ISEGEN: Adapting Kernighan-Lin Min-Cut Heuristic for Generation of Instruction Set Extensions

Partha Biswas, Sudarshan Banerjee, Nikil Dutt, Laura Pozzi[*], and Paolo Ienne[†]

CECS Technical Report #04-21
Center for Embedded Computer Systems School of Information and Computer Science
University of California, Irvine, CA 92697, USA

Aug 12, 2004

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 achieved 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. In this report, we present ISEGEN, an approach that identifies high-quality ISEs by iterative improvement following 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 approach matches the quality of the optimal solution obtained by exhaustive search. We also show that our ISEGEN technique is on average* $20\times$ *faster than a genetic formulation that generates equivalent solutions. Furthermore, the ISEs identified by our technique exhibit* 35% *more speedup than the genetic solution on a large cryptographic application (AES) by effectively exploiting its regular structure.*

[*]Laura Pozzi is affiliated with University of Lugano, Faculty of Informatics, CH-6900 Lugano, Switzerland

[†]Paolo Ienne is affiliated with Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland

Contents

List of Figures

1 Introduction

Continuing advances in manufacturing processes have made it possible for processor vendors to build increasingly fast processors. However, newer applications place an increasing demand on performance, at a rate faster than that achievable by processors. Furthermore, application requirements are also changing continuously. These trends have necessitated the migration of critical computations from the processor core 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)*. A conceptual representation of a processor subsystem with AFUs partaking in the processor pipeline is presented in Figure 1. The AFU accelerates critical operations of application algorithms by executing application-specific *Instruction Set Extensions (ISEs)*.

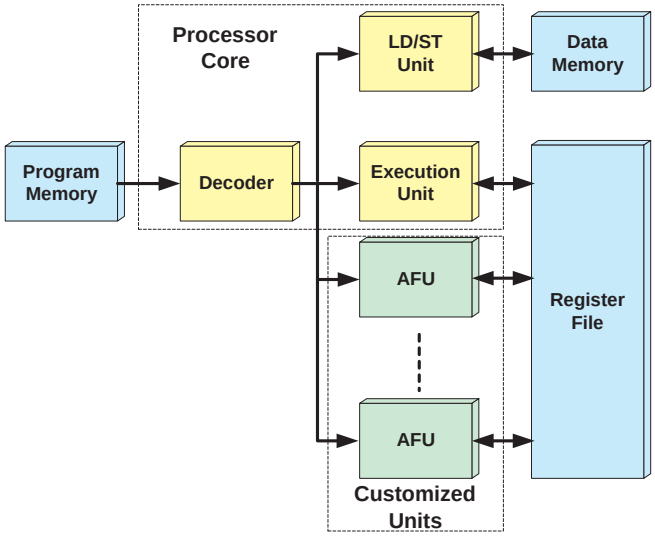


Figure 1. A Processor Subsystem with AFUs Tightly-coupled to the Processor Core

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 report, we apply the K-L heuristic at the instruction-level granularity to automatically generate ISEs. We refer to our approach as ISEGEN. Our motivation for employing an iterative improvement technique like K-L is to generate solutions close to those obtained manually by expert designers. In order to match such a solution quality, the control parameters of ISEGEN closely model the decisions taken by the designer. One of the main challenges in applying the K-L heuristic is to maintain low computational complexity in its critical section in order to achieve a fast turnaround time.

We show the efficacy of ISEGEN on a number of embedded applications selected from MediaBench, EEMBC and cryptographic suites by comparing our results with the best known approaches of ISE generation. We demonstrate that ISEGEN runs up to $29\times$ faster than the previous genetic formulation while yielding ISEs having speedup comparable with the optimal solution [8]. On a large cryptographic application (AES) for which the exhaustive techniques fail, ISEGEN — by effectively exploiting its regular structure — generates 35% more speedup than the genetic approach.

The rest of the report is organized as follows. In Section 2, we define our problem. In Section 3, we discuss related research work and our motivation. We propose our ISEGEN approach in Section 4. In

Section 5, we describe the experimental results that demonstrate the efficacy of our approach. Finally, Section 6 concludes the report.

2 Problem Definition

Instructions within a basic block are typically represented as a Directed Acyclic Graph (DAG), $G = (V, E)$: the nodes V represent instructions and the edges E capture the data dependencies between them. We define a *cut* C representing a potential ISE as a subgraph of G, $C \subseteq G$ 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. Let $I_{\mathrm{ISE}}(C)$ and $O_{\mathrm{ISE}}(C)$ respectively be the number of inputs and the number of outputs of C. The maximum number of operands of an ISE (or a cut) is limited by the number of register file ports in the underlying core.

Let N_{in} and N_{out} be the maximum number of input and output operands respectively. A cut C is architecturally feasible if its inputs are available at the time of issue. This is only possible if 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 $w \notin C$ [8]. The problem of ISE generation can be broken into the following two sub-problems:

Problem 1 *Given the data flow graph (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: $I_{\mathrm{ISE}}(C) \leq N_{\mathrm{in}}$ and $O_{\mathrm{ISE}}(C) \leq N_{\mathrm{out}}$.*
- *Convexity Constraint: C is convex.*

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.*

3 State of the Art and Motivation

The process of ISE generation involves clustering of simple operations of an application into an ISE that maps to specialized hardware. These ISEs capture the compute-intensive sections of the application. Because of similarity to high-level synthesis, the solutions to the problem of ISE generation are motivated from early CAD algorithms for component library mapping. The idea of clustering operations in ISE generation is similar to the concept of regularity extraction [3, 4, 5] and template matching [6, 7] that are used in a variety of CAD algorithms with the goal of increasing performance and reducing area under timing constraints.

The problem of ISE generation for application specific processors has been studied for almost a decade. Some of the earlier work in ISE generation applied to reconfigurable computing [17, 16] considers only single-output subgraphs in ISE generation. Even though a few recently proposed approaches [10, 11] handle multiple outputs, they identify only connected subgraphs. However, the opportunity to include independent subgraphs in the same ISE exposes speedup potentials, while algorithms identifying only connected graphs are unable to exploit high constraints of ISE outputs. Therefore, we also consider independent subgraphs in ISEGEN.

When the goal of ISE generation is speedup coupled with dynamic reuse, as in [14, 12, 13, 15], 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

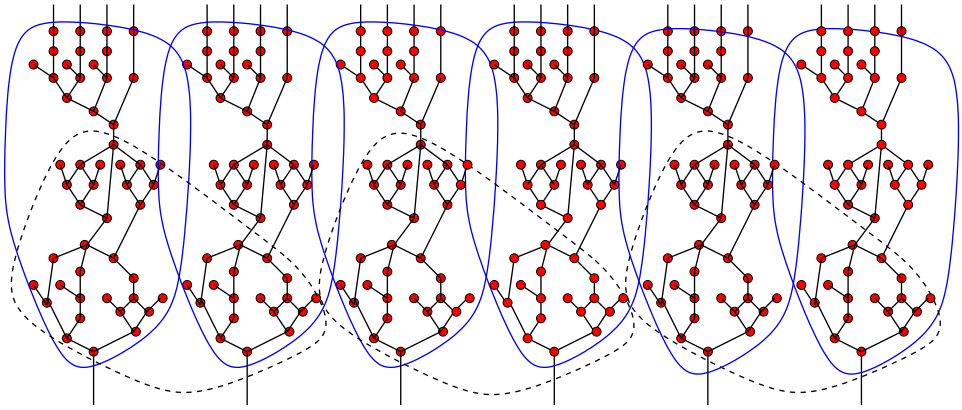


Figure 2. An example showing the advantage of large scale reuse — Finding three instances of the largest ISE (shown with a dotted boundary) is not as effective as finding a large ISE with six instances (shown with a solid boundary).

this level generally include only peculiar address generation patterns, pre- or post-shifting, or well-known arithmetic patterns such as multiply-accumulators. There is a need for algorithms that can identify large *and* reusable clusters, efficiently covering the application DFG. Figure 2 demonstrates this principle with the help of an example. 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 solution [8] that uses an exhaustive search with pruning is not practical for applications having large basic-blocks. A genetic formulation [9] presents a practical solution with results showing good speedup for the generated ISEs. However, the genetic algorithm is stochastic in nature and therefore multiple runs may result in different solutions. Our ISEGEN approach, on the other hand, is an iterative improvement technique that closely mimics the decisions taken by an expert designer; consequently we are able to match the solution quality of expert designers.

4 The ISEGEN Approach

We reiterate that ISEGEN essentially performs Hardware-Software partitioning at instruction-level granularity. The instructions belonging to the hardware partition map to an ISE to be executed on an AFU while those belonging to the software partition individually execute on the processor core. Our approach considers the basic blocks in an application based on their speedup potential — a function of its execution frequency and estimated gain from mapping all its nodes to hardware — and performs up to N_{ISE} successive bi-partitions into hardware and software within a basic block. After an ISE is found in a basic block, the speedup potential of the block is updated considering the remaining nodes.

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.

ISEGEN()

00: $SetInitialConditions$()
01: $last\_best\_C \Leftarrow C$
02: **loop** (until exit condition)
03: $best\_C \Leftarrow last\_best\_C$
04: **while** (there exists unmarked node in DFG)
05: **foreach** (unmarked node n)
06: Calculate M_{toggle}(**n,$best\_C$**)
07: **endfor**
08: $best\_node \Leftarrow$ Node with maximum Gain
09: **Toggle** and **Mark** $best\_node$
10: $CalcImpactOfToggle$($best\_node$,$best\_C$)
11: **if** (toggling $best\_node$ satisfies constraints)
12: Update $best\_C$ from toggling $best\_node$
13: Calculate M($best\_C$)
14: **endif**
15: **endwhile**
16: **if** (M($best\_C$) $>$ M($last\_best\_C$))
17: $last\_best\_C \Leftarrow best\_C$
18: Unmark all nodes
19: **endif**
20: **endloop**
21: $C \Leftarrow last\_best\_C$

Figure 3. The ISEGEN Algorithm

4.1 ISEGEN: A Modified Kernighan-Lin Algorithm

The ISEGEN algorithm that essentially performs a bi-partitioning of a DFG into *S* and *H* is depicted in Figure 3. This is an iterative improvement 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. Within each iteration of ISEGEN (line 02 to line 20), $last\_best\_C$ retains the best cut found so far with the help of $best\_C$ that maintains the intermediate best cuts. Initially, the cut C points to a configuration where all nodes belong to software and this configuration is passed down to $best\_C$. The decision to toggle n with respect to $best\_C$ is based on a gain function, $M_{\text{toggle}}()$. The gain function is evaluated for each node (line 06) and the node with the best gain, *best_node* (obtained in line 08) is then toggled and marked (line 09). 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 eventually grow into a valid cut.

If both convexity and I/O constraints are satisfied (line 11), $best\_C$ is updated through removal of $best\_node$ from the cut or its addition to the cut depending on whether $best\_node$ has toggled from *H* to *S* or *S* to *H* respectively. The speedup estimate or merit function, $\mathrm{M}()$ determines whether $best\_C$ should override $last\_best\_C$ (line 17). This process is carried on till no more unmarked nodes are left. In general, we found experimentally that 5 passes are enough for successive improvement of the solution. Therefore,

SetInitialConditions()

00: $I_{\text{ISE}} \Leftarrow 0$
01: $O_{\text{ISE}} \Leftarrow 0$
02: **foreach** (node $n \in$ DFG)
03: $\quad I_{\text{toggle}}(n) \Leftarrow Inputs(n)$
04: $\quad O_{\text{toggle}}(n) \Leftarrow Outputs(n)$
05: **endfor**

Figure 4. *SetInitialConditions()* Procedure

the exit condition in the outermost loop is set to 5 times or lower when there is no improvement in the merit of the solution across successive iterations. We call each iteration of the loop enclosed between line 02 and 20, a *K-L pass* for iterative improvement. The best cut ($last\_best\_C$) is stored back in C that further acts as a starting point for the next bi-partitioning of the DFG.

4.2 Details of Functions inside ISEGEN

The three important functions inside ISEGEN are: (1) the Gain function, M_{toggle}, (2) the function calculating the impact of toggling a node, i.e., $CalcImpactOfToggle()$, and (3) the Merit function, $M()$. In this section, we go through the details of these functions and then report the complexity of ISEGEN.

4.2.1 Gain Function

A 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 five 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 independently connected components if they have higher speedup potential. Thus, the gain function for determining a node n to toggle with respect to a cut C is a linear weighted sum of the five 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.

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

  This is an estimation of speedup exhibited by C and therefore a positive contributor. This is set to $-\infty$ if the convexity constraint is violated for C.

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

$$io\_pnlty = \left(\left(I_{\text{ISE}}\left(C'\right) - N_{\text{in}}\right) + \left(O_{\text{ISE}}\left(C'\right) - N_{\text{out}}\right)\right),$$

  This is a negative contributor.

- **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 removed from the cut. Let $num\_nbrs\_in\_cut\,(n, C)$ be the number of neighbors of n in C.

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

  Thus, it acts as a positive contributor for a node toggling from *S* to *H* and a negative contributor for a node toggling from *H* to *S*.

- **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 $d\_to\_bars\_up\,(n)$ be the minimum distance of n from the barriers in the upward direction and let $d\_to\_bars\_down\,(n)$ be the minimum distance of n from the barriers in the downward direction.

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

  We employ a directional growth strategy where nodes closer to the barrier (that have higher potential for cut growth) are consistently favored for inclusion in hardware; this strategy implicitly favors reusability of the cut without losing the benefit of having large cut as a solution.

- **Independent Cuts**: It is quite possible that the best cut is actually a combination of 2 or 3 large connected subgraphs and not necessarily the largest connected subgraph. So, ISE exploration needs to expand not only in the vertical direction favoring large cuts but also in the horizontal direction. Let $CS(G)$ be the independently connected subgraphs in the DFG G excluding the connected subgraph containing n.

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

  where $CP\_lat\,(cs)$ is the sum of the hardware latencies along the critical path of the independently 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 $M_{\text{toggle}}\,(n)$ with respect to the current cut C as follows:

$$\alpha_1 \cdot merit - \alpha_2 \cdot io\_pnlty + \alpha_3 \cdot conv\_cons + \alpha_4 \cdot cgp + \alpha_5 \cdot idc$$

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 [9] does not consider the last two components of M_{toggle} in its fitness function.

CalcImpactOfToggle(best_node,best_C)

00: $I_{\rm ISE}(best\_C) \Leftarrow I_{\rm ISE}(best\_C) + I_{\rm toggle}(best\_node)$
01: $O_{\rm ISE}(best\_C) \Leftarrow O_{\rm ISE}(best\_C) + O_{\rm toggle}(best\_node)$
02: $I_{\rm toggle}(best\_node) \Leftarrow -I_{\rm toggle}(best\_node)$
03: $O_{\rm toggle}(best\_node) \Leftarrow -O_{\rm toggle}(best\_node)$
04: **foreach** (node $m \in$ Parents—Children—Siblings($best\_node$))
05: Apply Rules for updating $I_{\rm toggle}(m)$ and $O_{\rm toggle}(m)$
06: **endfor**
07: Maintain appropriate data structure for fast eval of $merit$, **M**()
08: Maintain appropriate data structures for fast eval of $conv\_cons$

Figure 5. *CalcImpactOfToggle()* Procedure

4.2.2 Impact of Toggling a Node

The runtime complexity of $M_{\rm toggle}$ is significantly reduced by trading the majority of computations into appropriately evaluating the impact of toggling a node (using *CalcImpactOfToggle()* of line 10 in Figure 3 as shown in Figure 5). The number of inputs and the number of outputs of ISE at any stage of the partitioning process are given by $I_{\rm ISE}$ and $O_{\rm ISE}$ respectively. In order to quantify the impact of toggling a node, we introduce **addendums** $I_{\rm toggle}$ and $O_{\rm toggle}$ associated with every node. When a node is toggled, its addendums $I_{\rm toggle}$ and $O_{\rm toggle}$ are added to $I_{\rm ISE}$ and $O_{\rm ISE}$ respectively to get the new values of $I_{\rm ISE}$ and $O_{\rm ISE}$. Initially, all nodes are in *S* and therefore $I_{\rm ISE}$ = $O_{\rm ISE}$ = 0 and $I_{\rm toggle}$ and $O_{\rm toggle}$ equal the number of inputs and number of outputs respectively of the corresponding node (as shown in Figure 4).

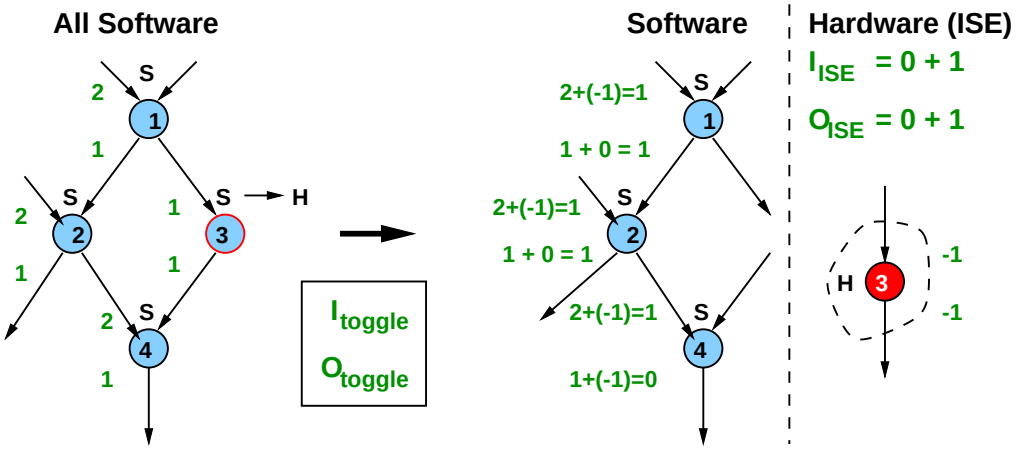


Figure 6. Instance of an Instruction-level Hardware-Software Partitioning

Depending on whether a node belongs to software or hardware, we call it an S-node or an H-node respectively and we denote a toggle of a node from *S* to *H* as $S \rightarrow H$. When a node is toggled, the $I_{\rm toggle}$ and $O_{\rm toggle}$ values of its neighbors (parents, children or siblings) may get affected as illustrated in Figure 6. When node 3 is toggled to H-node, its addendums are added to $I_{\rm ISE}$ and $O_{\rm ISE}$ respectively to reflect the number of inputs and outputs in the resulting ISE. After toggling, $I_{\rm toggle}$ and $O_{\rm toggle}$ of node 3 reverse in

sign showing that the changes to $I_{\rm ISE}$ and $O_{\rm ISE}$ will be undone if node 3 toggles back to S-node. It is easy to verify that the changed values of $I_{\rm toggle}$ and $O_{\rm toggle}$ for the neighbors of node 3 correctly account for the new values of $I_{\rm ISE}$ and $O_{\rm ISE}$ when any of these nodes is toggled. For example, if node 1 is toggled next, $I_{\rm ISE} = 1 + 1 = 2$ and $O_{\rm ISE} = 1 + 1 = 2$ with ISE containing nodes 1 and 3. Instead, if node 4 is toggled next, $I_{\rm ISE} = 1 + 1 = 2$ and $O_{\rm ISE} = 1 + 0 = 1$ with ISE containing nodes 3 and 4. On the other hand, if node 1 is toggled back to an S-node, $I_{\rm ISE} = 1 + (-1) = 0$ and $O_{\rm ISE} = 1 + (-1) = 0$ with ISE containing no nodes.

We developed a comprehensive set of rules to capture the effect of toggling a node that is pictorially presented in Figure 7 and Figure 8. The changes in $I_{\rm toggle}$ and $O_{\rm toggle}$ values are represented as $\Delta I_{\rm toggle}$ and $\Delta O_{\rm toggle}$ respectively such that the new values of $I_{\rm toggle}$ and $O_{\rm toggle}$ for the affected nodes are computed as $(I_{\rm toggle} + \Delta I_{\rm toggle})$ and $(O_{\rm toggle} + \Delta O_{\rm toggle})$ respectively. The rules are as follows:

Rule 1. After toggling, $I_{\rm toggle}$ and $O_{\rm toggle}$ values for a node reverse in sign.

Rule 2. If a node *n* is toggled from *S* to *H* or *H* to *S*, $I_{\rm toggle}$ and $O_{\rm toggle}$ 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 7(a-g)):

1. If *n* has one and only one child, then
   (a) If the child is an S-node, $\Delta I_{\rm toggle} = \Delta O_{\rm toggle} = -1$ for the child (Figure 7(a)).
   (b) If the child is an H-node, $\Delta I_{\rm toggle} = \Delta O_{\rm toggle} = +1$ for the child (Figure 7(b)).
2. If *n* has ≥ 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, $\Delta O_{\rm toggle} = -1$ and for the H-node, $\Delta I_{\rm toggle} = +1$ (Figure 7(c)).
   (b) If all the children are S-nodes, then for the S-nodes, $\Delta I_{\rm toggle} = -1$ (Figure 7(d)).
   (c) If all the children are H-nodes, then for the H-nodes, $\Delta O_{\rm toggle} = +1$ (Figure 7(e)).
   (d) If only one child is an S-node and > 1 nodes are H-nodes, then for the S-node, $\Delta O_{\rm toggle} = -1$ (Figure 7(f)).
   (e) If only one child is an H-node and > 1 nodes are S-nodes, then for the H-node, $\Delta I_{\rm toggle} = +1$ (Figure 7(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 7(h)):

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

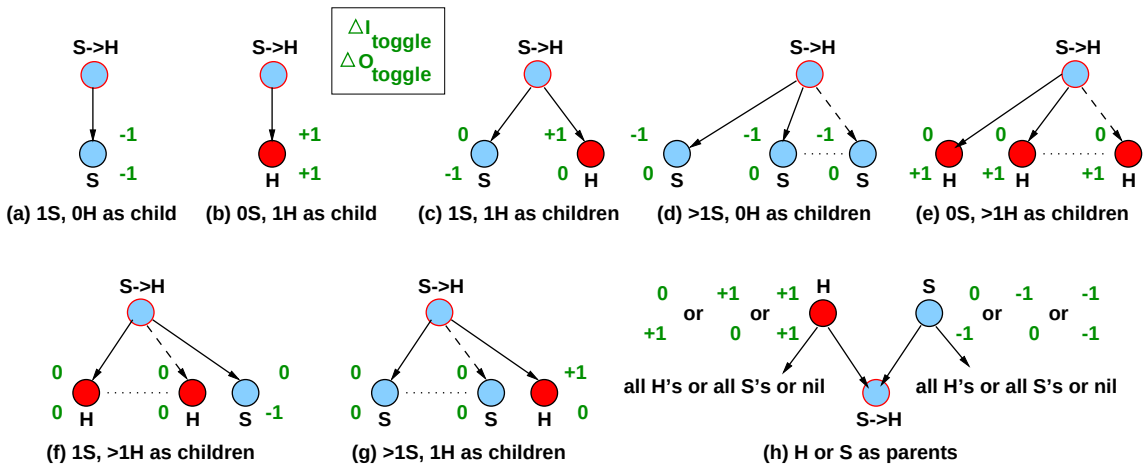


Figure 7. Basic Rules to project the effect of toggling a node from S to H to its parents and children.

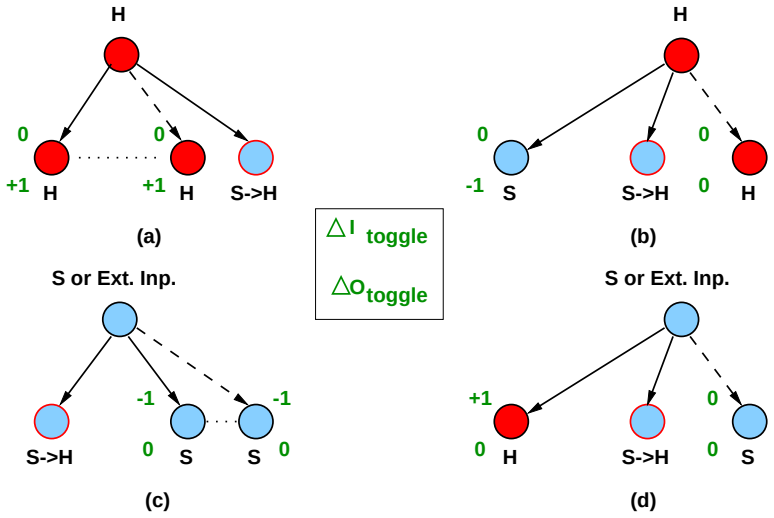


Figure 8. Basic Rules to project the effect of toggling a node from S to H to its siblings.

(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 8):

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 8(a)).

(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 8(b)).

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 8(c)).

   (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 8(d)).

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}.

The proofs of correctness for all the rules have been presented in Appendix A. The impact of toggling a node also involves maintenance of appropriate data structures for fast evaluation of $\mathrm{M}()$ and convexity violation.

4.2.3 Merit Function

We define the merit function as: $\mathrm{M}(C) = \lambda_{sw}(C) - \lambda_{hw}(C)$, where $\lambda_{\text{sw}}(C)$ is the software latency of C estimated by summing the latencies of the nodes in C; $\lambda_{\text{hw}}(C)$ is the hardware latency of C estimated from the critical path in 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)*.

4.2.4 Complexity of ISEGEN

The computational complexities of M_{toggle} and $CalcImpactOfToggle()$ are critical to the complexity of ISEGEN. Because of maintaining efficient data structures, precalculating node attributes and transferring significant portions of computations to $CalcImpactOfToggle()$, M_{toggle} has the worst case complexity of O(p), where p is the maximum number of neighbors that a node can have. For all the nodes, this amounts to O($p \cdot |V|$), i.e., O($|E|$). By choosing appropriate data structures, $CalcImpactOfToggle()$ can also be performed in O($|E|$) in the worst case.

Therefore, the worst-case running time of ISEGEN is O($|V| \cdot |E|$). For details, please refer to Appendix B.

4.3 A Detailed Running Example on ISEGEN

We illustrate running ISEGEN algorithm with a simple example shown in Figure 9. With an I/O constraints of $N_{\text{in}} = 4$ and $N_{\text{out}} = 2$, the solution is obtained in just 2 K-L passes. For simplicity, λ_{sw} and λ_{hw} for each node have been chosen as 2.0 and 1.0 respectively. Because of toggling a node, the addendums on the neighboring nodes (including parents, children and siblings) change according to the rules presented in the last section.

At the end of the first pass, a valid solution is obtained which is improved further in the next pass. For example, at the end of PASS 1, the software latency of the cut, $\lambda_{\text{sw}} = 6 \cdot 2.0 = 12.0$ because there are 6 nodes in the cut and the hardware latency of the cut, $\lambda_{\text{hw}} = 4 \cdot 1.0 = 4.0$ because there are

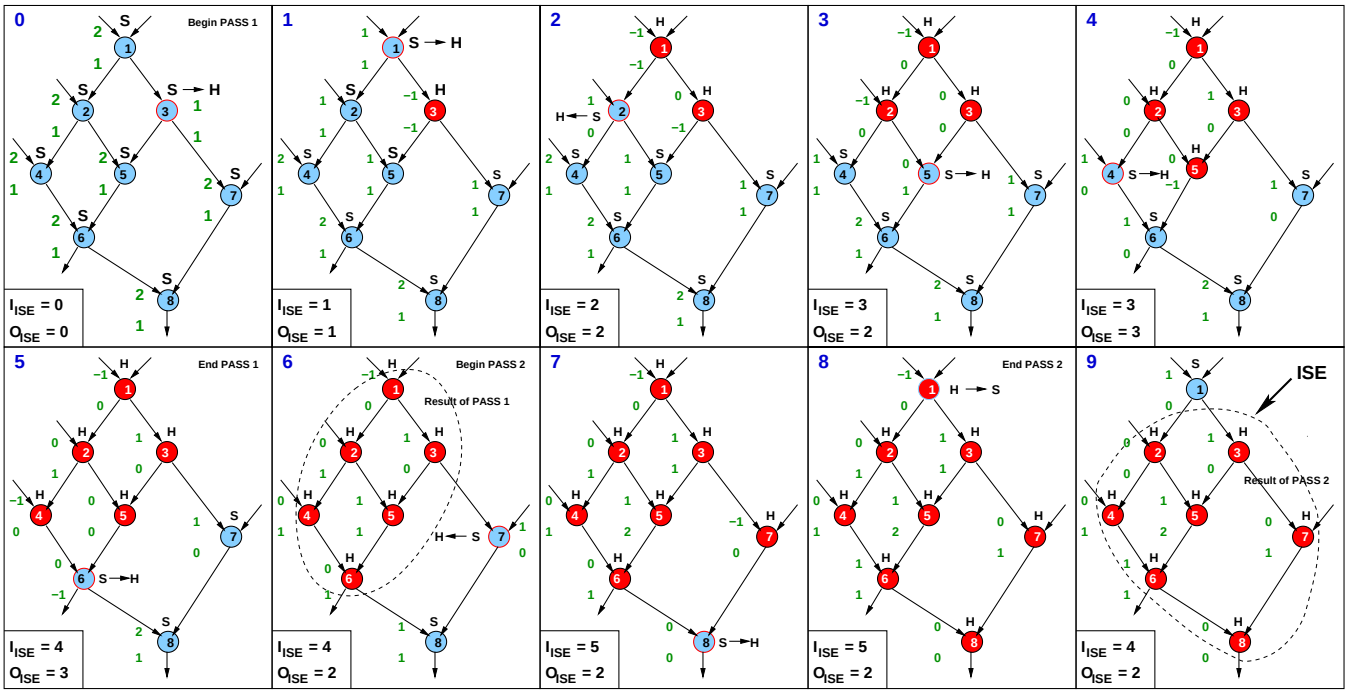


Figure 9. Running Example: The nodes are annotated with I_{toggle} and O_{toggle} values. Note that when a node is toggled (from *S* to *H* or *H* to *S*), the addendums of the node along with those of its neighbors change. The steps 1 through 5 are executed in PASS 1 and steps 6 through 8 in PASS 2 with the generated ISE shown in step 9. The solutions obtained at the end of PASS 1 and PASS 2 are shown in steps 6 and 9 respectively.

4 nodes in the critical path. Therefore, $M() = 12.0 - 4.0 = 8.0$. Similarly, at the end of PASS 2, $M() = 7 \cdot 2.0 - 4 \cdot 1.0 = 10.0$ which is greater than the previous gain. The PASS 3 does not show any further improvement in $M()$ and therefore the search is terminated. Note even in this small example that the passes have to incur violation of constraints (e.g. steps 4, 5, 7 and 8 violate I/O constraints) in their intermediate steps before converging to a valid solution.

5 Experimental Results

We integrated ISEGEN in the MachSUIF framework [18] and evaluated overall speedup for the entire application using all the generated cuts as follows:

$$\frac{\lambda_{\text{overall}}}{\lambda_{\text{overall}} - \sum_C N_C \cdot \mathrm{M}(C)}$$

The variable, λ_{overall} encapsulates the overall execution latency of the application i.e., when the application entirely runs in software, and N_C is the number of times C is executed based on profile information. Note that, in this work, we do not consider memory operations inside a cut.

To evaluate the efficacy of our ISEGEN approach, we chose benchmarks from diverse application domains in EEMBC (*autcor00*, *viterbi00*, *conven00*, *fft00* and *fbital00*) and MediaBench (*adpcm coder*

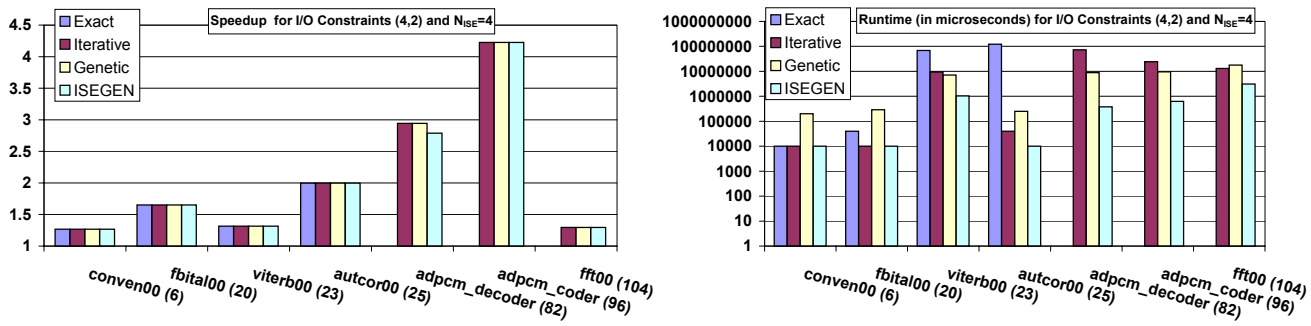


Figure 10. Comparison of Speedup and Runtime with number of AFUs = 4 and I/O constraints: (4,2)

and $adpcm\ decoder$) suites. In addition, we chose a cryptographic application viz. AES. Our baseline architecture is a simple RISC machine and we allow up to 4 AFUs (or ISEs) to be added.

5.1 Speedup and Runtime for Different Benchmarks

Keeping the I/O constraints fixed at $(4, 2)$, we study the overall speedup of applications obtained over execution on the core processor and the time taken to generate ISEs (or runtime) on Sun Ultra-5. We compare the quality of our results with the best known algorithms for ISE generation. The optimal algorithms for ISE generation [8] come in two flavors: Exact multiple-cut identification (or **Exact** in short) and Iterative exact single-cut identification (or **Iterative**), both of which employ exhaustive search with pruning. For applications having large basic blocks, we chose a genetic formulation [9] for comparing our results.

We associate with each benchmark the maximum number of nodes in its critical basic block (shown in parentheses) and arrange them in increasing order. It is evident from the first plot of Figure 10 that ISEGEN matches the solution quality of Exact, Iterative and Genetic algorithms. Note that because of effective pruning, Exact is able to handle up to 25 nodes and Iterative is able to handle up to 104 nodes in the selected benchmarks. As shown in the second plot of Figure 10, ISEGEN runs up to $29\times$ faster than the genetic approach with the generated ISEs having quality comparable with the optimal solution in terms of overall speedup. We observed that some of the ISEs identified by the optimal algorithms are independent subgraphs and therefore an ISE identification algorithm should not be restricted to identify only connected subgraphs.

Figure 11 collectively presents a comparison with the previous genetic approach [9] for application speedup and runtime of the algorithm obtained on four of the chosen benchmarks with varying I/O constraints. This set of plots shows that our ISEGEN closely matches the quality of the genetic solutions in terms of application speedup, but generates soulutions with a much quicker response time. Unlike [10] where substantial speedup was obtained only on cryptographic applications, we show considerable speedup on a diverse set of applications with varying microarchitectural constraints.

Recall that the ISEGEN heuristic starts walking the solution space from regions where cluster growth potential is higher. But, when an application has a large DFG, under tight I/O constraints, the solutions are expected to have small clusters and therefore may not lie in regions of expected growth. To circumvent this problem, we run ISEGEN multiple times (say, k iterations) in succession for a single cut identification and choose the cut having the highest speedup. For $adpcm\_coder$ and $fft00$ with I/O constraints of (2,1), (3,1) and (4,1), ISEGEN was run with $k = 14$ and $k = 6$ times respectively for the first cut identification and k was reduced on subsequent iterations. The runtime evaluation of ISEGEN (done for k iterations)

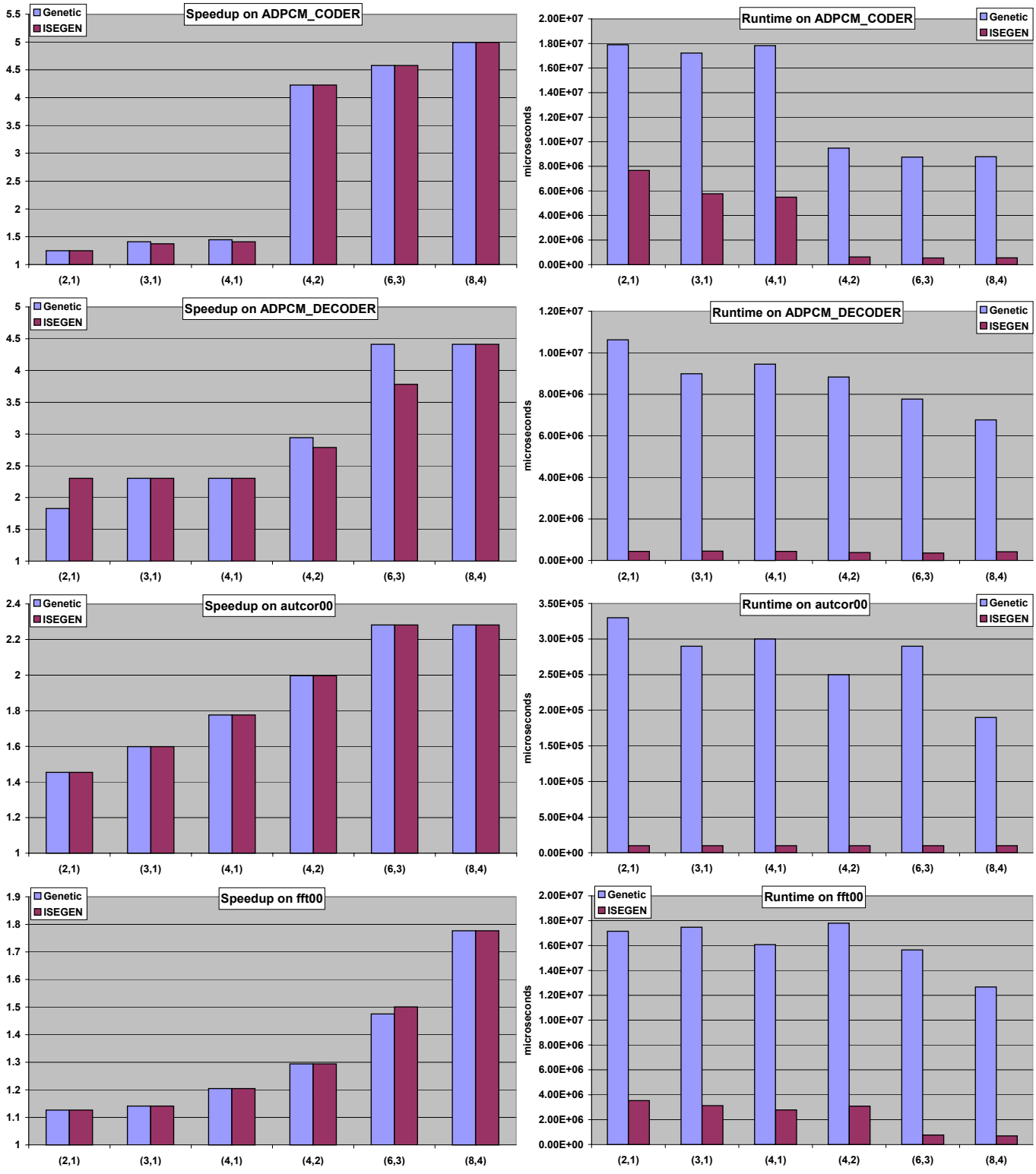


Figure 11. Comparison of Application Speedup and Algorithm Runtime with number of AFUs = 4

against the genetic algorithm clearly shows that our ISEGEN is still much faster. Note that all the other experiments were conducted with $k = 1$ and the evaluation of regions having a high growth potential was

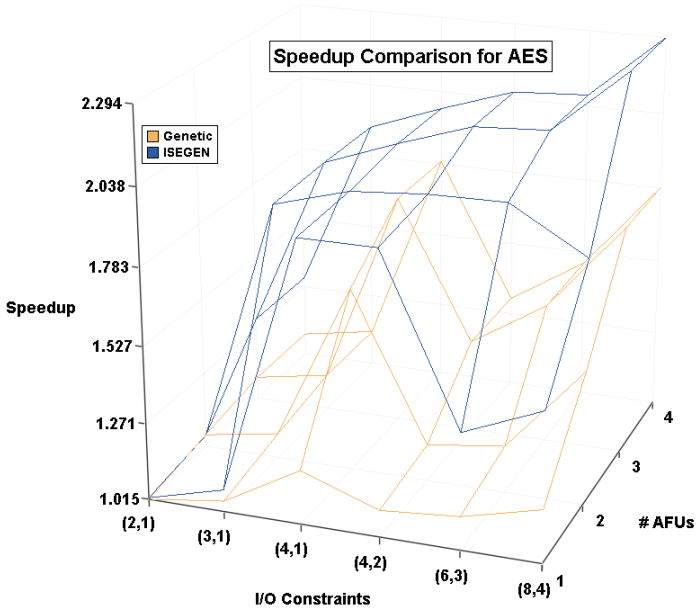


Figure 12. Speedup comparison for AES.

effective in obtaining a high speedup.

5.2 Experiments with AES

AES is a cryptographic benchmark with a large DFG; its critical basic block contains 696 nodes with a symmetric structure. Since the optimal algorithms (Exact and Iterative) could not run on such a large application, we chose the genetic solution (that also matches the optimal solution in smaller benchmarks) for comparing our results. Because of its non-exponential complexity, ISEGEN easily handles large DFGs. We deliberately chose AES to demonstrate the efficacy of our ISEGEN approach in matching expert design quality. We increased the maximum number of AFUs from 1 to 4, and studied the application speedup with variations in I/O constraints as shown in Figure 12.

On average, ISEGEN obtains 35% more speedup than the genetic solution by effectively exploiting the regularity in the data flow graph of AES. Figure 14 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 contrarily 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 12). The reason is clear from the plot of Figure 14. 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 Figure 12, the 12 instances generated for $(4, 1)$ cover 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

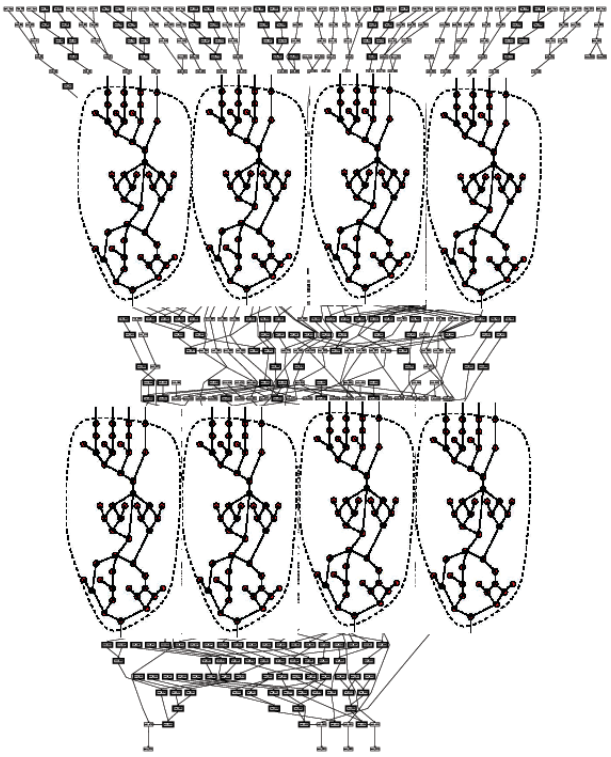


Figure 13. First cut (having 8 instances) generated for AES under I/O Constraints: (4,1). Each instance of the cut contains 49 nodes covering about 60% of the DFG.

with relaxing I/O constraints.

Figure 13 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 the reusability of ISEs by producing all the instances in the DFG (as also shown in Figure 14). Thus, the solutions generated by ISEGEN are indeed close to those generated by an expert designer.

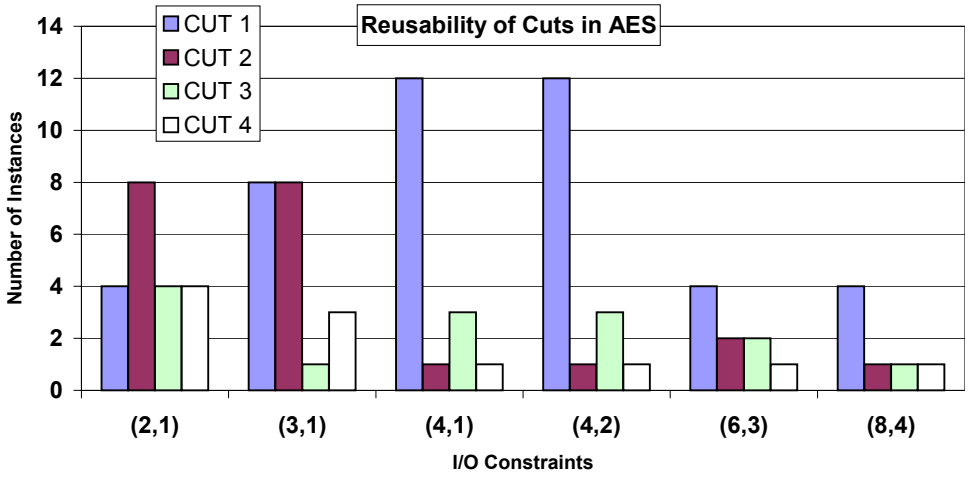


Figure 14. Study of Reusability of ISEs on AES with varying number of AFUs

6 Conclusions

The hardware-software partitioning problem when applied at the instruction-level granularity constitutes the problem of ISE generation. The contributions presented in this report are as follows. 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 ISE generation with a low computational complexity. Finally, we show that our ISEGEN approach produces high-quality ISEs — close to those sought after by an expert designer. Furthermore, ISEGEN runs up to $29\times$ faster than a previous genetic approach and generates solutions comparable with the optimal ISE generation approaches. Our future work will focus on the deployment of ISEs in a real system and evaluating the impact of ISEs on code size and energy reduction.

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 *Proc. of DAC*, 1982.

[3] L. Tai, D. Knapp, R. Miller, and D. Macmillen Scheduling using Behavioral Templates. In *Proc. of DAC*, 1995.

[4] T. J. Callahan, P. Chong, A. Dehon, and J. Wawrzynek. Fast Module Mapping and Placement for Datapaths in FPGAs. In *Proc. of FPGA*, 1998.

[5] D. S. Rao and F. J. Kurdahi. On Clustering for Maximal Regularity Extraction. *IEEE TCAD*, 1993.

[6] M. Kahrs. Matching a Parts Library in a Silicon Compiler. In *proc. of ICCAD*, 1986.

[7] K. Kuetzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *proc. of DAC*, 1987.

[8] K. Atasu, L. Pozzi and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *Proc. of DAC*, 2003.

[9] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt. Introduction of Local Memory Elements in Instruction Set Extensions. In *Proc. of DAC*, 2004.

[10] N. Clark, H. Zhong and S. Mahlke. Processor Acceleration through Automated Instruction Set Customization. In *Proc. of MICRO*, 2003.

[11] P. Yu and T. Mitra. Scalable Custom Instructions Identification for Instruction-Set Extensible Processors. In *Proc. of CASES*, 2004.

[12] F. Sun, S. Ravi, A. Raghunathan and N. K. Jha. Synthesis of Custom Processors based on Extensible Platforms. In *Proc. of ICCAD*, 2002.

[13] M. Arnold and H. Corporaal. Designing Domain-specific Processors. In *Proc. of CODES*, 2001.

[14] 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 TC*, 1999.

[15] J. Cong, Y. Fan, G. Han, and Z. Zhang Application-Specific Instruction Generation for Configurable Processor Architectures. In *Proc. of FPGA*, 2004.

[16] C. Alippi, W. Fornaciari, L. Pozzi and M. Sami. A DAG based Design Approach for Reconfigurable VLIW Processors. In *Proc. of DATE*, 1999.

[17] R. Razdan and M. D. Smith. A High-performance Microarchitecture with Hardware-programmable Functional Units. In *Proc. of MICRO*, 1994.

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

A Proof of the Rules for updating I_{toggle} and O_{toggle}

Rule 1: After toggling a node *n* from S to H or H to S, for a cut C, $I_{\text{ISE}}^{\text{new}}(C) = I_{\text{ISE}}(C) + I_{\text{toggle}}(n)$ and $O_{\text{ISE}}^{\text{new}}(C) = O_{\text{ISE}}(C) + O_{\text{toggle}}(n)$. This implies that $I_{\text{ISE}}(C) = I_{\text{ISE}}^{\text{new}}(C) + (-I_{\text{toggle}}(n))$ and $O_{\text{ISE}}(C) = O_{\text{ISE}}^{\text{new}}(C) + (-O_{\text{toggle}}(n))$.

Thus, toggling back the node *n* to restore the values of I_{ISE} and O_{ISE} necessitates negation of I_{toggle} and O_{toggle}. This proves *Rule 1*.

Rule 2: The inputs and outputs of a node *n* can be shared by the inputs or outputs of only parents, children or siblings of *n* and no other nodes. Therefore, *Rule 2* holds true.

For the proofs of Rules 3 through 5, please refer to Figure 15. The possible relationships between the node that is toggled i.e., *n* and the node *m* whose addendums are affected because of toggling *n* are depicted in Figure 15(a-c). In order to prove Rules 3 through 5, we need to infer the effect of toggling *m* before and after toggling *n* from *S* to *H*, on the number of input and output contributions to the ISE. Let the input and output contributions **exclusively** generated owing to the relationship between *m* and *n* be I_{contrib} and O_{contrib} respectively. A contribution to $I_{\text{toggle}}(m)$ due to this relationship can be obtained by subtracting the value of I_{contrib} before toggling *m* from the value of I_{contrib} after toggling *m*. Similarly, a contribution to $O_{\text{toggle}}(m)$ can be evaluated from the corresponding values of O_{contrib}. Let the contributions to $I_{\text{toggle}}(m)$ and $O_{\text{toggle}}(m)$ from the other input operands and the other output operands of *m* be *x* and *y* respectively. We now prove the Rules 3 through 5 by deducing the input and output contribution to the ISE owing to the relationship between *m* and *n* and then projecting its effect on the changes in the addendums of *m* with the toggling of *n*.

Rule 3: *n* has one or more children as shown in Figure 15(a).

1. *m* is the one and the only child of *n* i.e., $Y' = \phi$.

   (a) *m* is in software:
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x + 1$ and $O_{\text{toggle}}(m) = y$.
   After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y - 1$. Hence, $\Delta I_{\text{toggle}}(m) = -1$ and $\Delta O_{\text{toggle}}(m) = -1$. This proves Rule 3.1.a.

   (b) *m* is in hardware: The proof of Rule 3.1.b follows symmetrically from Rule 3.1.a.

2. *n* has ≥ 2 children i.e., $Y' \neq \phi$.

   (a) There are exactly two children, an S-node and an H-node: Let *m* refer to the S-node.
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.
   After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and

$O_{\text{toggle}}(m) = y - 1$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = -1$. Let *m* now refer to the H-node.

Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x - 1$ and $O_{\text{toggle}}(m) = y$.

After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = +1$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 3.2.a.

(b) All children are S-nodes: Let *m* refer to the any of the children.

Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x + 1$ and $O_{\text{toggle}}(m) = y$.

After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = -1$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 3.2.b.

(c) All children are H-nodes: Let *m* refer to the any of the children.

Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.

After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y + 1$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = +1$. This proves Rule 3.2.c.

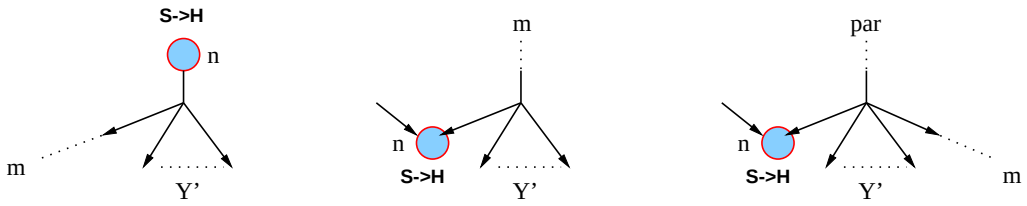
(d) Only one child is an S-node, other > 1 nodes are H-nodes: Let *m* refer to the S-node.

Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.

After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y - 1$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = -1$. Let *m* now refer to any of the children as H-nodes.

Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.

After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 3.2.d.



(a) m is a child of n (b) m is a parent of n (c) m is a sibling of n

Figure 15. Possible Relationships between nodes *m* and *n*.

(e) Only one child is an H-node, other > 1 nodes are S-nodes: Let *m* refer to the H-node.
Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x - 1$ and $O_{\text{toggle}}(m) = y$.
After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = +1$ and $\Delta O_{\text{toggle}}(m) = 0$. Let *m* now refer to any of the children as S-nodes.
Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.
After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 3.2.e.

Rule 4: *m* is the parent of *n* as in Figure 15(b).

1. *m* is an S-node.

   (a) *m* has no other children:
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y + 1$.
   After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x - 1$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = -1$ and $\Delta O_{\text{toggle}}(m) = -1$. This proves Rule 4.1.a.

   (b) *m* has some of the other children as S-nodes:
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y + 1$.

After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the input of ISE $\implies$ $I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore$ $I_{\text{toggle}}(m) = x - 1$ and $O_{\text{toggle}}(m) = y + 1$. Hence, $\Delta I_{\text{toggle}}(m) = -1$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 4.1.b.

(c) *m* has all the other children as H-nodes:
Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y + 1$.
After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = -1$. This proves Rule 4.1.c.

2. *m* is an H-node. By symmetry, Rules 4.2.a, 4.2.b and 4.2.c follow from the proofs of Rules 4.1.a, 4.1.b and 4.1.c respectively.

Rule 5: *n* has one or more siblings as shown in Figure 15(c).

1. The parent of *n* is an H-node i.e., *par* in *H*.

   (a) All the siblings are H-nodes: Let *m* refer to any of the siblings of *n*.
   Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.
   After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y + 1$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = +1$. This proves Rule 5.1.a.

   (b) Only one sibling is in *S*, while the rest, if they exist, are in *H*: Let *m* refer to the sibling in *S*.
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$.
   After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the output of ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 1$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore$ $I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y - 1$. Hence, $\Delta I_{\text{toggle}}(m) = 0$ and $\Delta O_{\text{toggle}}(m) = -1$. This proves Rule 5.1.b.

2. The parent of *n* is an S-node or an external input i.e., *par* in *S* or an external input.

   (a) All the siblings are S-nodes. Let *m* refer to any of the siblings of *n*.
   Before toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies$ $I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$

Contribution only to the input of ISE $\implies I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x + 1$ and $O_{\text{toggle}}(m) = y$.
After toggling *n* – Before toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ Contribution only to the input of ISE $\implies I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ Contribution only to the input of ISE $\implies I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = -1$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 5.1.c.

(b) Only one sibling is in *H*, while the rest, if they exist, are in *S*.. Let *m* refer to the sibling in *H*.
Before toggling *n* – Before toggling *m* $\implies$ *n* in *S* but *m* in *H* $\implies$ Contribution only to the input of ISE $\implies I_{\text{contrib}} = 1$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ Both *n* and *m* in *S* $\implies$ No contribution to ISE $\implies I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x - 1$ and $O_{\text{toggle}}(m) = y$.
After toggling *n* – Before toggling *m* $\implies$ Both *n* and *m* in *H* $\implies$ No contribution to ISE $\implies I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. After toggling *m* $\implies$ *n* in *H* but *m* in *S* $\implies$ No contribution to ISE $\implies I_{\text{contrib}} = 0$ and $O_{\text{contrib}} = 0$. $\therefore I_{\text{toggle}}(m) = x$ and $O_{\text{toggle}}(m) = y$. Hence, $\Delta I_{\text{toggle}}(m) = +1$ and $\Delta O_{\text{toggle}}(m) = 0$. This proves Rule 5.1.d.

Rule 6: When a node is toggled from *S* to *H*, changes in addendums of the node itself and its neighboring nodes take place based on the above rules. When the node is toggled back from *H* to *S*, the changes must be undone to restore the state of all the nodes. Therefore, all the above rules can be applied for toggling a node from *H* to *S* with the sign reversed for the values of ΔI_{toggle} and ΔO_{toggle}.

B Analysis of ISEGEN Complexity

With the help of the above six rules, the addendums, I_{toggle} and O_{toggle} can be easily updated in O(p), where p is the maximum number of neighbors that a node can have. The critical region of our ISEGEN algorithm lies in the gain function, $M_{\text{toggle}}()$ and $CalcImpactOfToggle()$ that maintains the data structures affected by toggling a node. The complexity of $M_{\text{toggle}}()$ is composed of the complexities of its 5 components: $merit$, io_pnlty, $conv\_cons$, cgp and idc.

Both io_pnlty and $conv\_cons$ can be computed in constant time. In order to ensure constant time evaluation of cgp, all the nodes are annotated with $d\_to\_bars\_up$ and $d\_to\_bars\_down$ before ISEGEN starts. $d\_to\_bars\_up$ can be statically computed by considering external inputs and other barrier nodes as a single node (with all edges from the individual nodes mapped to edges emanating from this node) and calculating the minimum distances to all the other nodes from this node. Similarly, $d\_to\_bars\_down$ can also be statically computed by coalescing external outputs and other barrier nodes into a single node. Again, constant time evaluation of idc is possible by statically precalculating the hardware latencies of the independently connected components of a DFG. The most critical component is $merit$ that involves computing the merit function, $M()$ and checking convexity violation, both of which can be done in O(p) with the help of the function, $CalcImpactOfToggle()$. Together with the innermost loop, the complexity of $M_{\text{toggle}}()$ is O($p \cdot |V|$) i.e., O($|E|$). The worst-case complexity of $CalcImpactOfToggle()$ is also O($|E|$) and therefore the worst-case running time of ISEGEN is O($|V| \cdot |E|$).