ISEGEN: An Iterative Improvement-Based ISE Generation Technique for Fast Customization of Processors

Partha Biswas, *Member, IEEE*, Sudarshan Banerjee, *Member, IEEE*, Nikil D. Dutt, *Senior Member, IEEE*, Laura Pozzi, *Member, IEEE*, and Paolo Ienne, *Member, IEEE*

***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 paper, we present ISEGEN, an approach that identifies high-quality ISEs by iterative improvement following the basic principles of the well-known Kernighan–Lin 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× 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 by effectively exploiting its regular structure.**

***Index Terms*—Application-specific processors, hardware-software partitioning, instruction set extensions.**

I. 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). The AFU accelerates critical operations of application algorithms by executing application-specific instruction set extensions (ISEs).

Manuscript received July 1, 2005; revised December 23, 2005. This work was supported in part by the National Science Foundation under Grants CCR-0203813 and CCR-0205712, and by the Semiconductor Research Corporation under Contract 2003-HJ1111.

P. Biswas is with The MathWorks Inc., Natick, MA 01760 USA (e-mail: Partha.biswas@gmail.com).

S. Banerjee and N. D. Dutt are with the Center for Embedded Systems, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697 USA.

L. Pozzi is with the Department of Informatics, University of Lugano, Lugano CH-6900, Switzerland.

P. Ienne is with the Processor Architecture Laboratory, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, Lausanne CH-1015, Switzerland.

Digital Object Identifier 10.1109/TVLSI.2006.878345

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]. This heuristic had 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 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 sought after 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× 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.

II. 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 $M(C)$ 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 subproblems.

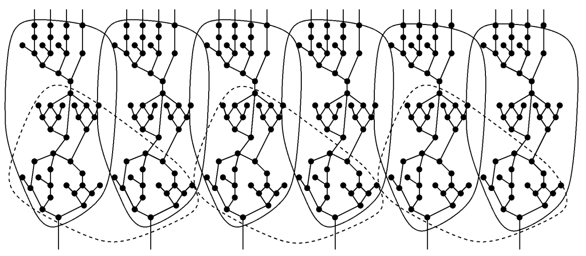


Fig. 1. Example showing the advantage of large-scale reuse—finding three instances of the largest ISE (shown with a dotted boundary) is not as effective as finding a large ISE with six instances (shown with a solid boundary).

1) Problem 1: Given the data flow graph (DFG) $G = (V, E)$ in a basic block, find a cut $C \subseteq G$ that maximizes $M(C)$ 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.

2) 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.

III. 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]–[5] and template matching [6], [7] that are used in a variety of computer-aided design (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 [18], [19] considers only single-output subgraphs in ISE generation. Even though a few recently proposed approaches [12], [13] 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]–[17], the resulting subgraphs are generally small. In practice, if one wants to mimic the excellent results targeted by expert designers, clusters of two or three 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. There is a need for algorithms that can identify large *and* reusable clusters, efficiently covering the application DFG. Fig. 1 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.

```
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: $C \Leftarrow last\_best\_C$
```

Fig. 2. ISEGEN algorithm.

An exact solution [8] that uses an exhaustive search with pruning is not practical for applications having large basic blocks. A genetic formulation [10] 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 aim to match the solution quality of expert designers.

IV. 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 K-L 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.

A. ISEGEN: A Modified K-L Algorithm

The ISEGEN algorithm that essentially performs a bi-partitioning of a DFG into S and H is depicted in Fig. 2. 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, $M()$ 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 five passes are enough for successive improvement of the solution. Therefore, the exit condition in the outermost loop is set to five 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.

B. 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.

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, which are: 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 following 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.

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

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

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

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 as follows:

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

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, as shown in the equation at the bottom of the page. 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 a large cut as a solution.

•**Independent Cuts**: 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. Thus, 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

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

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.

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

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 relations between the weights α_1, α_2, α_3, α_4, and α_5 have been determined experimentally to be as follows: $\alpha_3 = \alpha_4$, $\alpha_1 = 4\cdot\alpha_3$, $\alpha_2 = 2.5\cdot\alpha_1$, and $\alpha_5 = 25\cdot\alpha_1$. Thus, by using large factors, the speedup component is favored, the I/O violations are heavily penalized and ISE exploration is allowed to expand in the horizontal direction after the vertical direction has been already explored. We arrived at the above relations by studying the range of values each component can have and identifying the points in the iterative improvement steps where the weights must create a difference in the gain in order to induce a change in the cut growth pattern. It is to be noted here that the genetic algorithm [10] does not consider the last two components of M_{toggle} in its fitness function.

2) Impact of Toggling a Node: The runtime complexity of M_{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 Fig. 2). The number of inputs and the number of outputs of ISE at any stage of the partitioning process are given by I_{ISE} and O_{ISE}, respectively. In order to quantify the impact 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 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. Note that the number of edges incident on a node are its inputs, while the number of outputs is always 1 with the outgoing edges showing the data dependencies of the node.

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 $S \to H$. When a node is toggled, the I_{toggle} and O_{toggle} values of its neighbors (e.g., parents, children, or siblings) may get affected as illustrated in Fig. 3. When node 3 is toggled to H-node, 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. Instead, if node 4 is toggled next, $I_{\text{ISE}} = 1+1 = 2$ and $O_{\text{ISE}} = 1+0 = 1$ with ISE containing nodes 3 and 4. On the other hand, if node 3 is toggled back to an S-node, $I_{\text{ISE}} = 1 + (-1) = 0$ and $O_{\text{ISE}} = 1 + (-1) = 0$ with ISE containing no nodes.

We develop a comprehensive set of rules to capture the effect of toggling a node that is pictorially presented in Figs. 4 and 5. The changes in I_{toggle} and O_{toggle} values are represented as ΔI_{toggle} and ΔO_{toggle}, respectively, such that the new values

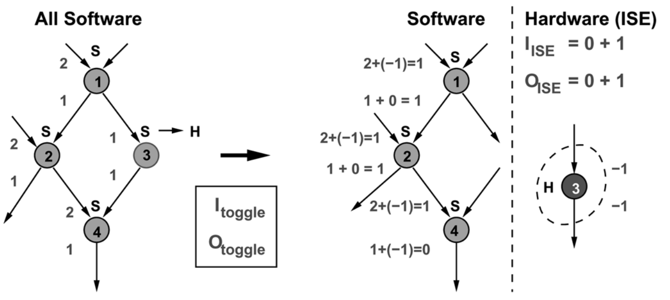


Fig. 3. Instance of an instruction-level hardware-software partitioning.

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.

We present the proofs of correctness for all of the rules in [9, Appendix]. The impact of toggling a node also involves maintenance of appropriate data structures for fast evaluation of $M()$ and convexity violation.

3) Merit Function: We define the merit function as: $M(C) = \lambda_{\text{sw}}(C) - \lambda_{\text{hw}}(C)$, where $\lambda_{\text{sw}}(C)$ is the software latency of C estimated by summing the latencies of the nodes in C and $\lambda_{\text{hw}}(C)$ 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-μm CMOS technology and then normalized to the delay of a 32-b multiply-accumulate (MAC).

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 [9].

C. Detailed Running Example on ISEGEN

We illustrate running ISEGEN algorithm (in Fig. 2) with a simple example shown in Fig. 6. With I/O constraints of $N_{\text{in}} = 4$ and $N_{\text{out}} = 2$, the solution is obtained in just two K-L passes. For simplicity, λ_{sw} and λ_{hw} for each node have been chosen as 2.0 and 1.0, respectively. At every step (in Fig. 6), $M_{\text{toggle}}()$ is calculated for each node (lines 05–07 in Fig. 2), and the best node is selected to toggle (line 08). In this example, nodes 3, 1, 2, 5, 4, 6, 7, and then 8 are selected in sequence as the best candidates for toggling in steps 0–7. 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 (line 10). After all the nodes are toggled and marked at the end of step 7, the last best cut *last_best_C* (which also satisfies the microarchitectural constraints) identified (line 17) is the cut in step 6 and is shown as "Result of PASS 1."

The valid solution thus found is iteratively refined in subsequent passes till there is no improvement in gain. 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 six nodes in the cut

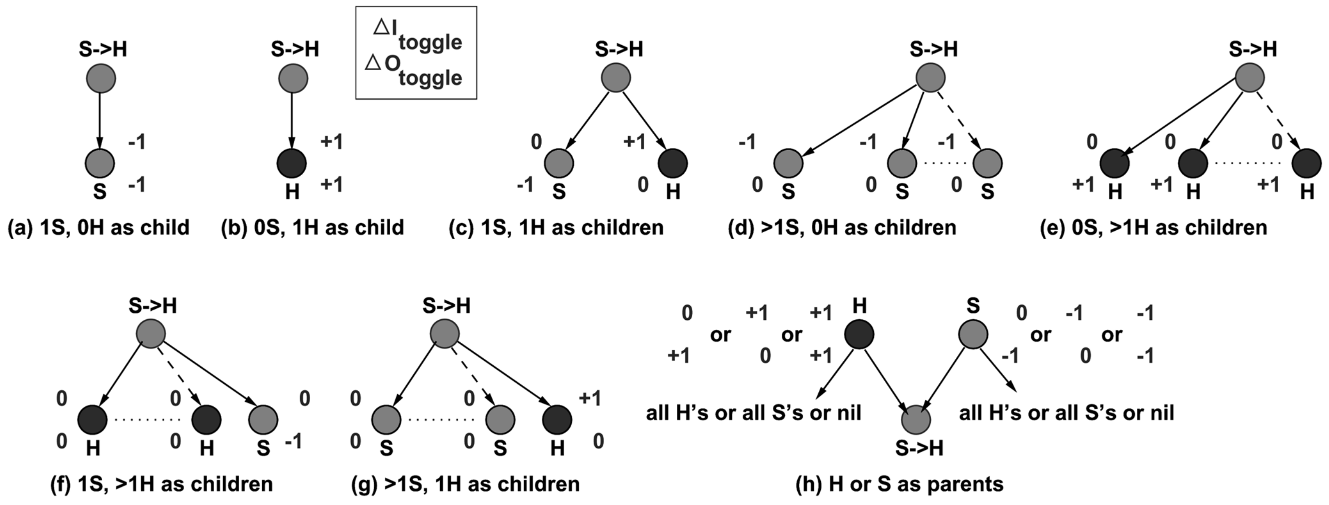


Fig. 4. Basic rules to project the effect of toggling a node from S to H to its parents and children.

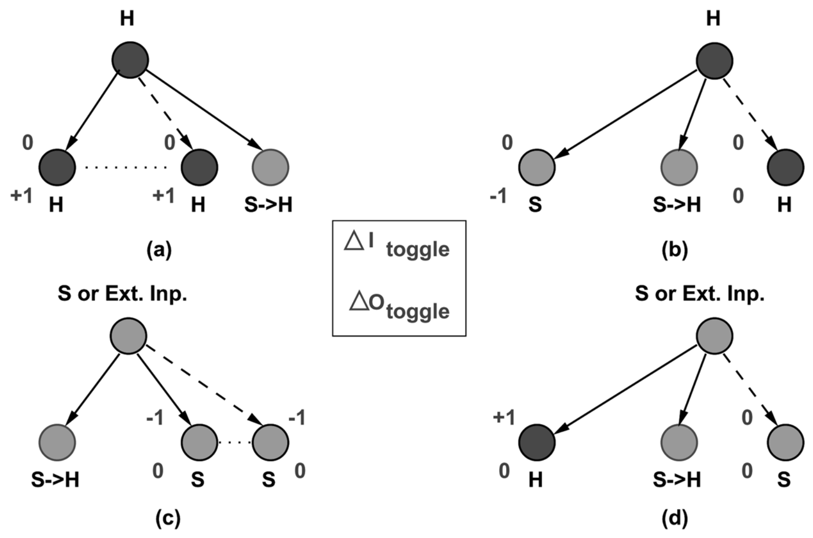


Fig. 5. Basic rules to project the effect of toggling a node from S to H to its siblings.

and the hardware latency of the cut, $\lambda_{\text{hw}} = 4 \cdot 1.0 = 4.0$ because there are four nodes in the critical path. Therefore, $M() = 12.0 - 4.0 = 8.0$ for the "Result of PASS 1." At the end of PASS 2, $M() = 7 \cdot 2.0 - 4 \cdot 1.0 = 10.0$, which is greater than the previous gain. 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.

V. Experimental Results

We integrated ISEGEN in the MachSUIF framework [20] and evaluated overall speedup for the entire application using all the generated cuts as follows:

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

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 study, 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* 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 four AFUs (or ISEs) to be added. Before running ISEGEN, we perform predication of small basic blocks in order to enlarge the scope of ISEs. As a result, the predicated part if chosen in ISE will map to a MUX realization in the corresponding AFU.

A. Speedup and Runtime for Different Benchmarks

Keeping the I/O constraints fixed at (4,2), we study the overall speedup of the chosen 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

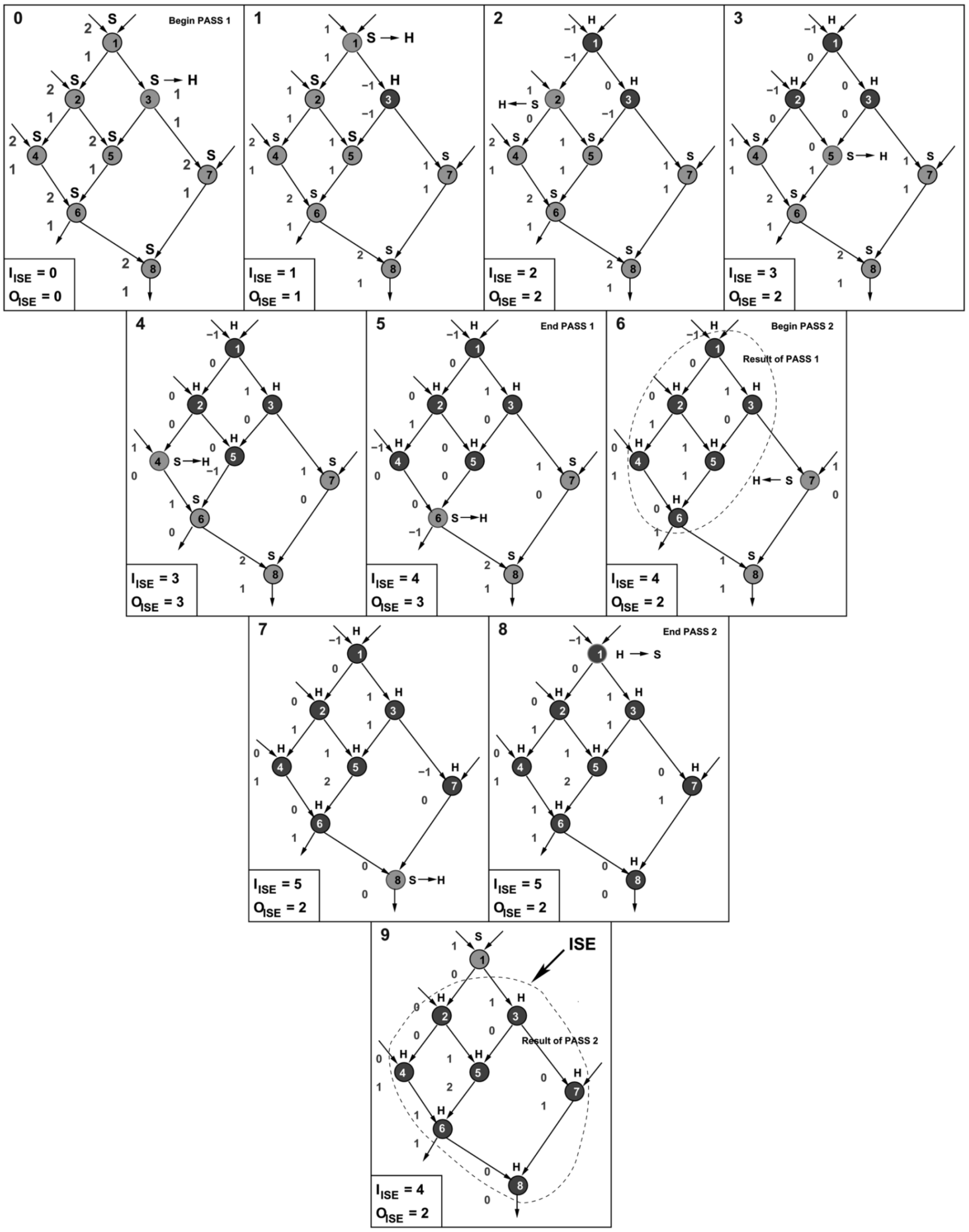
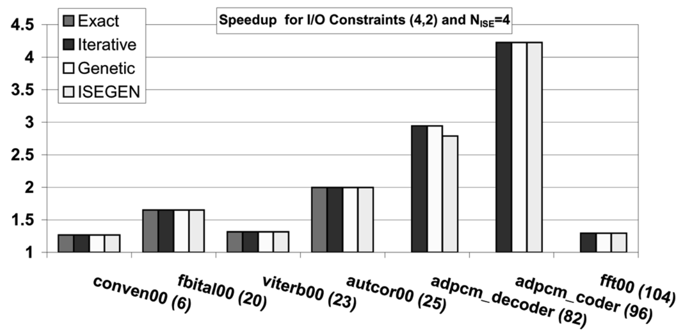


Fig. 6. Running example: the nodes are annotated with I_{toggle} and O_{toggle} values. Note that, when a node is toggled (from S to H or H to S), the addendums of the node along with those of its neighbors change. Steps 1–5 are executed in PASS 1 and steps 6–8 in PASS 2 with the generated ISE shown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

single-cut identification (or **Iterative**), both of which employ exhaustive search with pruning. For applications having large basic blocks, we chose a genetic formulation [10] 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 Fig. 7 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 Fig. 7, ISEGEN runs up to 29 × 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,



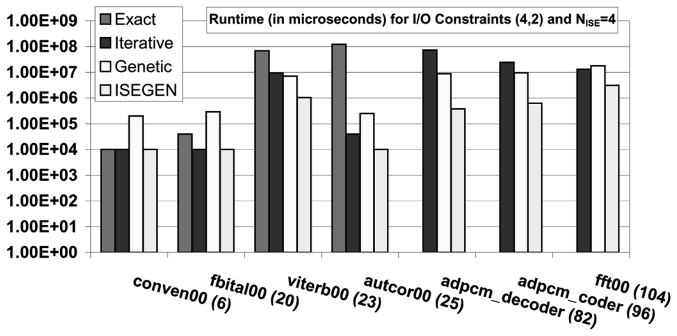


Fig. 7. Comparison of speedup and runtime with number of AFUs = 4 and I/O constraints: (4,2).

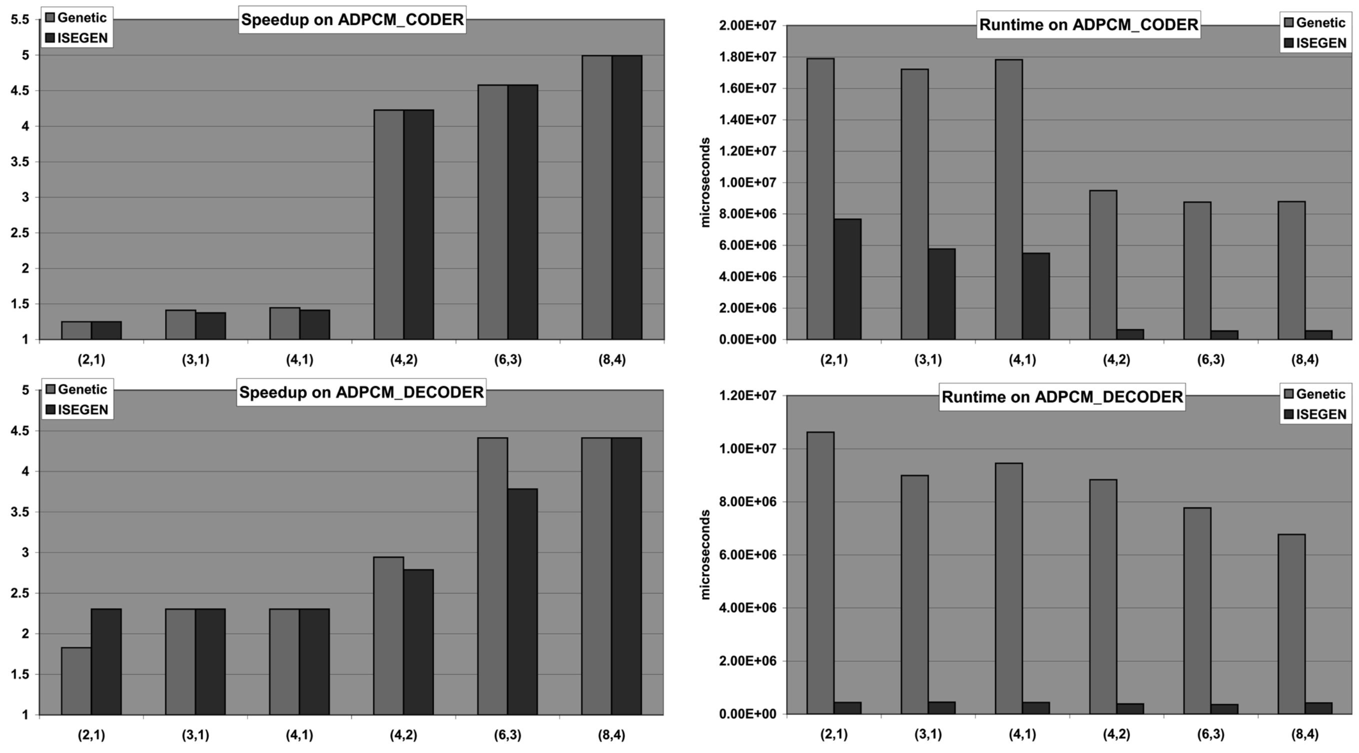


Fig. 8. Comparison of application speedup and algorithm runtime with number of AFUs = 4.

an ISE identification algorithm should not be restricted to identify only connected subgraphs.

Fig. 8 collectively presents a comparison with the previous genetic approach [10] for application speedup and runtime of the algorithm obtained on two 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 solutions with a much quicker response time.

B. 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 nonexponential 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 Fig. 9.

On average, ISEGEN obtains 35% more speedup than the genetic solution by effectively exploiting the regularity in the data flow graph of AES. Fig. 10 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 Fig. 9). The reason is that there are 12 instances of the first cut for the I/O constraint of (4,1) while there are only four instances for the I/O constraint of (6,3) (refer to Fig. 10). As is evident from Fig. 9, the 12 instances generated for (4,1) cover the DFG better

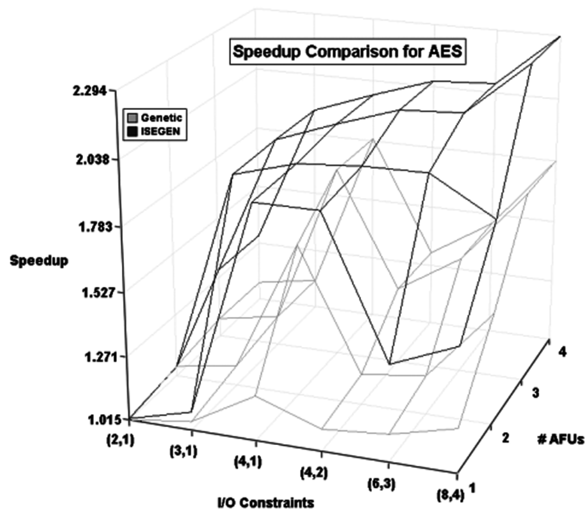


Fig. 9. Speedup comparison for AES.

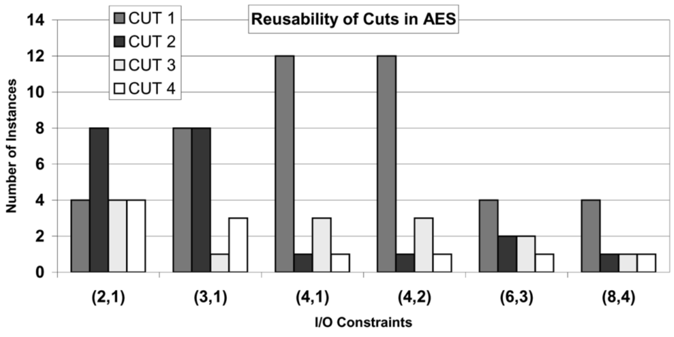


Fig. 10. Study of reusability of ISEs on AES with varying number of AFUs.

than the four instances generated for (6,3). However, with increase in the allowed number of AFUs, the speedup begins to scale with relaxing I/O constraints.

For an I/O constraint of (4,1), there are eight instances of the same cut covering 400 out of the 696 nodes (i.e., about 60% of the DFG), and all of 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 Fig. 10). This hints that the solutions generated by ISEGEN are indeed close to what an expert designer would manually find. In general, we envision long-latency ISEs to be run as multicycle operations causing processor stalls but not affecting the processor cycle time.

VI. Conclusion

The hardware–software partitioning problem when applied at the instruction-level granularity constitutes the problem of ISE generation. The contributions presented in this paper 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 K-L heuristic to perform ISE generation with a low computational complexity. Finally, the experimental results indicate that our ISEGEN approach produces high-quality ISEs—close to those sought after by an expert designer. Furthermore, ISEGEN runs up to 29 × faster than a previous genetic approach and generates solutions comparable with the optimal ISE generation approaches.

References

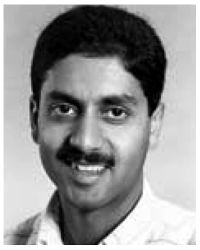
[1] F. Vahid and T. D. Le, "Extending the Kernighan/Lin heuristic for hardware and software functional partitioning," *Kluwer J. Design Autom. Embedded Syst.*, vol. 2, no. 2, pp. 237–261, 1997.
[2] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. DAC*, 1982, pp. 175–181.
[3] L. Tai, D. Knapp, R. Miller, and D. Macmillen, "Scheduling using behavioral templates," in *Proc. DAC*, 1995, pp. 101–106.
[4] T. J. Callahan, P. Chong, A. Dehon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in FPGAs," in *Proc. FPGA*, 1998, pp. 123–132.
[5] D. S. Rao and F. J. Kurdahi, "On clustering for maximal regularity extraction," *IEEE Trans. Comput-Aided Design Integr. Syst.*, vol. 12, no. 8, pp. 1198–1208, 1993.
[6] M. Kahrs, "Matching a parts library in a silicon compiler," in *Proc. ICCAD*, 1986, pp. 169–172.
[7] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," in *Proc. DAC*, 1987, pp. 341–347.
[8] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. DAC*, 2003, pp. 256–261.
[9] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne, ISEGEN: Adapting Kernighan-Lin min-cut heuristic for generation of instruction set extensions Center for Embedded Computer Systems, Univ. of California, Irvine, Tech. Rep. ICS-TR-04-21, 2004.
[10] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proc. DAC*, 2004, pp. 729–734.
[11] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, and L. Pozzi, "Performance and energy benefits of instruction set extensions in an FPGA soft core," in *Proc. VLSI-DESIGN*, 2006, pp. 651–656.
[12] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proc. MICRO*, 2003, pp. 129–140.
[13] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. CASES*, 2004, pp. 69–78.
[14] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proc. ICCAD*, 2002, pp. 641–648.
[15] M. Arnold and H. Corporaal, "Designing domain-specific processors," in *Proc. CODES*, 2001, pp. 61–66.
[16] 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 Trans. Comput.*, vol. 48, no. 6, pp. 603–614, Jun. 1999.
[17] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. FPGA*, 2004, pp. 183–189.
[18] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG based design approach for reconfigurable VLIW processors," in *Proc. DATE*, 1999, pp. 778–779.
[19] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. MICRO*, 1994, pp. 172–180.
[20] Machine SUIF [Online]. Available: http://www.eecs.harvard.edu/hube/software/software.html

Partha Biswas (M'01) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1998, and the M.S. and Ph.D. degrees in computer science from the University of California, Irvine, in 2002 and 2006, respectively.

He has served in the positions of Software Engineer and Member of Technical Staff with Cadence Design Systems, Noida, India, from 1998 to 2000. He is currently with The MathWorks Inc., Natick, MA. His primary interests include architectures and compilers for embedded systems and application-specific processor design.

Sudarshan Banerjee (M'99) received the B.Tech. degree from the Indian Institute of Technology, Kharagpur, India, and the M.Tech. degree from the Indian Institute of Technology, Kanpur, India, both in computer science and engineering, in 1992 and 1995, respectively, and is currently working toward the Ph.D. degree at the University of California, Irvine.

Prior to beginning his doctoral work, he had a long career in R&D of cutting-edge verification tools (as an employee in Synopsys and Cadence). His current areas of interest are partitioning and scheduling heuristics, dynamically reconfigurable FPGAs, and, approximation algorithms for strip-packing.



Nikil D. Dutt (S'82–M'84–SM'96) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, in 1989.

He is currently a Professor of Computer Science and Electrical Engineering and Computer Science with the University of California, Irvine (UCI), and he is affiliated with the following centers at UCI: CECS, CPCC, and CAL-IT2. His research interests are in embedded systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems. He currently serves as Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems* and as Associate Editor of the *ACM Transactions on Embedded Computer Systems*.

Dr. Dutt received Best Paper Awards at CHDL89, CHDL91, VLSIDesign2003, CODES+ISSS 2003, and ASPDAC-2006. He was an ACM SIGDA Distinguished Lecturer during 2001–2002 and an IEEE Computer Society Distinguished Visitor for 2003–2005. He has served on the steering, organizing, and program committees of several premier CAD and Embedded System Design conferences and workshops, including ASPDAC, CASES, CODES+ISSS, DATE, ICCAD, ISLPED, and LCTES. He serves or has served on the advisory boards of ACM SIGBED and ACM SIGDA and is Vice-Chair of IFIP WG 10.5.

Laura Pozzi (M'01) received the M.S. and Ph.D. degrees in computer engineering from the Politecnico di Milano, Milan, Italy, in 1996 and 2000, respectively.

She is an Assistant Professor with the faculty of Informatics, University of Lugano (USI), Lugano, Italy. From 2001 to 2005, she was a Postdoctoral Researcher with the School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland. Previously, she was a Research Engineer with STMicroelectronics, San Jose, CA, and an Industrial Visitor with the University of California, Berkeley. Her research interests include automating embedded processor customisation, high-performance compiler techniques, and reconfigurable computing.

Dr. Pozzi received a Best Paper Award in the embedded systems category at the Design Automation Conference in 2003, and she is on the Technical Program Committee of CASES 2005 (Conference in Compilers, Architectures and Software for Embedded Systems).

Paolo Ienne (S'90–M'96) received the Dottore in Ingegneria Elettronica degree from Politecnico di Milano, Milan, Italy, in 1991, and the Ph.D. degree from the Swiss Federal Institute of Technology Lausanne (EPFL), Lausanne, Switzerland, in 1996.

From 1990 to 1991, he was a Junior Researcher with Brunel University, Uxbridge, U.K. From 1992 to 1996, he was a Research Assistant with the Micro computing Laboratory (LAMI) and at the MANTRA Center for Neuro-Mimetic Systems of the EPFL. In December 1996, he joined the Semiconductors Group, Siemens AG, Munich, Germany (which later became Infineon Technologies AG). After working on datapath generation tools, he became Head of the Embedded Memory Unit in the Design Libraries Division. Since 2000, he has been a Professor with the EPFL and heads the Processor Architecture Laboratory (LAP). His research interests include various aspects of computer and processor architecture, reconfigurable computing, on-chip networks and multiprocessor systems-on-chip, and computer arithmetic.

Dr. Ienne was a recipient of the 40th Design Automation Conference Best Paper Award in 2003. He is or has also been a member of the program committees of several international workshops and conferences, including Design Automation and Test in Europe (DATE), the International Conference on Computer Aided Design (ICCAD), the International Symposium on High-Performance Computer Architecture (HPCA), the ACM International Conference on Supercomputing (ICS), the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), and the Workshop on Application-Specific Processors (WASP).