Fast Automated Generation of High-Quality Instruction Set Extensions for Processor Customization

Partha Biswas
partha@cecs.uci.edu

Sudarshan Banerjee
banerjee@cecs.uci.edu

Nikil Dutt
dutt@cecs.uci.edu

Center for Embedded Computer Systems
Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Laura Pozzi
laura.pozzi@epfl.ch

Paolo Ienne
paolo.ienne@epfl.ch

Processor Architecture Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
Lausanne, Switzerland

ABSTRACT

Customization of processor architectures through Instruction Set Extensions (ISEs) is an effective way to meet the growing performance demands of embedded applications. A high-quality ISE generation approach needs to obtain results close to those obtained by experienced designers, particularly for complex applications that exhibit regularity: expert designers are able to exploit manually such regularity in the data flow graphs to generate high-quality ISEs. Previously a genetic formulation has been proposed that partially addresses this goal but which lacks determinism in generating good solutions consistently. In this paper, we present ISEGEN — an ISE generation tool — that consistently generates high-quality ISEs that match structures manually identified by expert designers. ISEGEN is an iterative refinement technique based on the basic principles of the well-known *Kernighan-Lin (K-L)* min-cut heuristic. Experimental results on a number of MediaBench, EEMBC and cryptographic applications show that our ISEGEN technique results in an average application speedup of 2X over execution on the core processor. We also show that our ISEGEN approach achieves 35% more speedup than the previous genetic formulation on a large cryptographic application (AES) by exploiting effectively its regular structure. For applications that lack a regular structure, our ISEGEN technique is also 20X faster than the genetic formulation while yielding comparable solutions.

1. INTRODUCTION

Continuing advances in manufacturing processes have made it possible for processor vendors to build increasingly faster processors. However, newer applications place an increasing demand on performance, at a rate faster than that achievable by processors. These trends have necessitated the migration of critical computations from the core processor to an application-specific unit that is able to perform compute-intensive tasks efficiently. We call such a unit an *Ad-hoc Functional Unit (AFU)*. The AFU accelerates critical operations of application algorithms by executing application-specific *Instruction Set Extensions (ISEs)*. Automatic generation of ISEs is essentially the task of hardware-software partitioning applied at an instruction-level granularity.

The Kernighan-Lin (K-L) min-cut algorithm is a well-known graph partitioning heuristic originally designed for circuit partitioning [2]. Recently, this heuristic has been successfully adapted for task-level partitioning of a system into hardware and software [1]. In this paper, we apply the K-L heuristic at a finer granularity — at the instruction-level — so as to automatically generate ISEs. We call our solution of hardware-software partitioning at the instruction-level granularity ISEGEN.

Our motivation for employing the K-L heuristic is to generate solutions close to those obtained manually by expert designers. In order to match the solution quality of an experienced designer, it is important that the solution space-walker closely mimic the design decisions taken by the designer. An ISE generation tool having such a space-walker is especially useful for large complex applications where manual identification of ISEs is practically impossible.

Previously, a genetic formulation for ISE generation [4] was described that partially addressed this goal by generating good overall speedup. However, there is less determinism in the generated ISEs because of two reasons. First, the genetic algorithm is stochastic in nature and therefore may produce different structures of ISEs on different runs. Second, it does not model all the designer goals in its cost function. For example, the growth of large clusters is not explicitly favored through its cost function. On the other hand, our ISEGEN approach is based on iterative refinement of the solution space, whose control parameters closely model the decisions taken by an expert designer. Therefore,

the solutions generated by ISEGEN is more deterministic as compared to a genetic solution. We show the efficacy of ISEGEN on a number of embedded applications selected from MediaBench, EEMBC and cryptographic suites. On a large cryptographic application (AES), ISEGEN — by effectively exploiting its regular structure — generates 35% more speedup than the genetic solution. For all the other applications that lack a regular structure, ISEGEN also runs 20X faster than the genetic approach while yielding comparable speedup.

The rest of the paper is organized as follows. In Section 2, we discuss related research work and our motivation. Section 3 defines the problem at hand. We discuss our ISEGEN approach in Section 4. Section 5 describes the experimental results to show the efficacy of our approach. Finally, Section 6 concludes the paper.

2. STATE OF THE ART AND MOTIVATION

The problem of ISE generation for application specific processors has been studied for almost a decade. Loosely stated, the problem is to identify legal subgraphs/clusters of a set of *Data Flow Graphs (DFGs)* that are potential sources of speedup. In techniques developed for reconfigurable computing [12, 11], the subgraphs are often generated starting from an output node by adding predecessor nodes until some constraints are violated. As a result, the generated ISEs only have a single output. However, we support multiple outputs in the ISEs as long as they meet the architectural constraints. A greedy clustering heuristic proposed in [5] handles multiple outputs but considers only connected subgraphs. However, speedup opportunities may also be exploited by running multiple operations in parallel through independent subgraphs. Therefore, we also consider independent subgraphs in ISEGEN.

When the goal of ISE generation is speedup coupled with dynamic reuse, as in [6, 7, 8], the resulting subgraphs are generally small. In practice, if one wants to mimic the excellent results targeted by expert designers, clusters of 2 or 3 instructions are far too small for arousing real interest: typical results at this level generally include only peculiar address generation patterns, pre- or post-shifting, or well-known arithmetic patterns such as multiply-accumulators. One should also note that such compound instructions often exist already in embedded processors such as ARM and thus nullify the interest of such ISEs. So, there is a need to expand the reach of reusability by focussing on large reusable clusters. The resulting reusable ISEs can be used effectively to cover a larger portion of the DFG than a large non-reusable ISE thereby leading to higher speedup. This motivates our ISEGEN approach that not only generates ISEs having higher potential for speedup, but which also shows the efficacy of the generated ISEs in terms of their reusability.

An exact (optimum) solution [3] that uses an exhaustive search with pruning is not practical for applications having large basic-blocks. On the other hand, starting from a seed node to grow clusters having desirable properties [5] is a naive solution because a clustering solution is typically used only to induce a more tractable problem instance [9]. A more sophisticated approach for ISE generation is a genetic formulation [4] that serves as a practical solution with results showing good speedup for the generated ISEs. However, the solution space-walker visits the solutions stochastically and therefore it is hard to predict if the solution quality has improved enough to terminate the search. Moreover, the solutions obtained on different runs may be different because of the stochastic nature of the algorithm. Our ISEGEN approach, on the other hand is an iterative refinement technique where each iteration improves the quality of solution, with the search terminated as soon as any iteration fails to show improvement. Besides being more predictable in its approach, ISEGEN addresses more design concerns than the genetic algorithm in its gain function; consequently we are able to match the quality of an expert designer in generation of ISEs.

3. PROBLEM DEFINITION

At the instruction-level-granularity, the underlying data-structure is a DFG, $G(V, E)$; the nodes V represent instructions and the edges E capture the data dependencies between the nodes. Within a basic block, the graph G is essentially a Directed Acyclic Graph (DAG). We define a *cut* C to be a subgraph of $G (C \subseteq G)$ representing a potential ISE. Let $\mathrm{M}(C)$ be the function that measures the merit of a cut C as an estimation of the speedup achievable by implementing C as an ISE. Note that ISEGEN algorithm does not depend on the definition of $\mathrm{M}(C)$.

Let $\mathrm{IN}(C)$ be the number of predecessor nodes of those edges which enter the cut C from the rest of the graph G. They represent the number of input values used by the operations in C. Similarly, let $\mathrm{OUT}(C)$ be the number of predecessor nodes in C of edges exiting the cut C. They represent the number of values produced by C and used by other operations, either in G or in other basic blocks.

The problem of ISE generation can be broken into the following two sub-problems to be solved bottom-up:

PROBLEM 1. *Given the DFG $G(V, E)$ in a basic block, find a cut $C \subseteq G$ that maximizes $\mathrm{M}(C)$ under the following constraints:*

- *Input-Output (I/O) Constraints:* $\mathrm{IN}(C) \leq N_{\mathrm{in}}$ *and* $\mathrm{OUT}(C) \leq N_{\mathrm{out}}$. *These constraints ensure that the generated ISE is valid without changing the register file configuration of the given architecture.*
- *Convexity Constraints: C is convex, i.e., if there exists no path from a node $u \in C$ to another node $v \in C$ through a node $w \notin C$. If the cut is not convex, the input operands of the ISE respresented by the cut will not be available at the time of issue. The validity of this constraint certifies that the implementation of the cut is architecturally feasible. A non-convex cut can, in principle, be scheduled but it would greatly increase the complexity of the compiler.*

PROBLEM 2. *Given the basic blocks in an application and the maximum allowed number of ISEs as N_{ISE}, find cuts that maximize the speedup achievable for the entire application.*

The ISE represented by a cut C runs on an AFU. We envisage the AFUs to reside in a Field Programmable Gate Array (FPGA) fabric. Because of limitations on the maximum number of dedicated links allowed between the AFUs and the core, the process of ISE generation is constrained by the maximum number of AFUs (i.e., ISEs) that can be realized.

4. THE ISEGEN HEURISTIC

We reiterate that ISEGEN essentially performs Hardware-Software partitioning at instruction-level granularity. The instructions selected as an ISE to be mapped to hardware execute on an AFU while those mapped to software execute on the main processor. Each node in a basic block is assigned an *ISE-index* indicating its mapping with an AFU. For a node lying in the software part, the *ISE-index* is 0. The maximum value of the *ISE-index* is the allowed number of ISEs (N_{ISE}) in the system. The heuristic performs at the most N_{ISE} successive bi-partitions of the DFG into hardware and software that maximize speedup under microarchitectural constraints. The basic blocks are considered in decreasing order of their execution frequencies.

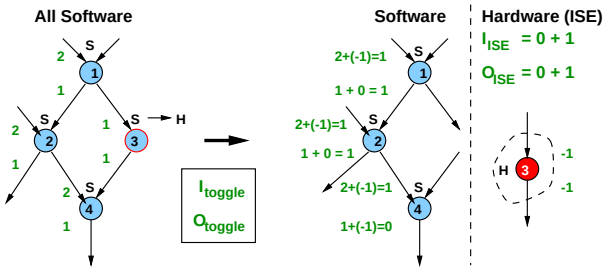

Figure 1: Instance of a Hardware-Software Partitioning

We borrow the idea from *Kernighan-Lin* min-cut partitioning heuristic to steer "toggling" of nodes in the DFG between software (S) and hardware (H) based on a gain function that captures the designer's objective. The effectiveness of the K-L heuristic lies in its ability to overcome many local maxima without using unnecessary moves. Let the number of inputs and the number of outputs of ISE at any stage of the partitioning process be denoted by I_{ISE} and O_{ISE} respectively. In order to quantify the effect of toggling a node, we introduce **addendums** I_{toggle} and O_{toggle} associated with every node. When a node is toggled, its addendums I_{toggle} and O_{toggle} are added to I_{ISE} and O_{ISE} respectively to get the new values of I_{ISE} and O_{ISE}. Initially, all the nodes are in S and therefore $I_{\text{ISE}} = O_{\text{ISE}} = 0$ and I_{toggle} and O_{toggle} equal the number of inputs and number of outputs respectively of the corresponding node.

When a node is toggled, the I_{toggle} and O_{toggle} values of its neighbors (parents, children and siblings) may get affected as illustrated in Figure 1. When node 3 is toggled from S to H, its addendums are added to I_{ISE} and O_{ISE} respectively to reflect the number of inputs and outputs in the resulting ISE. After toggling, I_{toggle} and O_{toggle} of node 3 reverse in sign showing that the changes to I_{ISE} and O_{ISE} will be undone if node 3 toggles back to S-node. It is easy to verify that the changed values of I_{toggle} and O_{toggle} for the neighbors of node 3 correctly account for the new values of I_{ISE} and O_{ISE} when any of these nodes is toggled. For example, if node 1 is toggled next, $I_{\text{ISE}} = 1 + 1 = 2$ and $O_{\text{ISE}} = 1 + 1 = 2$ with ISE containing nodes 1 and 3.

4.1 Modified Kernighan-Lin Algorithm

We now present our ISEGEN heuristic in Algorithm 1. This is an iterative refinement algorithm that starts with all nodes in software and tries to toggle each unmarked node, n, in the graph from S to H or H to S in every iteration. Initially, the best cut, *best_C* points to a configuration where all nodes belong to software. The decision to toggle n with respect to a cut is based on a gain function, $M_{\text{toggle}}()$. The gain function is evaluated for each node and the node with the best gain, *best_n* is extracted using function *GetMaxGainNode()*, and is then toggled and marked. Note that the chosen cut at this point may be violating input/output constraints and convexity constraints. In other words, we allow a cut to be illegal giving it an opportunity to grow eventually into a valid cut.

The function *SatisfiesConstraints()* checks if both convexity and I/O constraints are satisfied. If so, *last_best_C* is updated using function *ModifyCut()* that either removes *best_n* from the cut or adds it to the cut depending on whether *best_n* has toggled from H to S or S to H respectively. The cut *last_best_C* and its merit M() stored in *last_best_gain* are passed on to the next iteration for further refinement. This process is carried on till no more unmarked nodes are left. The best cut (*best_C*) corresponding to an *ISE-index* is stored back in C that will act as a starting point for finding the next cut identified by *(ISE-index + 1)*.

Algorithm 1 *ISEGEN()*

```
I_ISE ⇐ 0; O_ISE ⇐ 0;
∀n ∈ nodes(DFG), I_toggle[n] = I[n]; O_toggle[n] = O[n];
best_C ⇐ last_best_C ⇐ C;
best_gain ⇐ last_best_gain ⇐ −∞;
while # passes < 6 do
  C ⇐ last_best_C;
  while ∃ unmarked node in DFG do
    foreach unmarked node n ∈ nodes(DFG) do
      GainArray[n] ⇐ M_toggle(n,last_best_C);
    end for
    best_n ⇐ GetMaxGainNode(GainArray);
    ToggleAndMark(best_n);
    CalcImpOfTogg(best_n, last_best_C);
    if SatisfiesConstraints(best_n, last_best_C) then
      last_best_C ⇐ ModifyCut(best_n,last_best_C);
      last_best_gain ⇐ M(last_best_C);
      last_best_I_ISE ⇐ I_ISE; last_best_O_ISE ⇐ O_ISE;
    end if
  end while
  if last_best_gain > best_gain then
    best_C ⇐ last_best_C; best_gain ⇐ last_best_gain;
    I_ISE ⇐ last_best_I_ISE; O_ISE ⇐ last_best_O_ISE;
    UnmarkAllNodes();
  end if
end while
C ⇐ best_C;
```

Algorithm 2 *CalcImpOfTogg(best_n, last_best_C)*

```
I_ISE += I_toggle[best_n]; O_ISE += O_toggle[best_n];
I_toggle[best_n] ⇐ - I_toggle[best_n];
O_toggle[best_n] ⇐ - O_toggle[best_n];
foreach m ∈ Neighbors[best_n] do
  Apply Rules for updating I_toggle[m] and O_toggle[m];
end for
Maintain appropriate data structures;
```

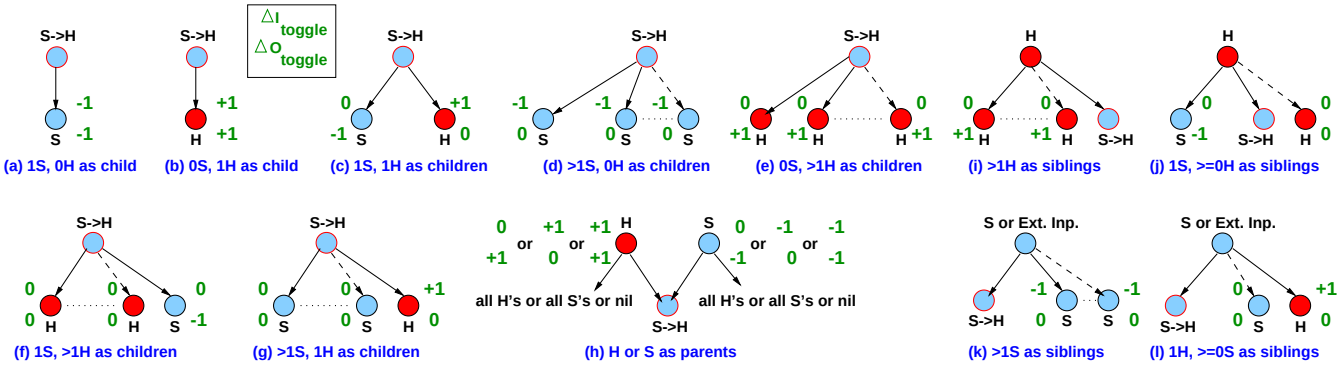

Figure 2: Basic Rules to project the effect of toggling a node from S to H to its parents (h), children (a-g) and siblings (i-l). $\{\Delta I_{\text{toggle}}, \Delta O_{\text{toggle}}\}$ pair shown associated with each node that get affected.

4.2 Gain Function

The gain function M_{toggle} is designed to estimate the gain of toggling a node after careful examination of goals that are of interest to an experienced designer. This gain function has 5 goals: (1) to maximize the speedup exhibited by the chosen cut, (2) to satisfy the input-output port constraints, (3) to satisfy the convexity constraints, (4) to favor generation of large cuts and (5) to enable search for independent connected components if they have higher speedup potential. Accordingly, the gain function for toggling a node n with respect to the current cut C is made up of following 5 components that act as control parameters for the algorithm:

- **Merit Function (Speedup Estimate)**: Let C' be the new cut after addition or removal of the node n from the cut, C as n toggles from S to H or H to S respectively.

$$merit = \begin{cases} \text{M}\left(C'\right), \text{C' obeys convexity constraint,} \\ -\infty, \text{C' violates convexity constraint.} \end{cases}$$

- **Input Output violation penalty**: A heavy penalty is applied with the help of a large factor if input-output port constraints are violated.

$$iop = \left(\left(I'_{\text{ISE}}\left(n\right) - N_{\text{in}}\right) + \left(O'_{\text{ISE}}\left(n\right) - N_{\text{out}}\right)\right),$$

$$I'_{\text{ISE}}\left(n\right) = I_{\text{ISE}} + I_{\text{toggle}}[n];$$

$$O'_{\text{ISE}}\left(n\right) = O_{\text{ISE}} + O_{\text{toggle}}[n].$$

- **Convexity Constraints**: Addition of a node to a cut is favored when its neighbors are already in the cut while a node already in the cut is not easily discarded.

$$conv = \begin{cases} +num\_nbrs\left(n, C\right), \text{if n is in } S, \\ -num\_nbrs\left(n, C\right), \text{if n is in } H. \end{cases}$$

- **Large Cut**: A cut is allowed to grow in regions where growth potential is higher. The external input and external output nodes act as barriers beyond which a cut cannot grow. Since we do not allow memory access from AFUs, memory operations are also barriers for cut growth. Let $d\_to\_bars\_up\left(n\right)$ be the minimum distance of n from external inputs and memory barriers in the upward direction and let $d\_to\_bars\_down\left(n\right)$ be the minimum distance of n from external outputs and memory barriers in the downward direction.

$$cgp = \begin{cases} +|d\_to\_bars\_up\left(n\right) - d\_to\_bars\_down\left(n\right)|, \\ \qquad \text{if n is in } S, \\ -|d\_to\_bars\_up\left(n\right) - d\_to\_bars\_down\left(n\right)|, \\ \qquad \text{if n is in } H. \end{cases}$$

The nodes closer to the barriers have higher potential for cut growth and therefore are consistently favored for inclusion in hardware. This component plays a pivotal role in driving the search into isomorphic regions of solution space. Thus, without losing the benefit of having large cut as a solution, it implicitly favors reusability of the cut.

- **Independent Cuts**: It is clear that owing to favoring the growth of large cuts, the connected nodes can be effectively explored in the vertical direction. However, it is quite possible that the best cut is actually a combination of two or three large connected subgraphs and not necessarily the largest connected subgraph satisfying I/O constraints. So, the ISE exploration needs to also expand in the horizontal direction. Let $CS(G)$ be the connected subgraphs in the DFG G excluding the connected subgraph containing n.

$$indp\_cuts = \begin{cases} +max_{cs \in CS(G)} CP\_lat\left(cs\right), \\ \qquad \text{if n is in } H, \\ 0, \text{if n is in } S. \end{cases}$$

where, $CP\_lat\left(cs\right)$ is the sum of the hardware latencies along the critical path of the connected subgraph, cs. Using this component, the nodes already in H are allowed to move back into S to favor the growth of other potentially large subgraphs.

We now express $M_{\text{toggle}}\left(n\right)$ with respect to the current cut C as follows:

$$\alpha_1 \cdot merit - \alpha_2 \cdot iop + \alpha_3 \cdot conv + \alpha_4 \cdot cgp + \alpha_5 \cdot indp\_cuts$$

The weights α_1, α_2, α_3, α_4 and α_5 have been determined experimentally. It is to be noted here that the genetic algorithm does not consider the last three components of

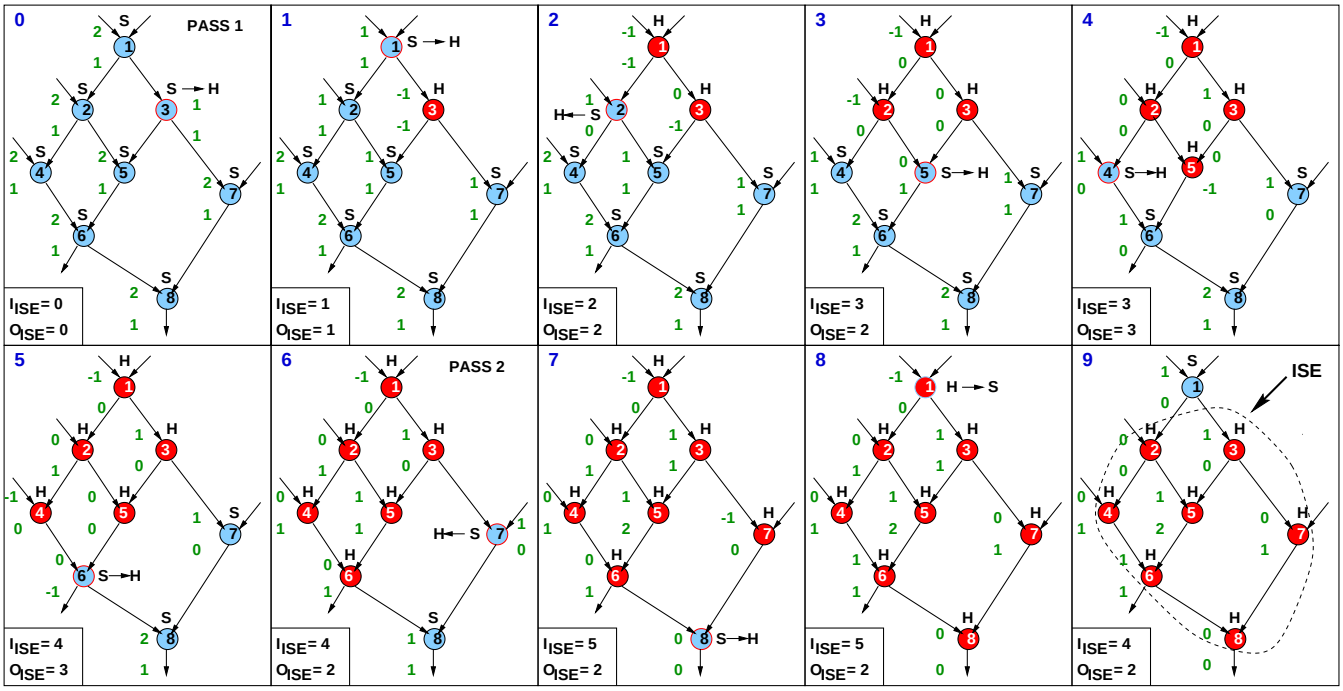

Figure 3: Running Example: The nodes are annotated with I_{toggle} and O_{toggle} values. Note that when a node is toggled (from S to H or H to S), the addendums of the node along with those of its neighbors change. The steps 1 through 5 are executed in PASS 1 and steps 6 through 9 in PASS 2 with the generated ISE shown in step 9.

M_{toggle} in its fitness function. We show in [10] that by maintaining appropriate data structures, the worst-case running time of ISEGEN can be restricted to $O(|V| \cdot |E|)$. The runtime complexity of M_{toggle} has been significantly reduced by transferring majority of computations into the function, $CalcImpOfTogg()$.

4.3 Effect of Toggling a Node

The function $CalcImpOfTogg()$ (Algorithm 2) captures the effect of toggling a node, which primarily consists of changes in I_{toggle} and O_{toggle}. We developed a comprehensive set of rules to capture the effect of toggling a node that is pictorially presented in Figure 2. The changes in I_{toggle} and O_{toggle} values are represented as ΔI_{toggle} and ΔO_{toggle} respectively such that the new values of I_{toggle} and O_{toggle} for the affected nodes are computed as $(I_{\text{toggle}} + \Delta I_{\text{toggle}})$ and $(O_{\text{toggle}} + \Delta O_{\text{toggle}})$ respectively. The toggle of a node from S to H is shown in Figure 2 as $S \rightarrow H$. Following rules are integrated inside the function $CalcImpOfToggle()$.

RULE 1. *After toggling, I_{toggle} and O_{toggle} values for a node reverse in sign.*

RULE 2. *If a node* n *is toggled from* S *to* H *or* H *to* S, *I_{toggle} and O_{toggle} of only the parents, children and siblings can get affected.*

RULE 3. *If a node* n *is toggled from* S *to* H *and is a parent of one or more nodes, then the addendums for the children change according to the following rules (Figure 2(a-g)):*

1. *If* n *has one and only one child, then*
   (a) *If the child is an S-node, $\Delta I_{\text{toggle}} = \Delta O_{\text{toggle}} = -1$ for the child (Figure 2(a)).*
   (b) *If the child is an H-node, $\Delta I_{\text{toggle}} = \Delta O_{\text{toggle}} = +1$ for the child (Figure 2(b)).*
2. *If* n *has $\geq$ 2 children, then*
   (a) *If there are exactly 2 children, of which one is an S-node and the other is an H-node, then for the S-node, $\Delta O_{\text{toggle}} = -1$ and for the H-node, $\Delta I_{\text{toggle}} = +1$ (Figure 2(c)).*
   (b) *If all the children are S-nodes, then for the S-nodes, $\Delta I_{\text{toggle}} = -1$ (Figure 2(d)).*
   (c) *If all the children are H-nodes, then for the H-nodes, $\Delta O_{\text{toggle}} = +1$ (Figure 2(e)).*
   (d) *If only one child is an S-node and > 1 nodes are H-nodes, then for the S-node, $\Delta O_{\text{toggle}} = -1$ (Figure 2(f)).*
   (e) *If only one child is an H-node and > 1 nodes are S-nodes, then for the H-node, $\Delta I_{\text{toggle}} = +1$ (Figure 2(g)).*

RULE 4. *If a node* n *is toggled from* S *to* H *and is a child of a node* m, *then the addendums for* m *change according to the following rules (Figure 2(h)):*

1. *If* m *is an S-node, then*
   (a) *If* m *has no other children, $\Delta I_{\text{toggle}} = -1$ and $\Delta O_{\text{toggle}} = -1$.*
   (b) *If* m *has some children as S-nodes, $\Delta I_{\text{toggle}} = -1$.*

(c) *If* m *has all other children as H-nodes,* $\Delta O_{\text{toggle}} = -1$.

2. *If* m *is an H-node, then*

   (a) *If* m *has no other children,* $\Delta I_{\text{toggle}} = +1$ *and* $\Delta O_{\text{toggle}} = +1$.

   (b) *If* m *has some children as S-nodes,* $\Delta I_{\text{toggle}} = +1$.

   (c) *If* m *has all other children as H-nodes,* $\Delta O_{\text{toggle}} = +1$.

RULE 5. *If a node* n *is toggled from* S *to* H, *then the addendums for its sibling nodes change according to the following rules(Figure 2(i-l)):*

1. *If the parent of* n *is an H-node, then*

   (a) *If the siblings are all H-nodes,* $\Delta O_{\text{toggle}} = +1$ *(Figure 2(i)).*

   (b) *If only one sibling is an S-node and rest may be H-nodes, for the S-node,* $\Delta O_{\text{toggle}} = -1$ *(Figure 2(j)).*

2. *If the parent of* n *is an S-node or an external input, then*

   (a) *If the siblings are all S-nodes,* $\Delta I_{\text{toggle}} = -1$ *(Figure 2(k)).*

   (b) *If only one sibling is an H-node and rest may be S-nodes, for the H-node,* $\Delta I_{\text{toggle}} = +1$ *(Figure 2(l)).*

RULE 6. *The toggle of a H-node negates the effect of its toggling from S. This implies that all the above rules can be applied for toggling from* H *to* S *with the sign reversed for the values of* ΔI_{toggle} *and* ΔO_{toggle}.

These rules can be empirically verified to work on any DFG (for example, Figure 1). The proofs of correctness for the rules have been omitted for the sake of brevity and presented in [10]. The maintenance of appropriate data structures for fast evaluation of M() and convexity violation is also discussed in [10].

4.4 Running ISEGEN

We demonstrate the application of rules while running ISEGEN algorithm with a simple example shown in Figure 3. With $N_{\text{in}} = 4$ and $N_{\text{out}} = 2$, the solution is obtained in just 2 passes. At the end of the first pass, a valid solution is obtained which is improved further in the next pass. The PASS 3 does not show any further improvement in $\text{M}(C)$ and therefore the search is terminated. Note that the passes have to incur violation of constraints in their intermediate steps (steps 4, 5, 7 and 8) before converging to a valid solution and thus our ISEGEN approach is able to get over the points of local maxima. In general, we found experimentally that 5 passes are enough for successive refinement of the solution.

5. EXPERIMENTS

We define the merit function as: $\text{M}(C) = \lambda_{sw}(C) - \lambda_{hw}(C)$, where $\lambda_{\text{sw}}(C)$ estimates the software latency of C as the accumulated latencies of the nodes in C; $\lambda_{\text{hw}}(C)$ estimates its hardware latency by summing hardware latencies of instructions in the critical path of C. The hardware latency for each instruction was obtained by synthesizing the constituent arithmetic and logic operators on a common $0.18\mu m$ CMOS technology and then normalized to the delay of a 32-bit *multiply-accumulate (MAC)*.

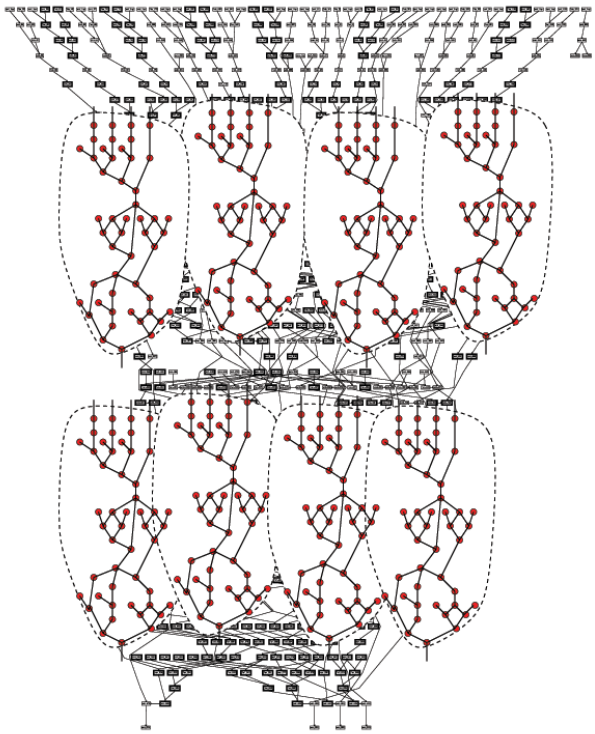

Figure 4: First cut (having 8 instances) generated for AES under I/O Constraints (4,1). Each instance of the cut contains 49 nodes covering about 60% of the DFG.

We integrated ISEGEN in the MachSUIF framework [14] and evaluated overall speedup for the cuts ($CUTS$) chosen by ISEGEN using the following expression:

$$\frac{\lambda_{\text{overall}}}{\lambda_{\text{overall}} - \sum_{C \subset CUTS} N_C \cdot \text{M}(C)}$$

The variable, λ_{overall} encapsulates the overall execution latency of the application i.e., when the application entirely runs on software and N_C is the execution frequency of C. Note that in this work, we do not consider memory operations for inclusion into a cut.

To evaluate the efficacy of our ISEGEN approach, we ran our experiments on benchmarks from diverse application domains in EEMBC (*autcor00*, *viterbi00*, *conven00*, *fft00*, *fbital00* and *ospf*) and MediaBench (*adpcm coder* and *adpcm decoder*) suites. In addition, we chose a cryptographic application viz. *AES*. Since *AES* is a large example exhibiting a regular structure, we describe those results separately later. Our baseline architecture is a simple RISC machine and we allow upto 4 AFUs (or ISEs) to be added. Keeping the I/O constraints fixed at (4,2), we present the

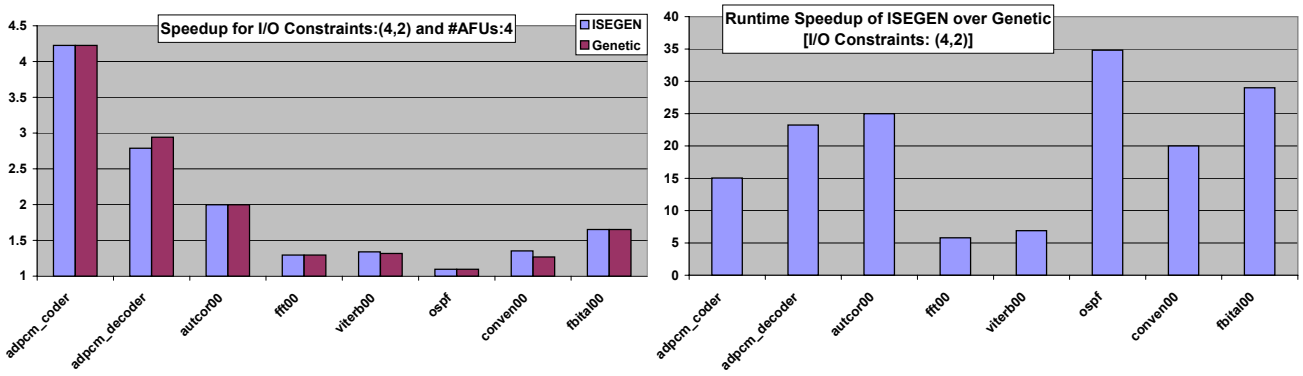

Figure 5: Comparison of Speedup and Runtime with number of AFUs = 4 and I/O constraints: (4,2)

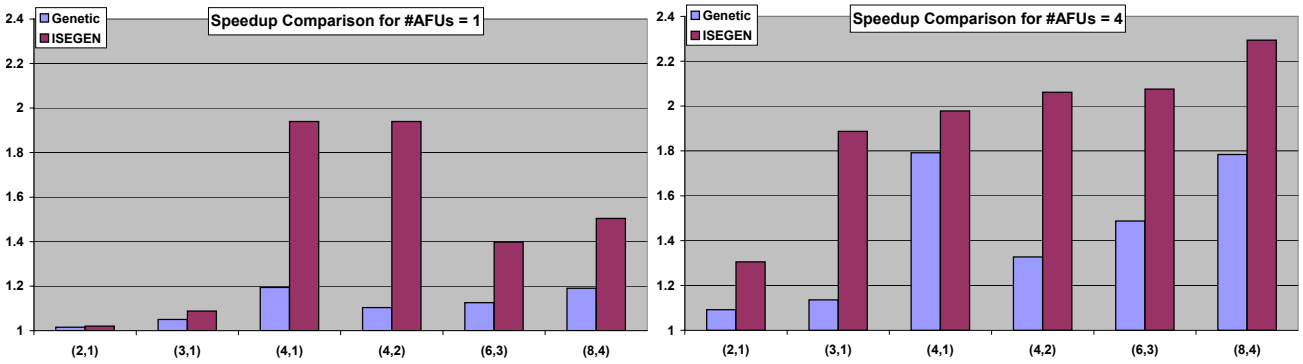

Figure 6: Comparison of Speedup on AES with varying number of AFUs

speedups obtained over execution on the core processor and the corresponding runtime improvement with respect to a previous genetic formulation [4] in Figure 5. These plots show that our ISEGEN closely matches the quality of genetic solutions for the benchmarks lacking a regular structure, while having a much quicker response time. On an average, ISEGEN exhibits more than 20X faster turn-around time than the genetic algorithm and obtains an average speedup of 2X over execution on the core processor.

Experiments with AES

Because of its non-exponential complexity, ISEGEN easily handles large DFGs. For instance, *AES* is a cryptographic benchmark with a large DFG; its critical basic block contains 696 nodes with a symmetric structure. While designers are manually able to identify large clusters, previous ISE approaches are not able to consistently generate good results. We deliberately chose *AES* to demonstrate the efficacy of our ISEGEN approach in matching manual expert design quality. We increased the maximum number of AFUs from 1 to 4, and studied the speedup over execution on the processor core as shown in Figure 6. On an average, ISEGEN obtains 35% more speedup than the genetic solution by effectively exploiting the regularity in the data flow graph of AES. Figure 7 shows how the structure yielded multiple instances of the same cut thereby exposing the regularity in the application.

Since AES has a large number of nodes, it is intuitive to expect an increase in speedup by increasing the allowed number of AFUs and I/O constraints. However, it is interesting to note that contrary to our expectation, for a smaller number of allowed AFUs (= 1), the speedup could not scale with relaxing I/O constraints (as shown in the first plot of Figure 6). The reason is clear from the plot of Figure 7. It shows that there are 12 instances of the first cut for the I/O constraint of (4,1) (or (4,2)), while there are only 4 instances for the I/O constraint of (6,3). As is evident from the first plot of Figure 6, 12 instances generated for I/O constraint of (4,1) covers the DFG better than the 4 instances generated for (6,3). However, with increase in the allowed number of AFUs, the speedup begins to scale with increasing I/O constraints (as shown in the last plot of Figure 6).

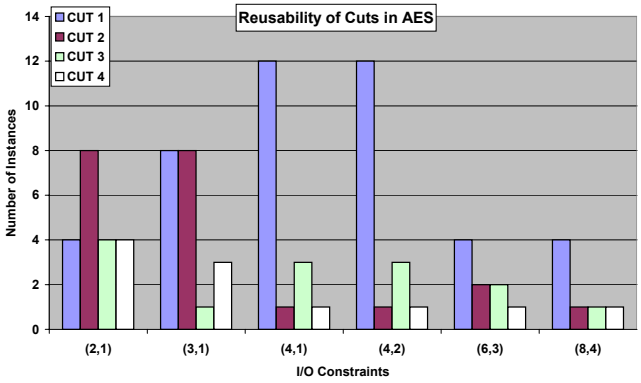

Figure 7: Study of Reusability of ISEs on AES with varying number of AFUs

A careful study of first cut generated by ISEGEN for an I/O constraint of (4,1) revealed that it has 12 instances (8 in one basic block and 4 in another), each containing on the order of 50 nodes. Figure 4 shows that there are 8 instances of the same cut (with an I/O constraint of (4,1)) covering 400 out of the 696 nodes (i.e., about 60% of the DFG) and all the instances were found by ISEGEN in the first cut. Therefore, our ISEGEN not only generates ISEs resulting in high speedup but also exploits their reusability by producing all the instances in the DFG (as shown in Figure 7. Thus, the solutions generated by ISEGEN are indeed close to those generated by an expert designer by manual inspection of the application DFG. Furthermore, ISEGEN through its iterative refinement strategy is deterministic in obtaining good solution consistently.

6. CONCLUSIONS

Automatic generation of Instruction Set Extensions (ISEs) to be executed on Ad-hoc Functional Units (AFUs) is a key step for running performance-critical application on a programmable processor. The hardware-software partitioning problem when applied at the instruction-level granularity constitutes the problem of ISE generation. Three main contributions were presented in this paper. First, we clearly identified the properties of ISEs that are of interest to an expert designer. Second, we adapted a well-known Kernighan-Lin heuristic to perform hardware-software partitioning at the instruction-level granularity with minimal computational complexity. The control parameters used for guiding the K-L heuristic were based on the desired properties of the ISEs. Finally, we show that our ISEGEN approach produces high-quality ISEs — close to those sought after by an expert designer. For applications having a regular structure, ISEGEN results in 35% higher speedup than the genetic algorithm by generating reusable ISEs. Furthermore, it runs 20X faster than a previous genetic algorithm and generates comparable high-quality solutions for applications lacking a regular structure. ISEGEN achieved an overall speedup on the order of 2X on a number of EEMBC, MediaBench and cryptographic benchmarks. Even though we applied our heuristic using recursive bipartitioning of the graph, it is possible to adapt our work to a multiway partitioning approach [13]. Future work will also study the incorporation of memory elements in our ISE generation approach.

7. REFERENCES

[1] F. Vahid and T. D. Le. Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning. In *Kluwer Journal on Design Automation of Embedded Systems*, 1997.

[2] C. M. Fiduccia and R. M. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of Design Automation Conference (DAC)*, 1982.

[3] K. Atasu, L. Pozzi and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *Proceedings of Design Automation Conference (DAC)*, 2003.

[4] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt. Introduction of Local Memory Elements in Instruction Set Extensions. In *Proceedings of Design Automation Conference (DAC)*, 2004.

[5] N. Clark, H. Zhong and S. Mahlke. Processor Acceleration through Automated Instruction Set Customization. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[6] F. Sun, S. Ravi, A. Raghunathan and N. K. Jha. Synthesis of Custom Processors based on Extensible Platforms. In *Proceedings of International Conference on Computer Aided Design (ICCAD)*, 2002.

[7] M. Arnold and H. Corporaal. Designing Domain-specific Processors. In *Proceedings of International Conference on Hardware Software Codesign (CODES)*, 2001.

[8] H. Choi, J. S. Kim, C. W. Yoon, I. C. Park, S. H. Hwang and C. M. Kyung. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Transactions on Computers*, 1999.

[9] C. J. Alpert and A. B. Kahng. Recent Developments in Netlist Partitioning: A Survey. *Integration: the VLSI Journal*, 19(1-2), 1995, pp. 1-81.

[10] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi and P. Ienne. ISEGEN: Adapting Kernighan-Lin Min-Cut Heuristic for Generation of Instruction Set Extensions. CECS, UC Irvine, Technical Report CECS-TR-04-21.

[11] C. Alippi, W. Fornaciari, L. Pozzi and M. Sami. A DAG based Design Approach for Reconfigurable VLIW Processors. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 1999.

[12] R. Razdan and M. D. Smith. A High-performance Microarchitecture with Hardware-programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, 1994.

[13] G. Karypis and V. Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *SIAM Review*, 41 (1999) 278-300.

[14] Machine SUIF. `http://www.eecs.harvard.edu/hube/software/software.html`.