∈ (ise → nodes), node ∈ forbidden_nodes then
16:               SE_Buffer.remove(ise)
17:               disabled_ISEs.remove(ise)
18:               forbidden_nodes = forbidden_nodes ∪ (ise → nodes)
19:            end if
20:         end for
21:         clear SE_Buffer
22:         if (∄node ∈ (NEW_ISE → nodes), node ∈ forbidden_nodes) then
23:            forbidden_nodes = forbidden_nodes ∪ (NEW_ISE → 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 ∪ (NEW_ISE → nodes)
30:      end if
31:   else
32:      SE_Buffer.add(NEW_ISE)
33:      disabled_ISEs.add(NEW_ISE)
34:   end if
35:   for (ise ∈ disabled_ISEs) do
36:      if (forbidden_nodes ∪ (ise → nodes)) ≡ 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 design of LSR is an attempt to take the best of both worlds by finding the ISE with the largest speedup (in the same fashion as SR), and then enumerate a set number of ISEs according to the SE algorithm. During the SE steps, LSR puts the identified ISEs into an ordered queue called the SE buffer. Each time LSR switches from an SE step to an SR step, the SE buffer is searched to find any mutually exclusive ISEs. Mutually exclusive ISEs in the buffer are ISEs normally found by the SR steps that happen to have been found during an SE step. Nodes of any identified mutually exclusive ISEs from

the SE steps are marked as forbidden (as though they were found in an SR step). LSR then empties the SE buffer. After processing the SE buffer, the instruction enumerated during the current SR step may no longer be mutually exclusive. In this case, the identified ISE is added to the emptied SE buffer. LSR then continues searching the design space, repeating this process until no more ISEs can be identified. Additionally, each time LSR performs an SR step it checks to see if any SE ISEs contain only forbidden nodes. If so, then the ISE is already disabled by the fact that its nodes are forbidden, and that ISE is removed from the list of disabled ISEs.

3.2 Motivating Example

The following motivating example will be used to show how SR and LSR work. The small example shown in Fig. 1 is a basic block written in C. The dataflow graph for this program is shown in Fig. 2. 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 listed in Table 1 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 1 Assumed hardware sizes and latencies for example shown in Fig. 1

Instruction	Hardware size (LEs)	Latency (cycles)
BAND	32	1
ADD	49	2
MUL	1598	3

```
r1 = x1 & x2;
r2 = x2 & x3;
r3 = x3 + x4;
r4 = x5 + x6;
r5 = x6 * x7;
r6 = x7 * x8;
```

Fig. 1 Motivating example program. All variables are integer.

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 is below the hardware size limit, 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

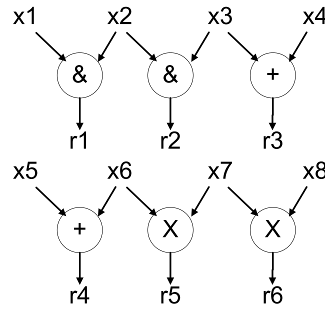

Fig. 2 Dataflow graph for program in Fig. 1

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 overlap the first two ISEs enumerated and save 3 cycles. In the instruction selection pass of the ISE identification 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.

3.3 Compiler and Algorithm Implementation Details

This work focuses on improving the state of the art in codesign compilers by trading compilation time for solution quality. Our toolchain contains an eclipse-based user interface for program entry and hardware design, an extended version of the COINS compiler, a customized copy of the GNU binutils package, and a module for generating VHDL hardware descriptions of the designed system [42], [43], [44]. In this work we focus entirely on the compiler aspect of the toolchain, which takes as inputs the source code for a program and the design constraints (such as the available hardware resources and register bandwidth). The COINS compiler can accept as input either C or Fortran, and can target many processor architectures including Alpha, ARM, ARM with Thumb extension, MicroBlaze, MIPS, PowerPC, SH4, SPARC, x86, x86 64-bit, and x86 with a Single Instruction Multiple Data (SIMD) extension [45]. The COINS compiler was extended for this work to include our own implementation of a basic block profiler, and a pass for performing ISE identification. For this work we targeted the SPARC-V8 ISA.

A customized basic block profiler was implemented for the COINS compiler and used to measure basic block execution frequency for the input program. The profiles were executed in the Cygwin environment [46]. Custom profilers are common in ISE toolchains [9].

3.3.1 ISE Identification Compiler Pass

The ISE identification algorithm is executed late in the compilation flow so that most code optimizations do not change the structure and function of the code after custom instructions have been selected. The pass is also scheduled late because software optimizations are a preferred method for getting faster code than hardware optimizations. The number of ISEs that can be added to a processor is usually limited. For example, the NIOS-II processor can be extended by no more than 256 ISEs [47].

Inside the ISE identification pass, the statements of the input program are rewritten into three address code using a Static Single Assignment (SSA) transformation. In order to perform ISE identification, basic blocks from the control flow graph of the program are converted into dataflow graphs in the backend of the compiler. Each dataflow graph is a Directed Acyclic Graph (DAG) composed of graph nodes representing computation and arcs between nodes representing data dependencies. These graphs are then parsed to generate models which find custom instructions, also referred to as ISEs.

To ensure proper scheduling, the model of each basic block enumerates only convex ISEs [3]. A subgraph of a given graph is convex only if for every pair of nodes in the subgraph, all of the paths connecting them involve only nodes belonging to the subgraph. Non-convex instructions are any subgraph of the dataflow graph which has an output which leads back to any input of the subgraph. Any non-convex subgraph could never be scheduled because the operands of the instruction depend on its results. A set of ISEs is identified as candidate instructions from within the model of each basic block in a step called instruction enumeration. The generated models are written in the format specified by compiler flags: either a free tool called lp_solve [48], or a commercial tool called LINGO [49]. The models are also compatible with solvers such as IBM's CPLEX [50].

Once enumerated, the list of candidate ISEs is compared to identify instructions with identical datapaths in a step called directed labeled graph isomorphism testing. The JGraphT package was extended to perform this task [51]. The duplicate dataflow patterns are called isomorphic ISEs [2].

Next, an instruction selection model is used to maximize the speedup of the application without exceeding the limit on the available hardware in the design space. A cost function is used in the instruction selection model to evaluate the design space and select a subset of the available instructions given their cost in terms of design parameters (hardware size and speedup of the application). We have added another dimension to the cost function of the instruction selection model which resolves the conflict between any mutually exclusive ISEs. The model discovers a subset of all candidate ISEs to maximize the speedup of the program under the specified hardware size constraint and input/output constraint (available hardware area, maximum number of inputs, maximum number of outputs). Finally, the selected ISEs are passed along to the hardware implementation module and the code generation package of the COINS compiler.

3.3.2 Further ISE Identification Details

As described in Section 3.3.1, Algorithm 4 was implemented in the COINS compiler to profile the user's application, enumerate ISEs, analyze the ISEs, and then select ISEs. The inputs to our ISE identification pass are a program representation called a *module* in the COINS compiler, the execution profile of the program and a set of design space constraints. Design space constraints that can be specified by the user are limitations on the area available for the hardware implementation of the ISEs (the hardware constraint), limitations on the number of input/output operands for the ISEs (the I/O constraints), and the ISE enumeration algorithm to use (SE, SR, or LSR). In Algorithm 4, the call to ISE_ENUMERATION points to the selected ISE enumeration algorithm: LSR(n), SE, or SR. The details of the instruction selection knapsack model INSTRUCTION_SELECTION(candidateISEs, isomorphismData), we used for this work are presented in the Instruction Selection section of [41].

The conversion of each basic block in the program to SSA form was required before the dataflow graphs could be built in Algorithm 4. Therefore, the readily available dataflow graph analysis code built into the SIMD backend module in the COINS compiler was utilized [45]. Code was also duplicated directly from the SSA pass of the compiler.

Algorithm 4 ISE Identification

```
1: ALGORITHM : ISE_IDENTIFICATION_PASS(module)
2: Parse profile of basic block execution frequency
3: candidateISEs = {∅};
4: blocks ← unsorted basic blocks from every function of module
5: for (each function in module) do
6:    convert function into three address code
7:    convert function into single assignment form
8: end for
9: for (each basicBlock in blocks) do
10:    if basicBlock is called in the profile then
11:       candidateISEs = candidateISEs ∪ ISE_ENUMERATION(basicBlock)
12:    end if
13: end for
14: isomorphismData = GRAPH_ISOMORPHISM(candidateISEs)
15: selectedISEs =
    INSTRUCTION_SELECTION(candidateISEs, isomorphismData)
```

4 Experiments

This section begins by explaining the assumptions that went into our experiments. Four experiments are described, followed by the presentation of results, and hypothesis validation from the presented data. In the first experiment, the design space is not constrained by a binding hardware constraint. As expected, LSR does not provide significant benefit. In the second experiment, the hardware size is binding as well as the I/O constraints, and we observe a tangible benefit from using LSR compared to SR. For the third experiment,

we validated the assumption that tuning LSR trades execution time for solution quality, and in the fourth experiment we compare LSR to SR for various MiBench benchmarks [52].

4.1 Experiment Setup

The four experiments performed in this section were executed on an x86-64 with an Intel Core i7 processor and 12 GB of DDR3 RAM. The operating system was Windows 7 x64. Several optimization passes were used to improve the code performance before the ISE identification pass was called. Software optimizations which accelerate the application are strongly preferred over adding hardware to the embedded system. These optimizations are there to ensure that we did not needlessly add hardware to the embedded system. Application speedup is measured compared to the optimized code instead of the unoptimized code to present a more realistic picture of the real-life design space.

The $-O3$ option was selected as the optimization level for the COINS compiler, triggering several optimizations. Flags were set to specify additional front-end optimizations: loop expansion, and constant propagation and folding. The SSA optimizations specified were constant folding and propagation with conditional branches.

There were two sources for the programs used to obtain our results. Benchmarks from [52] were used along with a set of generated single basic block programs containing randomly assigned operations between integer and/or floating point variables. The approach of processing various sized programs as well as fixed sized benchmarks is followed by [3] and many others. The generated programs had between 10 and 1040 statements, increasing in increments of 10. Each program contained a mixture of fixed point and floating point operations, and a random amount of dependencies. An example of one of the programs with 10 statements is displayed in Fig. 3. Conversion of each generated program to a dataflow graph resulted in graphs with between 14 and 900 nodes per program.

```
i8=i8 < i5;
i2=i2 & i9;
f1=f5 * f3;
i6=i3 & i4;
i8=i5 & i2;
f6=f7 / f6;
i0=i6 / i2;
f6=f3 + f3;
i4=i8 * i5;
i7=i1 % i8;
f3=(float)i9;
```

Fig. 3 Example of a randomly generated basic block used in the experiments.

In the first three experiments, the generated programs of varying size were compiled with different hardware size constraints, different I/O constraints, and different ISE enumeration algorithms. We varied these parameters to discover trends in compilation time and speedup. Each generated program is different, and so the design space for these first experiments stays the same while the compilation parameters change. Because the generated programs are incrementally larger in size, we can observe the effect of the compiler changes on various basic block sizes. The I/O constraint for the experiments presented in Fig. 4 and 5 was set to 8 inputs and 8 outputs, for Fig. 6 and 7 was set to 4 inputs and 4 outputs. The lp_solve solver was used for the experiments in this paper. Other solvers such as CPLEX and LINGO are supported through available hooks but are not examined here.

In our first experiment we observed the performance of SR, SE, and LSR(6) for a set of computer generated basic blocks, and the hardware constraint was not binding. The second experiment was a repetition of the first with the following modification: the available hardware area was reduced to make the constraint binding. Next, we compared the performance of SR to that of LSR(2), LSR(4), and LSR(6). For each of these three experiments, we observed the compiler execution time and the speedup for each algorithm. The fourth experiment is an evaluation of LSR using the MiBench benchmark suite.

4.2 Assumptions

Speedup is calculated using the estimated time saved, and the estimated execution time before the ISEs were added to the processor. We estimate the execution time of the program before ISEs are added by multiplying the execution frequency of each basic block (data obtained from our profiler) by the software latency of the operations in the basic block. We calculated the program speedup by comparing the time saved by the ISE algorithms for each program, and compared it to the purely sequential runtime. The equation used to calculate the program speedup was:

$$Speedup = \frac{ExecutionTime}{ExecutionTime - TimeSaved} \quad (1)$$

This speedup calculation does not take into account cache misses, the forwarding/hazards within the processor, the pipelining of ISEs within the ALU, or the effect on the clock rate of the design caused by adding ISEs to the ALU. The hardware size of the system is measured in Logic Elements (LEs) and latency is measured in clock cycles. Memory bits were modeled such that each bit of memory costs one LE. The hardware size and latency estimates were obtained using functional simulations in Quartus-II, with the target device set to target a Cyclone I EP1C FPGA [53]. The results in this paper are based on estimates from simulation, and the inputs to the simulation are from real measurements from the FPGA or from the compiler.

4.3 Results

4.3.1 Experiment 1: Non-binding hardware size constraint and loose I/O constraint

The compiler execution time for SE, SR, and LSR(6) were compared using programs of varying size. The SR algorithm finds large custom instructions, and so performs well in tests where there is nearly unlimited hardware available. As can be seen in Fig. 4, LSR(6) and SR both perform similarly.

As shown in Fig. 4, SE has better overall results in terms of speedup in comparison to either SR or LSR(6), but this gain is not worth the cost in terms of compiler execution time as can be seen in Fig. 5. The peak improvement of LSR(6) over SR was 25.8%. However, since there was no binding hardware constraint we found that as expected the average result was a 0.18% speedup improvement for LSR(6) over SR. Effectively there is little benefit to using LSR(6) with no binding hardware constraint. Comparing the execution time of Fig. 5 to Fig. 7 of [2], we show similar performance for 10 nodes (500 ms), and for 100 nodes we show results falling in the upper range of their data (10,000 - 100,000 ms).

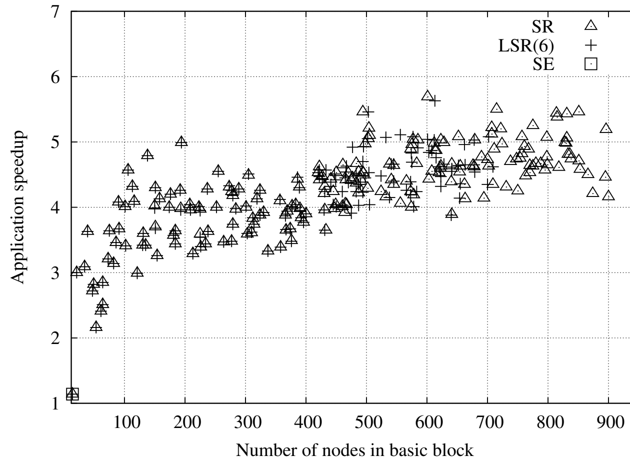

Fig. 4 Program speedup comparison with a non-binding hardware constraint and I/O constraint of (8,8)

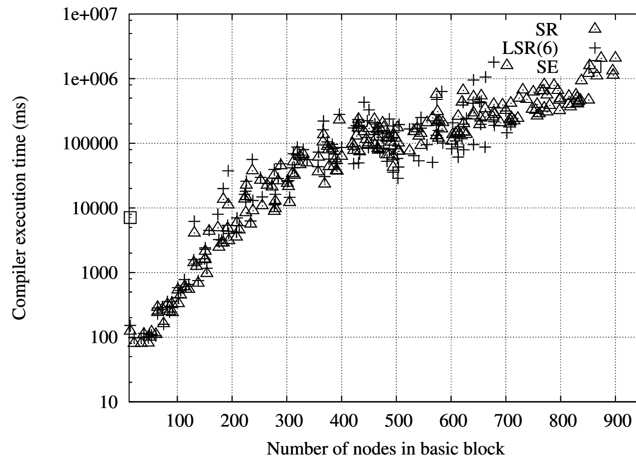

Fig. 5 Compiler execution time comparison with non-binding hardware constraint and I/O constraint of (8,8)

4.3.2 Experiment 2: Binding hardware size constraint and I/O constraints

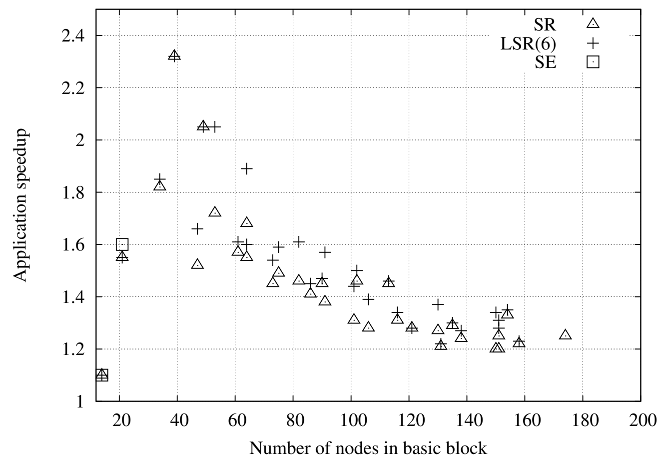

Fig. 6 Program speedup comparison with low hardware constraint of 10,000 LEs and I/O constraint of (4,4)

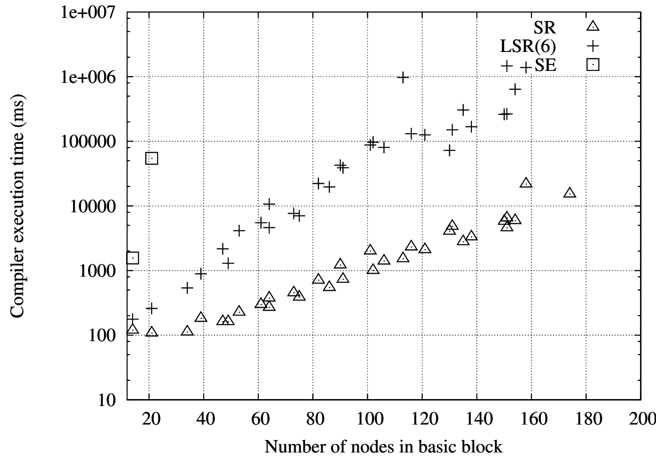

Fig. 7 Compiler execution time comparison with low hardware constraint of 10,000 LEs and I/O constraint of (4,4)

In this second set of tests with a binding hardware constraint, SR's ability to implement large custom instructions is hampered by the lack of hardware. LSR, with its ability to find and implement more isomorphic custom instructions than SR, performs substantially better. On average, LSR(6) outperformed SR by 4.5% for the speedup comparison shown in Fig. 6. For the basic block with 53 nodes LSR(6) caused a 19.5% improvement in speedup compared to SR.

The reason that only two of the smallest basic blocks could be analyzed with SE becomes apparent when the compiler execution times are shown. While SE requires only 10 times more compilation time than SR or LSR(6) for a basic block with 10 nodes, it requires over 500 times what SR requires for a basic block with 21 nodes, as we see in Fig. 7. This curve increased so rapidly that for larger basic blocks the compiler execution time exceed 36 hours while the other algorithms needed only seconds. LSR(6) and SR follow similar complexity curves, with LSR(6) requiring more compiler time to execute.

It is important to clarify why the increase in the number of nodes in Fig. 6 results in a decrease in the speedup. As the available hardware stays the same, the program size is increasing. We can see that the ratio of available hardware to program size is therefore also decreasing, and the ability to accelerate the program drops off towards a low constant. Having more nodes to analyze in the program, it is possible to find many ISEs, but the fact that the speedup hovers slightly above 2.5x shows that the algorithm can find some isomorphic ISEs which keep the speedup above 1. In Fig. 6 we observe the limits of hardware acceleration using a small area for ISEs relative to program size.

LSR(6) and SR are constrained by the small hardware size, and they run out of custom instructions that can fit in the available space. Once the enumeration is completed the models return their results to the compiler [41]. This

is seen as the curve in Fig. 6 decays to a speedup of 1. As the program size is increased, the available hardware did not increase. Thus the relative speedup provided by the ISEs shrinks and the program size grows. While the same number of custom instructions are created, their overall percentage in comparison to the size of the basic block is smaller. The high hardware constraint in Fig. 4 allows for nearly all of the ISEs to be implemented in hardware, keeping the overall speedup relatively consistent and the shape of the speedup curve is linear.

As with the high hardware constraint, SE results in slightly better speedup than SR and LSR(6) for tests with a low hardware constraint. However, the SE algorithm takes such a long time to execute that it can only be applied in practice to the smallest basic blocks.

There are trends in the speedup (rising and then dropping off) which indicate that LSR is improving the solution over SR. When the I/O constraint is high, for example (8,8), and the hardware size constraint is binding, then we observed less than a 1 percent improvement in speedup for small basic blocks when using LSR compared to using SR. Only for larger basic blocks does the speedup difference start to appear. If however, the I/O constraint is (4,4) or (2,1), as we would see in a contemporary processor design, we see a much better outcome of up to 19.5% better than SR for the smaller basic blocks. We deduce from these findings that we are are able to push around the range where LSR is most helpful by varying the I/O and hardware size constraints. We conclude based on the experiments above, that the benefit of LSR is greatly increased when there is a binding hardware size constraint on the design space. As shown in Fig. 4 and 5 there are many cases where LSR provides the same speedup as SR but with a much longer algorithm execution time.

4.3.3 Experiment 3: Tuning LSR

Comparing SR with various settings for LSR in Fig. 6 and Fig. 8, we see that LSR performs better on average than SR when there is a binding hardware constraint. The average application speedup in Fig. 8 for basic blocks with less than 255 nodes was 3.36 for SR, while LSR(6) shows the best speedup with a result of 3.40. We observe in these numbers a correlation between the number of SE steps in LSR, and the average speedup. LSR(2), LSR(4), and LSR(6) are variants of LSR which perform differing amounts of subgraph enumeration during each LSR iteration. We observe that the added compiler execution time for deep exploration with LSR(6) is only a few minutes even for basic blocks with over 100 nodes. The user can tune LSR to whatever level of design space exploration that they feel is appropriate. There was a peak 18.9% speedup improvement of LSR(6) over SR, and 17.4% speedup improvement of LSR(4) and LSR(2) over SR. On average there was a 1.42% speedup improvement for LSR(6) over SR, a 1.27% speedup improvement of LSR(4) over SR, and a 0.84% improvement of LSR(2) over SR. These are small but tangible gains in performance when there is a severe hardware limitation on the design space.

It was hypothesized that a higher number of SE steps would lead to a higher speedup and a longer compiler execution time. As shown in Fig. 8, this hypothesis was validated, with LSR(6) providing the best speedup on average. Since the hardware constraint was tightly binding, the overall differences in speedup were also small for the large graphs. We observed that each of the algorithms runs out of available hardware rapidly as the basic block size increases but the area constraint remains constant. For small basic block no differences were observed because each of the algorithms arrived at the best possible solution. This result indicates that LSR improves with the depth of the search, and the improvement is best when there is a good match between the basic block size and the available hardware area.

Fig. 9 shows the compiler execution times for the LSR variants and SR. LSR(6) has the worst execution time performance of the three LSR algorithms with a predictable increase in compiler execution time. This experiment provided evidence that we have successfully created a tunable algorithm. Further experimentation will be needed to provide information on which depth constraint provides the best trade-off of speedup and compiler execution time for differing hardware constraints and perhaps even for different programs.

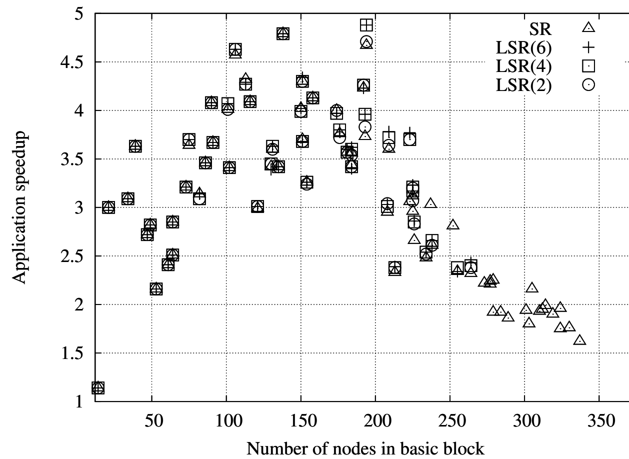

Fig. 8 Speedup comparison for SR, LSR(2), LSR(4) and LSR(6) with a hardware constraint of 60,000 LEs and an I/O constraint of (8,8)

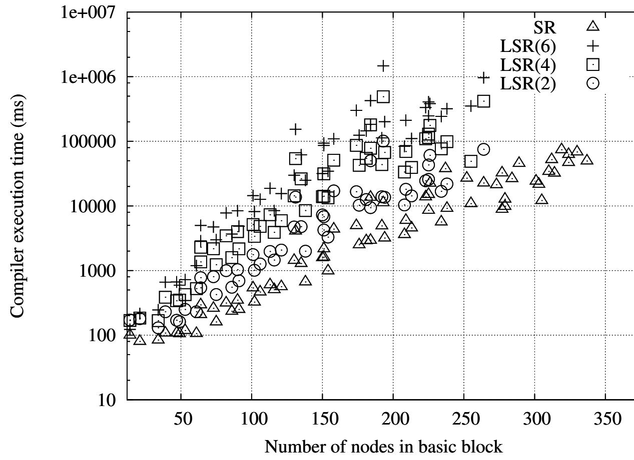

Fig. 9 Compiler execution time comparison for SR, LSR(2), LSR(4) and LSR(6) with a hardware constraint of 60,000 LEs and an I/O constraint of (8,8)

4.3.4 Experiment 4: LSR for MiBench

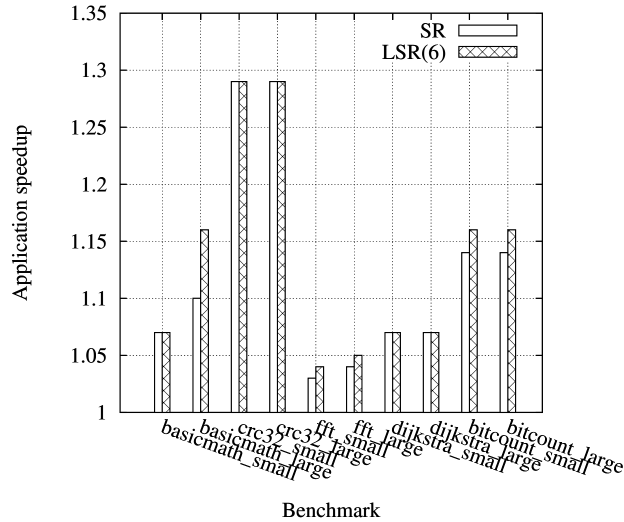

Fig. 10 Benchmark speedup resulting from the use of LSR(6) with a hardware constraint of 1,000 LEs and an I/O constraint of (4,4)

When comparing SR to LSR(6) using a binding hardware constraint, it is clear that there is often added benefit for our approach compared to the prior art. In Fig. 10, the speedup results for both algorithms are presented side

by side. The benchmarks were taken from the MiBench benchmark suite and executed using each of the two workloads provided: small and large [52]. There is a possibility that a search of the design space will yield no improvement; however, when it does pan out, the improvement can make a big difference to a processor designer. The moderate speedup displayed here is a result of the limited design space, and should not be taken to imply a weakness in the underlying methodology. Of course the speedup would be higher without any I/O constraints or hardware constraints, comparable to the work of [6], [2]. In such a case, we know from previous experiments that our approach would not be more beneficial, and would in fact execute slower by comparison. The most significant improvement in speedup was for the *basicmath* benchmark under the large workload, which saw a 6 percent higher speedup using our approach.

4.3.5 Discussion

We found that SR discovers excellent solutions when there is a loose (non-binding in the ILP models) hardware constraint, but the solution quality degrades as the hardware constraint tightens. This is because SR removes a portion of the design space from consideration during each iteration of the algorithm. Quickly shrinking the design space limits SRs ability to find isomorphic instructions. Furthermore, large instructions such as those found by SR are less flexible to code changes, and are less likely to be found multiple times in a program. By comparison, SE enumerates all valid ISEs which will result in an equal or better speedup than SR in all cases. As SE enumerates all valid ISEs, the compiler execution time is prohibitively long.

5 Conclusions

ISE identification was implemented in the COINS compiler infrastructure for the first time. Our experimentation and observations showed that we were able to create an ISE identification algorithm that balances application speedup and compiler execution time. In addition to being able to provide more speedup than the state of the art for area constrained designs, we were able to develop a tunable algorithm. This tunability allows designers to easily adjust the amount of effort the tool will put into finding more application speedup at the cost of a higher compiler execution time. Our new algorithm (LSR) was able to improve the speedup of a program by as much as 25.8% compared to the state of the art. Our instruction selection model is able to understand both mutually exclusive ISEs and overlapping ISEs, maximizing the speedup of the application in a way that has not been tried before.

We may proceed to investigate the possibility that there exist faster algorithms similar to LSR which are tunable, and effective in design spaces where there is a binding hardware constraint. We wish to find an algorithm more generally applicable to the design space regardless of the number of nodes in a basic block.

Acknowledgements The authors would like to thank NSERC, and the COINS compiler group. Also, we wish to express special thanks to Dr. Subhasis Banerjee for helping to prepare this paper for publication.

References

1. Tensilica - hardware and software development tools. URL http://www.tensilica.com/products/hw-sw-dev-tools.htm
2. K. Atasu, G. Dundar, C. Ozturan, in *Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2005), pp. 172–177
3. K. Atasu, L. Pozzi, P. Ienne, in *Design Automation Conference* (2003), pp. 256–261
4. P. Bonzini, L. Pozzi, in *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2006), pp. 242–252
5. K. Atasu, Hardware/software partitioning for custom instruction processors. Ph.D. thesis, Bogazii University, Istanbul, Turkey (2007)
6. K. Atasu, R.G. Dimond, O. Mencer, W. Luk, C. Ozturan, G. Dundar, in *Proc. Design, Automation & Test in Europe Conference & Exhibition, DATE* (2007), pp. 588–593
7. C. Galuzzi, K. Bertels, in *ARC '08: Proceedings of the 4th international workshop on Reconfigurable Computing* (2008), pp. 209–220
8. J. Ahn, I. Lee, K. Choi, in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific* (2011), pp. 573 –578. DOI 10.1109/ASPDAC.2011.5722255
9. H. Hubert, B. Stabernack, Circuits and Systems for Video Technology, IEEE Transactions on **19**(11), 1680 (2009). DOI 10.1109/TCSVT.2009.2031522
10. H. Lin, Y. Fei, in *Computer Design, 2009. ICCD 2009. IEEE International Conference on* (2009), pp. 158 –165. DOI 10.1109/ICCD.2009.5413161
11. N. Pothineni, P. Brisk, P. Ienne, A. Kumar, K. Paul, in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific* (2010), pp. 707 –712. DOI 10.1109/ASPDAC.2010.5419795
12. A. Verma, Y. Zhu, P. Brisk, P. Ienne, in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on* (2009), pp. 54 –57. DOI 10.1109/SASP.2009.5226336
13. A. Verma, P. Brisk, P. Ienne, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **29**(3), 341 (2010). DOI 10.1109/TCAD.2010.2041849
14. D. Wu, I. Lee, J. Ahn, K. Choi, JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE **11**(1), 8 (2011). DOI DOI:10.5573/JSTS.2011.11.1.051
15. A. Yazdanbakhsh, M. Salehi, S. Fakhraie, in *Future Information Technology (FutureTech), 2010 5th International Conference on* (2010), pp. 1 –6. DOI 10.1109/FUTURETECH.2010.5482719
16. M. Zuluaga, N. Topham, in *Embedded Computer Systems (SAMOS), 2010 International Conference on* (2010), pp. 282 –291. DOI 10.1109/ICSAMOS.2010.5642056
17. L. Pozzi, P. Ienne, in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2005), pp. 2–10
18. K. Atasu, O. Mencer, W. Luk, C. Ozturan, G. Dundar, in *International Conference on Application-Specific Systems, Architectures and Processors* (2008), pp. 1–6
19. J. Cong, Y. Fan, G. Han, Z. Zhang, in *Proceedings ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays* (ACM, New York, NY, USA, 2004), pp. 183–189
20. S. Das, P. Chakrabarti, P. Dasgupta, in *Proceedings of the IEEE International Conference on VLSI Design* (2006), pp. 293–298. DOI 10.1109/VLSID.2006.106
21. C. Galuzzi, E.M. Panainte, Y. Yankova, K. Bertels, S. Vassiliadis, in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis* (2006), pp. 160–165
22. C. Galuzzi, K. Bertels, S. Vassiliadis, in *ICFPT 2007 - International Conference on Field Programmable Technology* (2007), pp. 337 –340. DOI 10.1109/FPT.2007.4439280
23. G. David, P. Darin, in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (ACM, New York, NY, USA, 2003), pp. 137–147

24. S. Mohanty, V.K. Prasanna, S. Neema, J. Davis, in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems* (ACM, New York, NY, USA, 2002), pp. 18–27. DOI http://doi.acm.org/10.1145/513829.513835
25. P. Bonzini, L. Pozzi, in *Proc. Design, Automation and Test in Europe Conference and Exhibition, 2007. DATE '07* (2007), pp. 1331–1336. DOI 10.1109/DATE.2007.364482
26. A. Peymandoust, L. Pozzi, P. Ienne, G.D. Micheli, in *IEEE International Conference on Application-Specific Systems, Architectures, and Processors* (2003), pp. 108–118
27. C. Galuzzi, D. Theodoropoulos, R. Meeuws, K. Bertels, in *Proc. Design, Automation Test in Europe Conference Exhibition, DATE* (2009), pp. 548 –553
28. C. Galuzzi, D. Theodoropoulos, K. Bertels, in *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, IC-SAMOS* (2008), pp. 65 –73. DOI 10.1109/ICSAMOS.2008.4664848
29. M. Zuluaga, N. Topham, in *2008 Symposium on Application Specific Processors, SASP 2008* (2008), pp. 7 –13
30. M. Zuluaga, N. Topham, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **28**(12), 1788 (2009)
31. N. Pothineni, A. Kumar, K. Paul, in *Proc. IEEE International Frequency Control Symposium and Exposition* (2008), pp. 348 –353. DOI 10.1109/VLSI.2008.63
32. Y.S. Lu, L. Shen, L.B. Huang, Z.Y. Wang, N. Xiao, Computers Digital Techniques, IET **3**(1), 14 (2009). DOI 10.1049/iet-cdt:20070104
33. M. Zuluaga, T. Kluter, P. Brisk, N. Topham, P. Ienne, in *IEEE 7th Symposium on Application Specific Processors, SASP* (2009), pp. 114 –121. DOI 10.1109/SASP.2009.5226328
34. J. Reddington, G. Gutin, A. Johnstone, E. Scott, A. Yeo, in *Proc. 12th IEEE International Conference on Computational Science and Engineering, CSE*, vol. 2 (2009), vol. 2, pp. 17 –24. DOI 10.1109/CSE.2009.167
35. P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, N. Dutt, in *Proceedings of the 41st Design Automation Conference* (2004), pp. 729–734
36. P. Biswas, N. Dutt, P. Ienne, L. Pozzi, in *Proc. Design, Automation and Test in Europe* (2006), pp. 212–217
37. J. Wu, C. Lin, D. Chen, Y. Wang, in *International Conference on Embedded Software and Systems* (2008), pp. 471–478
38. T. Kluter, P. Brisk, P. Ienne, E. Charbon, in *Proc. Design Automation Conference, DAC* (2009), pp. 31 –36
39. R.V. Bennett, A.C. Murray, B. Franke, N. Topham, in *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES* (ACM, New York, NY, USA, 2007), pp. 83–92
40. H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems* (Springer-Verlag, Berlin, Germany, 2004)
41. D. Shapiro, M. Bolic, Improved ise identification under hardware constraint. Tech. Rep. TR-2011-1, Faculty of Engineering, University of Ottawa, Canada (2011)
42. Eclipse IDE. URL http://www.eclipse.org/
43. A compiler infrastructure project. URL http://www.coins-project.org/international/
44. Gnu binutils. URL http://www.gnu.org/software/binutils/
45. T. Watanabe, T. Fujise, K. Mori, K. Iwasawa, I. Nakata, in *Innovative Architecture for Future Generation High-Performance Processors and Systems* (2007), pp. 60–69
46. D. Lazenby, Linux J. p. 14 (2000)
47. Altera. Nios II custom instruction user guide
48. lp_solve. URL http://sourceforge.net/projects/lpsolve
49. L. Schrage, *Optimization Modeling with LINGO*, 6th edn. (Lindo Systems Inc., Chicago, IL., 2006)
50. Ibm ilog cplex optimizer. features and benefits. URL http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/about/
51. B. Naveh. The eigenbase project
52. M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, in *Proceedings of IEEE 4th Annual Workshop on Workload Characterization* (2001), pp. 3–14
53. Quartus II 9.0 sp2 web edition. URL http://www.altera.com